Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Essential Skills for Agile Development

Tong Ka Iok, Kent

Copyright (c) 2003-2004. Macau Productivity and Technology Transfer Center.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Essential Skills for Agile Development, Tong Ka Iok (author).

Copyright © 2003-2004 by the Macau Productivity and Technology Transfer Center.

All rights reserved. No part of this book may be reproduced or transmitted in any form without the written permission of the publisher.

Published in June 2004.

Java, JDBC, JavaServer Pages are trademarks or registered trademarks of SUN Microsystems. Other product names or company names mentioned in the book may be trademarks or registered trademarks of their respective owners.

Although we have made significant effort to minimize the errors in the book, we cannot provide any warranty that there is none. The author and the publisher will assume absolutely no responsibility for the accuracy or suitability of the information in the book to the extent that is permitted by applicable laws.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Foreword

Software development has been plagued by problems. Fortunately, at the same time innovations in programming techniques are continuously made in order to deliver quality software that meets customers' requirements within schedule and budget. Starting from around 1998, a new school of software development methodologies collectively known as "Agile Software Development" is starting to capture the minds and hearts of developers worldwide because it can effectively address the various problems surrounding software development.

In 2002, CPTTM, the Macau Productivity and Technology Center, started to hold courses on eXtreme Programming and OO design to promote the skills in agile development. At the beginning, they hired a well known software training and consulting company in US to teach and later switched to use local instructors. However, because software design principles are abstract by nature, the students did not learn very well and never really acquired the expected skills on completion of the courses.

According to his many years of experiences in software development and training, Tong Ka Iok, the manager of the IT department of CPTTM, decided to develop a new set of training materials, by selecting only the essential skills in Agile Development, while ignoring those useful but non-essential skills. He explained these skills and principles in terms of examples and added a lot of real world examples as exercises. After adopting this set of materials, with exactly the same instructor, the new students learned better and more easily and really acquired the skills on completion.

This publication "Essential Skills for Agile Development" reflects the many years of experiences of Tong Ka Iok in software development and training. It is an effective way to learn to apply the skills in Agile Development and is an indispensable book for software development in general.

More specifically, "Essential Skills for Agile Development" employs a lot of easy-to-understand examples to demonstrate some essential skills in software development (including how to handle duplicate code, comments, bloated code), how to develop to the requirements of the customer, how to perform TDD (including functional testing, user interface testing and unit testing), how to effectively use a database in Agile Development and more. The book is heavily based on examples and does not contain much abstract description. By following the examples, readers will be able to learn the skills and take them to practice in a snap.

How is this book different from others? In summary:

- 80/20 rule. The book only talks about that 20% of skills that determine 80% of the software quality.
- One skill at a time. Nobody can learn many different skills at the same time.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Therefore, this book talks about only one skill at a time and do not move on until the readers master it.

- Learn by doing. It is best to learn programming by programming. Therefore, this book includes a lot of exercises for the readers to practice to gain experience.

- Feedback. Programming needs feedbacks from others and self tests. Therefore, this book provides sample solutions to the exercises so that the readers can find out their strengths and weaknesses by comparing their solutions to sample ones.

- Reinforcement. Without practicing a newly learned programming skill frequently, one may have difficulty applying it later or may simply forget about it. Therefore, this book emphasizes repeated exercises that test the readers on the skills taught previously, so that those skills are reinforced.

Some say that writing a book is easy but writing course materials is difficult; Some say it is the opposite. The truth lies in the targeted audience. Who is the book written for? Who are the course materials written for? What is the targeted achievement in knowledge and what is the course evaluation? Without a doubt, this book can be used as training materials for software developers. It is also an ideal reference for anyone who enjoys programming in Java.

Tong Chi Kin
Spokesman of the Executive Council of the Macao Special Administrative Region
Aug. 26, 2004

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Preface

Who should read this book

Software developers who would like to learn more about Agile Development.

Prior knowledge required:

- Can read Java code.
- Can read and write SQL.
- Can read simple HTML code.
- Some basic concepts on Swing (e.g., JDialog, JButton, ActionListener), JDBC (e.g., Connection, ResultSet) and Servlet are desirable but not required.

How to read this book

The most important thing is that you do the exercises and compare your solutions to the sample solutions. Let me repeat: Do the exercises!

How to get in touch

The book has a web site: http://www.agileskills.org. If you find an error in the book or have any suggestions, please post it there. It also hosts the errata and a soft copy of the book for public download.

Acknowledgments

I'd like to thank:

- Don Wells for reviewing the book and giving lots of valuable suggestions. In particular, he made a few significant suggestions that I didn't like but turn out to add great values to the book. I am very grateful to have had such an expert reviewer.
- Tong Chi Kin, spokesman of the Executive Council of the Macao Special Administrative Region, for writing the foreword for the book.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- Brian Iu Ion Tong for reviewing the book. Brian has always been a good friend and an inspiring and caring mentor. I was surprised that he spent so much time and effort reviewing the book.

- Eric Yeung, Chairman of CPTTM, and Victor Manuel Kuan, Director General of CPTTM, for supporting the publication of this book.

- Helena Lei for giving suggestions on how to write about pair programming.

- Jack Wu Chan Keong and Simon Pau Chi Wai for discussing with me on various examples in the book.

- The participants of our OO instructor incubation scheme for contributing code examples: Antonio Man Ho Lo, Carita Ma Lai Peng, Carol U Man Leng, Eugen Chan Peng U, Franky Tou Io Kuok, Malaquias Man Kit Lo and YK Wong Io Kuan.

- Eugen Chan Peng U for the elegant graphical and layout design for the book.

Kent Ka Iok Tong

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Table of Contents

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 1

Removing Duplicate Code

How duplicate code may be created

Consider the following code:

```
public class BookRental {
   String id;
   String customerName;
   ...
}
public class BookRentals {
   private Vector rentals;
   public String getCustomerName(String rentalId) {
      for (int i = 0; i < rentals.size(); i++) {
         BookRental rental = (BookRental) rentals.elementAt(i);
         if (rental.getId().equals(rentalId)) {
            return rental.getCustomerName();
         }
      }
      throw new RentalNotFoundException();
   }
}
public class RentalNotFoundException extends Exception {
   ...
}
```

Suppose that you need to add a new method to delete a rental given its id. You figure that you need to search for the rental one by one, just like what you did in getCustomerName. So you copy and paste getCustomerName and modify the result to get the new method:

```
public class BookRentals {
    private Vector rentals;
   public String getCustomerName(String rentalId) {
      for (int i = 0; i < rentals.size(); i++) {
         BookRental rental = (BookRental) rentals.elementAt(i);
         if (rental.getId().equals(rentalId)) {
            return rental.getCustomerName();
         }
      }
      throw new RentalNotFoundException();
   }
    public void deleteRental(String rentalId) {
        for (int i = 0; i < rentals.size(); i++) {
            BookRental rental = (BookRental) rentals.elementAt(i);
            if (rental.getId().equals(rentalId)) {
                rentals.remove(i);
                return;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            }
        }
        throw new RentalNotFoundException();
    }
}
```

Does the code look fine? No, the two methods share quite a lot of code (code duplication).

Removing duplicate code

To remove the duplication, you can change the BookRentals class ("refactor" it):

```
public class BookRentals {
   private Vector rentals;
   public String getCustomerName(String rentalId) {
      int rentalIdx = getRentalIdxById(rentalId);
      return ((BookRental) rentals.elementAt(rentalIdx)).getCustomerName();
   }
   public void deleteRental(String rentalId) {
      rentals.remove(getRentalIdxById(rentalId));
   }
   private int getRentalIdxById(String rentalId) {
      for (int i = 0; i < rentals.size(); i++) {
         BookRental rental = (BookRental) rentals.elementAt(i);
         if (rental.getId().equals(rentalId)) {
            return i;
         }
      }
      throw new RentalNotFoundException();
   }
}
```

Why remove duplicate code

Why remove duplicate code? Suppose that you would like to change the "rentals" field from a vector into an array, you need to change "rentals.size()" to "rentals.length". In the refactored version you only need to change it once in getRentalIdxById, while in the original version you need to change it twice in getCustomerName and deleteRental. Similarly, you need to change "rentals.elementAt(i)" to "rentals[i]". In the refactored version you do it only once, while in the original version you do it twice.

In general, if the same code is duplicated in 10 places, when you need to change the code, you will have to find all these 10 places and change them all. If you forget to change some places, in this particular case (change from vector to array) the compiler will detect the errors, but in some other cases they will become bugs that will take a lot of your precious time to find.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

References

- http://c2.com/cgi/wiki?OnceAndOnlyOnce.
- http://www.martinfowler.com/ieeeSoftware/repetition.pdf.
- A. Hunt, and D. Thomas, Pragmatic Programmer, Addison Wesley, 1999.
- Kent Beck, Extreme Programming Explained, Addison-Wesley, 2000.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

1. The solutions should be in the format of source code in Java, C++ or Delphi. It does not have to be free of syntax errors as long as it can communicate the design clearly. If required, add some text explanations to justify the design.

2. For some problems, there are hints at the end of the problems. However, you should try to solve the problems yourself first! Look at the hints only if you are desperate.

3. After the hints, there are sample solutions. After working out your own solutions, you should compare them to yours.

Problems

1. Point out and remove the duplication in the code:

```
class Organization {
   String id;
   String eName; //English name
   String cName; //Chinese name
   String telCountryCode;
   String telAreaCode;
   String telLocalNumber;
   String faxCountryCode;
   String faxAreaCode;
   String faxLocalNumber;
   String contactPersonEFirstName; //First name and last name in English
   String contactPersonELastName;
   String contactPersonCFirstName; //First name and last name in Chinese
   String contactPersonCLastName;
   String contactPersonTelCountryCode;
   String contactPersonTelAreaCode;
   String contactPersonTelNumber;
   String contactPersonFaxCountryCode;
   String contactPersonFaxAreaCode;
   String contactPersonFaxLocalNumber;
   String contactPersonMobileCountryCode;
   String contactPersonMobileAreaCode;
   String contactPersonMobileLocalNumber;
   ...
}
```

2. Point out and remove the duplication in the code:

```
class ParticipantsInDB {
   Connection db;
   ParticipantsInDB(){
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      Class.forName("org.postgresql.Driver");
      db =
         DriverManager.getConnection(
            "jdbc:postgresql://myhost/ConferenceDB",
            "PaulChan",
            "myP@ssword");
   }
   void addParticipant(Participant part) {
      PreparedStatement st = db.prepareStatement("INSERT INTO participants
VALUES (?,?,?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         st.setString(2, part.getEFirstName());
         st.setString(3, part.getELastName());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
class OrganizationsInDB {
   Connection db;
   OrganizationsInDB() {
      Class.forName("org.postgresql.Driver");
      db =
         DriverManager.getConnection(
            "jdbc:postgresql://myhost/ConferenceDB",
            "PaulChan",
            "myP@ssword");
   }
   void addOrganization(Organization o) {
      PreparedStatement st =
         db.prepareStatement(
            "INSERT INTO organizations VALUES (?,?,?,?,?,?,?,?,?,?,)");
      try {
         st.setString(1, o.getId());
         st.setString(2, o.getEName());
         st.setString(3, o.getCName());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

3. Point out and remove the duplication in the code:

```
class ParticipantsInDB {
   Connection db;
   void addParticipant(Participant part) {
      PreparedStatement st = db.prepareStatement("INSERT INTO participants
VALUES (?,?,?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         ...
         st.executeUpdate();
      } finally {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
         st.close();
      }
   }
   void deleteAllParticipants() {
      PreparedStatement st = db.prepareStatement("DELETE FROM participants");
      try {
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   int getCount() {
      PreparedStatement st = db.prepareStatement("SELECT COUNT(*) FROM
participants");
      try {
         ResultSet rs = st.executeQuery();
         rs.next();
         return rs.getInt(1);
      } finally {
         st.close();
      }
   }
}
```

4. Point out and remove the duplication in the code:

```
class ParticipantsInDBTest extends TestCase {
   ParticipantsInDB p;
   void setUp() {
      p=ParticipantsInDB.getInstance();
   }
   void tearDown() {
      ParticipantsInDB.freeInstance();
   }
   void testAdd() {
      Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
      p.deleteAllParticipants();
      p.addParticipant(part1);
      assertEquals(p.getCount(),1);
   }
   void testAdd2() {
      Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
      Participant part2=new Participant("ABC003","Paul","Chan",true,"Manager");
      p.deleteAllParticipants();
      p.addParticipant(part1);
      p.addParticipant(part2);
      assertEquals(p.getCount(),2);
   }
   void testEnum() {
      Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
      Participant part2=new Participant("ABC003","Paul","Chan",true,"Manager");
      p.deleteAllParticipants();
      p.addParticipant(part2);
      p.addParticipant(part1);
      ParticipantEnumeratorById penum=new ParticipantEnumeratorById();
      assertTrue(penum.next());
      assertTrue(penum.get().equals(part1));
      assertTrue(penum.next());
      assertTrue(penum.get().equals(part2));
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      assertTrue(!penum.next());
      penum.close();
   }
}
```

5. Point out and remove the duplication in the code:

```
class ReportCatalogueIndexCommandParser {
   final String NO_GROUPING = "orgNoGrouping";
   static final int ORG_CATALOG = 0;
   static final int PART_CATALOG = 1;
   int getGroupingType(String grouping)      {
      if (grouping.equals(NO_GROUPING)) {
         return ORG_CATALOG;
      } else if (grouping.equals("orgGroupByCountry")) {
         return ORG_CATALOG;
      } else if (grouping.equals("orgGroupByTypeOfOrgName")) {
         return ORG_CATALOG;
      } else if (grouping.equals("part")) {
         return PART_CATALOG;
      } else
         throw new IllegalArgumentException("Invalid grouping!");
   }
   ...
}
```

6. Point out and remove the duplication in the code:

```
class Restaurant extends Account {
   //the string "Rest" is concated with the restaurant ID to
   //form the key.
   final static String RestaurantIDText = "Rest";
   String website;
   //address in Chinese.
   String addr_cn;
   //address in English.
   String addr_en;
   //the restaurant would like to update its fax # with this. After it is
   //confirmed, it will be stored in Account. Before that, it is stored
   //here.
   String numb_newfax;
   boolean has_NewFax = false;
   //a list of holidays.
   Vector Holiday; // a holiday
   //id of the category this restaurant belongs to.
   String catId;
   //a list of business session. Each business session is an array
   //of two times. The first time is the start time. The second time
   //is the end time. The restaurant is open for business in each
   //session.
   Vector BsHour; //Business hour
   ...
   //y: year.
   //m: month.
   //d: date.
   void addHoliday(int y,int m,int d){
      if(y<1900) y+=1900;
      Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      Holiday.add(aHoliday);
   }
   public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
      int fMin = fromHr*60 + fromMin;  //start time in minutes.
      int tMin = toHr*60 + toMin;   //end time in minutes.
      //make sure both times are valid and the start time is earlier
      //than the end time.
      if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
         GregorianCalendar bs[] = {
            new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
            new GregorianCalendar(1900,1,1, toHr, toMin,0)
         };
         BsHour.add(bs);
         return true;
      } else {
         return false;
      }
   }
}
```

7. Point out and remove the duplication in the code:

```
class Order {
   ...
   boolean IsSameString(String s1, String s2){
      if(s1==s2) return true;
      if(s1==null) return false;
      return(s1.equals(s2));
   }
}
class Mail {
   ...
   static boolean IsSameString(String s1, String s2) {
      if (s1 == s2)
         return true;
      if (s1 == null)
         return false;
      return (s1.equals(s2));
   }
}
```

8. Point out and remove the duplication in the code:

```
class FoodDB {
   //find all foods whose names contain both the keywords. returns
   //an iterator on these foods.
   public Iterator srchFood(String cName, String eName){
      //it contains all the foods to be returned.
      TreeMap foodTree = new TreeMap();
      Iterator foodList;
      Food food;
      foodList = getAllRecords();
      while (foodList.hasNext()){
         food = (Food) foodList.next();
         //do its names contain both keywords?
         if ((cName==null || //null or "" means no key is specified
            cName.equals("") ||
            food.getCName().indexOf(cName)!=-1)
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            &&
            (eName==null || //null or "" means no key is specified
            eName.equals("") ||
            food.getEName().indexOf(eName)!=-1)){
            foodTree.put(food.getFoodKey(),food);
         }
      }
      return foodTree.values().iterator();
   }
}
```

9. Point out and remove the duplication in the code:

```
class UIDialogCustomerMain extends JDialog {
   JButton btnOrderDel;
   JButton btnCustChangePassword;
   void bindEvents() {
      ...
      btnOrderDel.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIDialogCustomerDeleteOrder(UIDialogCustomerMain.this,
"Del Order", true);
            d.pack();
            d.setLocation(400,200);
            d.setVisible(true);
         }
      });
      btnCustChangePassword.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIChangeAccountPW(UIDialogCustomerMain.this, "chg pw",
true);
            d.pack();
            d.setLocation(400,200);
            d.setVisible(true);
         }
      });
   }
   ...
}
class UIDialogRestaurantMain extends JDialog {
   JButton btnChangePassword;
   void bindEvents() {
      ...
   btnChangePassword.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIChangeAccountPW(UIDialogRestaurantMain.this, "chg
pw", true);
            d.pack();
            d.setLocation(400,200);
            d.setVisible(true);
         }
      });
   }
   ...
}
```

10. Suppose that there are two kinds of rentals: book and movie. Point out and remove the duplication in the code:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
public class BookRental {
   private String bookTitle;
   private String author;
   private Date rentDate;
   private Date dueDate;
   private double rentalFee;
   public boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   public double getTotalFee() {
      return isOverdue() ? 1.2*rentalFee : rentalFee;
   }
}
public class MovieRental {
   private String movieTitle;
   private int classification;
   private Date rentDate;
   private Date dueDate;
   private double rentalFee;
   public boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   public double getTotalFee() {
      return isOverdue() ? 1.3*rentalFee : rentalFee;
   }
}
```

11. Come up with some code that contains duplicate code/structure, point out the duplication and eliminate it.

Hints

1. The triple (country code, area code, local number) appears several times. It should be extracted into a class such as "TelNo". The word "contactPerson" is duplicated in many fields. The fields should be extracted into a class such as "ContactPerson".

2. The code to load the PostgreSQL JDBC driver and obtain a connection is duplicated in both classes. The code should be extracted into a class such as "ConferenceDBConnection".

3. The table name (i.e., "participants") is duplicated in the three methods. It should be extracted into a static final String in the class or into a class such as "ParticipantsTable".

4. The participant objects are duplicate. They should be extracted to become instance variables.

5. Each branch of if-then is doing similar thing. You may extract the strings used in the conditions into an array or a map. Then just do a lookup instead of multiple if-then's.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Point out and remove the duplication in the code:

```
class Organization {
   String id;
   String eName; //English name
   String cName; //Chinese name
   String telCountryCode;
   String telAreaCode;
   String telLocalNumber;
   String faxCountryCode;
   String faxAreaCode;
   String faxLocalNumber;
   String contactPersonEFirstName; //First name and last name in English
   String contactPersonELastName;
   String contactPersonCFirstName; //First name and last name in Chinese
   String contactPersonCLastName;
   String contactPersonTelCountryCode;
   String contactPersonTelAreaCode;
   String contactPersonTelNumber;
   String contactPersonFaxCountryCode;
   String contactPersonFaxAreaCode;
   String contactPersonFaxLocalNumber;
   String contactPersonMobileCountryCode;
   String contactPersonMobileAreaCode;
   String contactPersonMobileLocalNumber;
   ...
}
```

Turn the code into:

```
class Organization {
   String id;
   String eName;
   String cName;
   TelNo telNo;
   TelNo faxNo;
   ContactPerson contactPerson;
   ...
}
class ContactPerson{
   String eFirstName;
   String eLastName;
   String cFirstName;
   String cLastName;
   TelNo tel;
   TelNo fax;
   TelNo mobile;
}
class TelNo {
   String countryCode;
   String areaCode;
   String localNumber;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

If you are worried that the pair (FirstName, LastName) appears twice, you may further extract the pair into a class:

```
class ContactPerson{
   FullName eFullName;
   FullName cFullName;
   TelNo tel;
   TelNo fax;
   TelNo mobile;
}
class FullName {
   String firstName;
   String lastName;
}
```

2. Point out and remove the duplication in the code:

```
class ParticipantsInDB {
   Connection db;
   ParticipantsInDB(){
      Class.forName("org.postgresql.Driver");
      db =
         DriverManager.getConnection(
            "jdbc:postgresql://myhost/ConferenceDB",
            "PaulChan",
            "myP@ssword");
   }
   void addParticipant(Participant part) {
      PreparedStatement st = db.prepareStatement("INSERT INTO participants VALUES
(?,?,?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         st.setString(2, part.getEFirstName());
         st.setString(3, part.getELastName());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
class OrganizationsInDB {
   Connection db;
   OrganizationsInDB() {
      Class.forName("org.postgresql.Driver");
      db =
         DriverManager.getConnection(
            "jdbc:postgresql://myhost/ConferenceDB",
            "PaulChan",
            "myP@ssword");
   }
   void addOrganization(Organization o) {
      PreparedStatement st =
         db.prepareStatement(
            "INSERT INTO organizations VALUES (?,?,?,?,?,?,?,?,?,?,)");
      try {
         st.setString(1, o.getId());
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
         st.setString(2, o.getEName());
         st.setString(3, o.getCName());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

The code to setup the DB connection is duplicated. Extract it into a separate class:

```
class ConferenceDBConnection {
   static Connection makeConnection() {
      Class.forName("org.postgresql.Driver");
      return
         DriverManager.getConnection(
            "jdbc:postgresql://myhost/ConferenceDB",
            "PaulChan",
            "myP@ssword");
   }
}
class ParticipantsInDB {
   Connection db;
   ParticipantsInDB(){
      db = ConferenceDBConnection.makeConnection();
   }
   void addParticipant(Participant part) {
      PreparedStatement st = db.prepareStatement("INSERT INTO participants
VALUES (?,?,?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         st.setString(2, part.getEFirstName());
         st.setString(3, part.getELastName());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
class OrganizationsInDB {
   Connection db;
   OrganizationsInDB() {
      db = ConferenceDBConnection.makeConnection();
   }
   void addOrganization(Organization o) {
      PreparedStatement st =
         db.prepareStatement(
            "INSERT INTO organizations VALUES (?,?,?,?,?,?,?,?,?,?,)");
      try {
         st.setString(1, o.getId());
         st.setString(2, o.getEName());
         st.setString(3, o.getCName());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
}
```

3. Point out and remove the duplication in the code:

```
class ParticipantsInDB {
   Connection db;
   void addParticipant(Participant part) {
      PreparedStatement st = db.prepareStatement("INSERT INTO participants VALUES
(?,?,?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteAllParticipants() {
      PreparedStatement st = db.prepareStatement("DELETE FROM participants");
      try {
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   int getCount() {
      PreparedStatement st = db.prepareStatement("SELECT COUNT(*) FROM
participants");
      try {
         ResultSet rs = st.executeQuery();
         rs.next();
         return rs.getInt(1);
      } finally {
         st.close();
      }
   }
}
```

The table name is duplicated. The call to "db.preparedStatement" is duplicated.

```
class ParticipantsInDB {
   static final String tableName="participants";
   Connection db;
   PreparedStatement makeStatement(String sql) {
      return  db.prepareStatement(sql);
   }
   void addParticipant(Participant part) {
      PreparedStatement st = makeStatement("INSERT INTO "+tableName+" VALUES
(?,?,?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         ...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void deleteAllParticipants() {
      PreparedStatement st = makeStatement("DELETE FROM "+tableName);
      try {
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   int getCount() {
      PreparedStatement st = makeStatement("SELECT COUNT(*) FROM "+tableName);
      try {
         ResultSet rs = st.executeQuery();
         rs.next();
         return rs.getInt(1);
      } finally {
         st.close();
      }
   }
}
```

4. Point out and remove the duplication in the code:

```
class ParticipantsInDBTest extends TestCase {
   ParticipantsInDB p;
   void setUp() {
      p = ParticipantsInDB.getInstance();
   }
   void tearDown() {
      ParticipantsInDB.freeInstance();
   }
   void testAdd() {
      Participant part1 =
         new Participant("ABC001", "Kent", "Tong", true, "Manager");
      p.deleteAllParticipants();
      p.addParticipant(part1);
      assertEquals(p.getCount(), 1);
   }
   void testAdd2() {
      Participant part1 =
         new Participant("ABC001", "Kent", "Tong", true, "Manager");
      Participant part2 =
         new Participant("ABC003", "Paul", "Chan", true, "Manager");
      p.deleteAllParticipants();
      p.addParticipant(part1);
      p.addParticipant(part2);
      assertEquals(p.getCount(), 2);
   }
   void testEnum() {
      Participant part1 =
         new Participant("ABC001", "Kent", "Tong", true, "Manager");
      Participant part2 =
         new Participant("ABC003", "Paul", "Chan", true, "Manager");
      p.deleteAllParticipants();
      p.addParticipant(part2);
      p.addParticipant(part1);
      ParticipantEnumeratorById penum = new ParticipantEnumeratorById();
      assertTrue(penum.next());
      assertTrue(penum.get().equals(part1));
      assertTrue(penum.next());
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      assertTrue(penum.get().equals(part2));
      assertTrue(!penum.next());
      penum.close();
   }
}
```

The two participant objects are duplicate.

```
class ParticipantsInDBTest extends TestCase {
   ParticipantsInDB p;
   Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
   Participant part2=new Participant("ABC003","Paul","Chan",true,"Manager");
   void setUp() {
      p=ParticipantsInDB.getInstance();
   }
   void tearDown() {
      ParticipantsInDB.freeInstance();
   }
   void testAdd() {
      p.deleteAllParticipants();
      p.addParticipant(part1);
      assertEquals(p.getCount(),1);
   }
   void testAdd2() {
      p.deleteAllParticipants();
      p.addParticipant(part1);
      p.addParticipant(part2);
      assertEquals(p.getCount(),2);
   }
   void testEnum() {
      p.deleteAllParticipants();
      p.addParticipant(part2);
      p.addParticipant(part1);
      ParticipantEnumeratorById penum=new ParticipantEnumeratorById();
      assertTrue(penum.next());
      assertTrue(penum.get().equals(part1));
      assertTrue(penum.next());
      assertTrue(penum.get().equals(part2));
      assertTrue(!penum.next());
      penum.close();
   }
}
```

5. Point out and remove the duplication in the code:

```
class ReportCatalogueIndexCommandParser {
   final String NO_GROUPING = "orgNoGrouping";
   static final int ORG_CATALOG = 0;
   static final int PART_CATALOG = 1;
   int getGroupingType(String grouping) {
      if (grouping.equals(NO_GROUPING)) {
         return ORG_CATALOG;
      } else if (grouping.equals("orgGroupByCountry")) {
         return ORG_CATALOG;
      } else if (grouping.equals("orgGroupByTypeOfOrgName")) {
         return ORG_CATALOG;
      } else if (grouping.equals("part")) {
         return PART_CATALOG;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      } else
         throw new IllegalArgumentException("Invalid grouping!");
   }
   ...
}
```

Each branch of if-then is doing a similar thing. Extract the strings used in the conditions into a set.

```
class ReportCatalogueIndexCommandParser {
   final String NO_GROUPING = "orgNoGrouping";
   static final int ORG_CATALOG = 0;
   static final int PART_CATALOG = 1;

   int getGroupingType(String grouping) {
      Set orgGroupings = new TreeSet();
      orgGroupings.add(NO_GROUPING);
      orgGroupings.add("orgGroupByCountry");
      orgGroupings.add("orgGroupByTypeOfOrgName");
      if (orgGroupings.contains(grouping)) {
         return ORG_CATALOG;
      }
      if (grouping.equals("part")) {
         return PART_CATALOG;
      } else
         throw new IllegalArgumentException("Invalid grouping!");
   }
}
```

Using a set may be an overkill. A simpler way is to just extract the return statement:

```
class ReportCatalogueIndexCommandParser {
   final String NO_GROUPING = "orgNoGrouping";
   static final int ORG_CATALOG = 0;
   static final int PART_CATALOG = 1;
   int getGroupingType(String grouping)      {
      if (grouping.equals(NO_GROUPING) ||
         grouping.equals("orgGroupByCountry") ||
         grouping.equals("orgGroupByTypeOfOrgName")) {
         return ORG_CATALOG;
      } else if (grouping.equals("part")) {
         return PART_CATALOG;
      } else
         throw new IllegalArgumentException("Invalid grouping!");
   }
   ...
}
```

6. Point out and remove the duplication in the code:

```
class Restaurant extends Account {
   //the string "Rest" is concated with the restaurant ID to
   //form the key.
   final static String RestaurantIDText = "Rest";
   String website;
   //address in Chinese.
   String addr_cn;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   //address in English.
   String addr_en;
   //the restaurant would like to update its fax # with this. After it is
   //confirmed, it will be stored in Account. Before that, it is stored
   //here.
   String numb_newfax;
   boolean has_NewFax = false;
   //a list of holidays.
   Vector Holiday; // a holiday
   //id of the category this restaurant belongs to.
   String catId;
   //a list of business session. Each business session is an array
   //of two times. The first time is the start time. The second time
   //is the end time. The restaurant is open for business in each
   //session.
   Vector BsHour; //Business hour
   ...
   //y: year.
   //m: month.
   //d: date.
   void addHoliday(int y,int m,int d){
      if(y<1900) y+=1900;
      Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
      Holiday.add(aHoliday);
   }
   public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
      int fMin = fromHr*60 + fromMin;  //start time in minutes.
      int tMin = toHr*60 + toMin;   //end time in minutes.
      //make sure both times are valid and the start time is earlier
      //than the end time.
      if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
         GregorianCalendar bs[] = {
            new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
            new GregorianCalendar(1900,1,1, toHr, toMin,0)
         };
         BsHour.add(bs);
         return true;
      } else {
         return false;
      }
   }
}
```

The code to convert from hours plus minutes into just minutes is duplicated. The code to validate the number of minutes is duplicated. The code to create a GregorianCalendar on January 1 in year 1900 is duplicated.

```
class Restaurant extends Account {
   ...
   int getMinutesFromMidNight(int hours, int minutes) {
      return hours*60+minutes;
   }
   boolean isMinutesWithinOneDay(int minutes) {
      return minutes>0 && minutes<=1440;
   }
   GregorianCalendar convertTimeToDate(int hours, int minutes) {
      return new GregorianCalendar(1900, 1, 1, hours, minutes, 0);
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
      int fMin = getMinutesFromMidNight(fromHr, fromMin);
      int tMin = getMinutesFromMidNight(toHr, toMin);
      if (isMinutesWithinOneDay(fMin) &&
         isMinutesWithinOneDay(tMin) &&
         fMin < tMin){
         GregorianCalendar bs[] = {
            convertTimeToDate(fromHr, fromMin),
            convertTimeToDate(toHr, toMin)
         };
         BsHour.add(bs);
         return true;
      } else {
         return false;
      }
   }
}
```

7. Point out and remove the duplication in the code:

```
class Order {
   ...
   boolean IsSameString(String s1, String s2){
      if(s1==s2) return true;
      if(s1==null) return false;
      return(s1.equals(s2));
   }
}
class Mail {
   ...
   static boolean IsSameString(String s1, String s2) {
      if (s1 == s2)
         return true;
      if (s1 == null)
         return false;
      return (s1.equals(s2));
   }
}
```

The IsSameString method is duplicated.

```
class StringComparer {
   static boolean IsSameString(String s1, String s2) {
      if (s1 == s2)
         return true;
      if (s1 == null)
         return false;
      return (s1.equals(s2));
   }
}
class Order {
   ...
   //If possible, remove this method and let the other code call
   //StringComparer.IsSameString directly.
   boolean IsSameString(String s1, String s2){
      return StringComparer.IsSameString(s1, s2);
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class Mail {
   ...
   //If possible, remove this method and let the other code call
   //StringComparer.IsSameString directly.
   static boolean IsSameString(String s1, String s2) {
      return StringComparer.IsSameString(s1, s2);
   }
}
```

8. Point out and remove the duplication in the code:

```
class FoodDB {
   //find all foods whose names contain both the keywords. returns
   //an iterator on these foods.
   public Iterator srchFood(String cName, String eName){
      //it contains all the foods to be returned.
      TreeMap foodTree = new TreeMap();
      Iterator foodList;
      Food food;
      foodList = getAllRecords();
      while (foodList.hasNext()){
         food = (Food) foodList.next();
         //do its names contain both keywords?
         if ((cName==null || //null or "" means no key is specified
            cName.equals("") ||
            food.getCName().indexOf(cName)!=-1)
            &&
            (eName==null || //null or "" means no key is specified
            eName.equals("") ||
            food.getEName().indexOf(eName)!=-1)){
            foodTree.put(food.getFoodKey(),food);
         }
      }
      return foodTree.values().iterator();
   }
}
```

The checking of whether the name contains a keyword is duplicated.

```
class FoodDB {
   boolean nameMatchKeyword(String name, String keyword) {
      return keyword==null ||
            keyword.equals("") ||
            name.indexOf(keyword)!=-1;
   }
   public Iterator srchFood(String cName, String eName){
      TreeMap foodTree = new TreeMap();
      Iterator foodList;
      Food food;
      foodList = getAllRecords();
      while (foodList.hasNext()){
         food = (Food) foodList.next();
         if (nameMatchKeyword(food.getCName(), cName)
            &&
            nameMatchKeyword(food.getEName(), eName)) {
            foodTree.put(food.getFoodKey(),food);
         }
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      return foodTree.values().iterator();
   }
}
```

9. Point out and remove the duplication in the code:

```
class UIDialogCustomerMain extends JDialog {
   JButton btnOrderDel;
   JButton btnCustChangePassword;
   void bindEvents() {
      ...
      btnOrderDel.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIDialogCustomerDeleteOrder(UIDialogCustomerMain.this,
"Del Order", true);
            d.pack();
            d.setLocation(400,200);
            d.setVisible(true);
         }
      });
      btnCustChangePassword.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIChangeAccountPW(UIDialogCustomerMain.this, "chg pw",
true);
            d.pack();
            d.setLocation(400,200);
            d.setVisible(true);
         }
      });
   }
   ...
}
class UIDialogRestaurantMain extends JDialog {
   JButton btnChangePassword;
   void bindEvents() {
      ...
   btnChangePassword.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIChangeAccountPW(UIDialogRestaurantMain.this, "chg
pw", true);
            d.pack();
            d.setLocation(400,200);
            d.setVisible(true);
         }
      });
   }
   ...
}
```

The code packing the dialog, setting its location and showing it is duplicated. The change password button is also duplicated.

```
class UIDialogShower {
   public static void show(Dialog d, int left, int top) {
      d.pack();
      d.setLocation(left, top);
      d.setVisible(true);
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public static void showAtDefaultPosition(Dialog d) {
      show(d, 400, 200);
   }
}
class ChangePasswordButton extends JButton {
   public ChangePasswordButton(final JDialog enclosingDialog) {
      addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIChangeAccountPW(enclosingDialog, "chg pw", true);
            UIDialogShower.showAtDefaultPosition(d);
         }
      });
   }
}
class UIDialogCustomerMain extends JDialog {
   JButton btnOrderDel;
   ChangePasswordButton btnCustChangePassword = new ChangePasswordButton(this);
   void bindEvents() {
      ...
      btnOrderDel.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Dialog d = new UIDialogCustomerDeleteOrder
                      (UIDialogCustomerMain.this, "Del Order", true);
            UIDialogShower.showAtDefaultPosition(d);
         }
      });
   }
   ...
}
class UIDialogRestaurantMain extends JDialog {
   ChangePasswordButton btnChangePassword = new ChangePasswordButton(this);
   void bindEvents() {
      ...
   }
   ...
}
```

10. Suppose that there are two kinds of rentals: book and movie. Point out and remove the duplication in the code:

```
public class BookRental {
   private String bookTitle;
   private String author;
   private Date rentDate;
   private Date dueDate;
   private double rentalFee;
   public boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   public double getTotalFee() {
      return isOverdue() ? 1.2*rentalFee : rentalFee;
   }
}
public class MovieRental {
   private String movieTitle;
   private int classification;
   private Date rentDate;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   private Date dueDate;
   private double rentalFee;
   public boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   public double getTotalFee() {
      return isOverdue() ? 1.3*rentalFee : rentalFee;
   }
}
```

Some fields such as rentDate, dueDate and etc. are duplicated. The isOverdue method is duplicated. The structure of the getTotalFee method is duplicated.

```
public abstract class Rental {
   private Date rentDate;
   private Date dueDate;
   private double rentalFee;
   protected abstract double getOverduePenaltyRate();
   public boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   public double getTotalFee() {
      return isOverdue() ? getOverduePenaltyRate()*rentalFee : rentalFee;
   }
}
public class BookRental extends Rental {
   private String bookTitle;
   private String author;
   protected double getOverduePenaltyRate() {
      return 1.2;
   }
}
public class MovieRental extends Rental {
   private String movieTitle;
   private int classification;
   protected double getOverduePenaltyRate() {
      return 1.3;
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 2

Turning Comments into Code

Example

This is a conference management application. In the conference, every participant will wear a badge. On the badge there is some information of the participant (e.g., name and etc.). In the application the Badge class below is used to store this information. Please read the code and comments below:

```
//It stores the information of a participant to be printed on his badge.
public class Badge {
   String pid;  //participant ID
   String engName; //participant's full name in English
   String chiName; //participant's full name in Chinese
   String engOrgName; //name of the participant's organization in English
   String chiOrgName; //name of the participant's organization in Chinese
   String engCountry; //the organization's country in English
   String chiCountry; //the organization's country in Chinese

   //***********************
   //constructor.
   //The participant ID is provided. It then loads all the info from the DB.
   //***********************
   Badge(String pid) {
      this.pid = pid;
      //***********************
      //get the participant's full names.
      //***********************
      ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
      Participant part = partsInDB.locateParticipant(pid);
      if (part != null) {
         //get the participant's full name in English.
         engName = part.getELastName() + ", " + part.getEFirstName();
         //get the participant's full name in Chinese.
         chiName = part.getCLastName()+part.getCFirstName();
         //***********************
         //get the organization's name and country.
         //***********************
         OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
         //find the ID of the organization employing this participant.
         String oid = orgsInDB.getOrganization(pid);
         if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engCountry = org.getEAddress().getCountry();
            chiCountry = org.getCAddress().getCountry();
         }
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
    ...
}
```

Turn comments into code, making the code as clear as the comments

Consider the first comment:

```
//It stores the information of a participant to be printed on his badge.
public class Badge {
    ...
}
```

Why do we need this comment? Because the programmer thinks the name "Badge" is not clear enough, so he writes this comment to complement this insufficiency. However, if we can use this comment directly as the name of the class, the name will be almost as clear as the comment, then we will not need this separate comment anymore, e.g.:

```
public class ParticipantInfoOnBadge {
    ...
}
```

Why do that? Isn't writing comments a good programming style? Before the explanation, let's see how to turn the other comments in the above example into code.

Turn comments into variable names

Consider:

```
public class ParticipantInfoOnBadge {
    String pid;  //participant ID
    String engName; //participant's full name in English
    String chiName; //participant's full name in Chinese
    String engOrgName; //name of the participant's organization in English
    String chiOrgName; //name of the participant's organization in Chinese
    String engCountry; //the organization's country in English
    String chiCountry; //the organization's country in Chinese
    ...
}
```

We can turn the comments into variables, then delete the separate comments, e.g.:

```
public class ParticipantInfoOnBadge {
    String participantId;
    String participantEngFullName;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   String participantChiFullName;
   String engOrgName;
   String chiOrgName;
   String engOrgCountry;
   String chiOrgCountry;
   ...
}
```

Turn comments into parameter names

Consider:

```
public class ParticipantInfoOnBadge {
   ...
   //***********************
   //constructor.
   //The participant ID is provided. It then loads all the info from the DB.
   //***********************
   ParticipantInfoOnBadge(String pid) {
      this.pid = pid;
      ...
   }
}
```

We can turn the comments into parameter names, then delete the separate comments, e.g.:

```
public class ParticipantInfoOnBadge {
   ...
   //***********************
   //constructor.
   //It loads all the info from the DB.
   //***********************
   ParticipantInfoOnBadge(String participantId) {
      this.participantId = participantId;
      ...
   }
}
```

Turn comments into a part of a method body

How to get rid of the comment "It loads all the info from the DB" in the above example? It describes how the constructor of ParticipantInfoOnBadge is implemented (load the information from the database), therefore, we can turn it into a part of the body of the constructor, then delete it:

```
public class ParticipantInfoOnBadge {
   ...
   //***********************
   //constructor.
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    //***********************
    ParticipantInfoOnBadge(String participantId) {
       loadInfoFromDB(participantId);
    }
    void loadInfoFromDB(String participantId) {
       this.participantId = participantId;
       ...
    }
}
```

Delete useless comments

Sometimes we may come across some comments that are obviously useless, e.g.:

```
public class ParticipantInfoOnBadge {
   ...
   //***********************
   //constructor.
   //***********************
   ParticipantInfoOnBadge(String participantId) {
      ...
   }
}
```

This comment is useless because even without it anyone can tell that this is a constructor. It not only is useless, but also takes up the precious visual space: A screen can display at most 20 and odds lines, but this useless comment already takes up 3 lines, squeezing out the useful information (e.g., code), making this code fragment hard to understand. Therefore, we should delete it as quickly as possible:

```
public class ParticipantInfoOnBadge {
   ...
   ParticipantInfoOnBadge(String participantId) {
      ...
   }
}
```

Extract some code to form a method and use the comment to name the method

Consider the first comment below:

```
    void loadInfoFromDB(String participantId) {
       this.participantId = participantId;
       //***********************
       //get the participant's full names.
       //***********************
       ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   Participant part = partsInDB.locateParticipant(participantId);
   if (part != null) {
      //get the participant's full name in English.
      engFullName = part.getELastName() + ", " + part.getEFirstName();
      //get the participant's full name in Chinese.
      chiFullName = part.getCLastName()+part.getCFirstName();
      //***********************
      //get the organization's name and country.
      //***********************
      OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
      //find the ID of the organization employing this participant.
      String oid = orgsInDB.getOrganization(participantId);
      if (oid != null) {
         Organization org = orgsInDB.locateOrganization(oid);
         engOrgName = org.getEName();
         chiOrgName = org.getCName();
         engOrgCountry = org.getEAddress().getCountry();
         chiOrgCountry = org.getCAddress().getCountry();
      }
   }
}
```

This comment says that the code fragment following it will get the full name of the participant. In order to make the code fragment as clear as this comment, we can extract the code fragment into a method and use this comment to name the method, making this separate comment no longer necessary:

```
void loadInfoFromDB(String participantId) {
   this.participantId = participantId;
   getParticipantFullNames();
   //***********************
   //get the organization's name and country.
   //***********************
   //find the ID of the organization employing this participant.
   OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
   String oid = orgsInDB.getOrganization(participantId);
   if (oid != null) {
      Organization org = orgsInDB.locateOrganization(oid);
      engOrgName = org.getEName();
      chiOrgName = org.getCName();
      engOrgCountry = org.getEAddress().getCountry();
      chiOrgCountry = org.getCAddress().getCountry();
   }
}
void getParticipantFullNames() {
   ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
   Participant part = partsInDB.locateParticipant(participantId);
   if (part != null) {
      //get the participant's full name in English.
      engFullName = part.getELastName() + ", " + part.getEFirstName();
      //get the participant's full name in Chinese.
      chiFullName = part.getCLastName()+part.getCFirstName();
   }
}
```

Likewise, the code fragment to get the information of the organization of the participant can be extracted into a method and be named by the comment, making the comment unnecessary:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void loadInfoFromDB(String participantId) {
      this.participantId = participantId;
      getParticipantFullNames();
      getOrgNameAndCountry();
   }
   void getParticipantFullNames() {
      ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
      Participant part = partsInDB.locateParticipant(participantId);
      if (part != null) {
         //get the participant's full name in English.
         engFullName = part.getELastName() + ", " + part.getEFirstName();
         //get the participant's full name in Chinese.
         chiFullName = part.getCLastName()+part.getCFirstName();
      }
   }
   void getOrgNameAndCountry() {
      OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
      //find the ID of the organization employing this participant.
      String oid = orgsInDB.getOrganization(participantId);
      if (oid != null) {
         Organization org = orgsInDB.locateOrganization(oid);
         engOrgName = org.getEName();
         chiOrgName = org.getCName();
         engOrgCountry = org.getEAddress().getCountry();
         chiOrgCountry = org.getCAddress().getCountry();
      }
   }
```

An extracted method can be put into another class

Consider the two comments below:

```
public class ParticipantInfoOnBadge {
   ...
   void getParticipantFullNames() {
      ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
      Participant part = partsInDB.locateParticipant(participantId);
      if (part != null) {
         //get the participant's full name in English.
         engFullName = part.getELastName() + ", " + part.getEFirstName();
         //get the participant's full name in Chinese.
         chiFullName = part.getCLastName()+part.getCFirstName();
      }
   }
}
```

As the programmer thinks these code fragments not clear enough, then he should extract them and use the comments to name them. But this time the extracted methods should be put into the Participant class instead of the ParticipantInfoOnBadge class:

```
public class ParticipantInfoOnBadge {
   ...
   void getParticipantFullNames() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
      Participant part = partsInDB.locateParticipant(participantId);
      if (part != null) {
         engFullName = part.getEFullName();
         chiFullName = part.getCFullName();
      }
   }
}
public class Participant {
   String getEFullName() {
      return getELastName() + ", " + getEFirstName();
   }
   String getCFullName() {
      return getCLastName() + getCFirstName();
   }
}
```

Use a comment to name an existing method

Consider the comment below:

```
public class ParticipantInfoOnBadge {
   ...
   void getOrgNameAndCountry() {
      OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
      //find the ID of the organization employing this participant.
      String oid = orgsInDB.getOrganization(participantId);
      if (oid != null) {
         Organization org = orgsInDB.locateOrganization(oid);
         engOrgName = org.getEName();
         chiOrgName = org.getCName();
         engOrgCountry = org.getEAddress().getCountry();
         chiOrgCountry = org.getCAddress().getCountry();
      }
   }
}
```

We need the comment of "find the ID of the organization employing..." only because the name "getOrganization" is not clear enough, so, we should directly use the comment as the name:

```
public class ParticipantInfoOnBadge {
   ...
   void getOrgNameAndCountry() {
      OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
      String oid = orgsInDB.findOrganizationEmploying(participantId);
      if (oid != null) {
         Organization org = orgsInDB.locateOrganization(oid);
         engOrgName = org.getEName();
         chiOrgName = org.getCName();
         engOrgCountry = org.getEAddress().getCountry();
         chiOrgCountry = org.getCAddress().getCountry();
      }
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
public class OrganizationsInDB {
   ...
   void findOrganizationEmploying(String participantId) {
      ...
   }
}
```

The improved code

The improved code is shown below (all the comments have been turned into code and no longer exist separately):

```
public class ParticipantInfoOnBadge {
   String participantId;
   String participantEngFullName;
   String participantChiFullName;
   String engOrgName;
   String chiOrgName;
   String engOrgCountry;
   String chiOrgCountry;

   ParticipantInfoOnBadge(String participantId) {
      loadInfoFromDB(participantId);
   }
   void loadInfoFromDB(String participantId) {
      this.participantId = participantId;
      getParticipantFullNames();
      getOrgNameAndCountry();
   }
   void getParticipantFullNames() {
      ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
      Participant part = partsInDB.locateParticipant(participantId);
      if (part != null) {
         participantEngFullName = part.getEFullName();
         participantChiFullName = part.getCFullName();
      }
   }
   void getOrgNameAndCountry() {
      OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
      String oid = orgsInDB.findOrganizationEmploying(participantId);
      if (oid != null) {
         Organization org = orgsInDB.locateOrganization(oid);
         engOrgName = org.getEName();
         chiOrgName = org.getCName();
         engOrgCountry = org.getEAddress().getCountry();
         chiOrgCountry = org.getCAddress().getCountry();
      }
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Why delete the separate comments?

Why delete the separate comments? In fact, comments by themselves are not bad. The problem is that we often do not write clear code (because it is hard), so we take a shortcut (use comments) to hide the problem. The result is, nobody will try to make the code clearer. Later, as the code is updated, commonly nobody updates the comments accordingly. In time, opposed to making the code easier to read, these outdated comments will actually mislead the readers. At the end of the day, what we have is: Some code that is unclear by itself, mixed with some incorrect comments.

Therefore, whenever we see a comment or would like to write one, we should think twice: Can the comment be turned into code, making the code as clear as the comment? You will find that in most of the time the answer is yes. That is, every comment in the code is a good opportunity for us to improve our code. To say in another way, if the code includes a lot of comments, it probably means that the code quality is not that high (however, including a few or no comments does not necessarily mean the code quality is high).

Method name is too long

Consider the example below:

```
class StockItemsInDB {
   //find all the stock items from overseas whose inventory is smaller than 10.
   StockItem[] findStockItems() {
      ...
   }
}
```

In order to turn this comment into code, in principle we should change the code like this:

```
class StockItemsInDB {
   StockItem[] findStockItemsFromOverseasWithInventoryLessThan10() {
      ...
   }
}
```

However, this method name is too long, warning us that the code has problems. What should we do? We should determine: Is the customer of this system really only interested in those stock items from overseas and whose inventory are less than 10? Would he be interested in those stock items from overseas and whose inventory are less than 20? Would he be interested in those stock items from local and whose inventory are greater than 25? If yes, we can turn the comment into parameters:

```
class StockItemsInDB {
   StockItem[] findStockItemsWithFeatures(
      boolean isFromOverseas,
      InventoryRange inventoryRange) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        ...
    }
}
class InventoryRange {
    int minimumInventory;
    int maximumInventory;
}
```

If the customer is really only interested in those stock items from overseas and whose inventory are less than 10, he must have some particular reason (why only those but not the others?). After further conversation he may say it is because he needs to replenish the stock, because the shipping from overseas takes longer. Therefore, we find out that what he is really interested is the stock items that need replenishing, instead of those from overseas and whose inventory are less than 10. Therefore, we can turn the real purpose into the method name and turn the comment into the method body:

```
class StockItemsInDB {
    StockItem[] findStockItemsToReplenish() {
        StockItem stockItems[];
        stockItems = findStockItemsFromOverseas();
        stockItems = findStockItemsInventoryLessThan10(stockItems);
        return stockItems;
    }
}
```

References

- http://c2.com/cgi/wiki?TreatCommentsWithSuspicion.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Problems

1. Turn the comments into code:

```
class InchToPointConvertor {
   //convert the quantity in inches to points.
   static float parseInch(float inch) {
      return inch * 72;  //one inch contains 72 points.
   }
}
```

2. Turn the comments into code:

```
class Restaurant extends Account {
   //the string "Rest" is concated with the restaurant ID to
   //form the key.
   final static String RestaurantIDText = "Rest";
   String website;
   //address in Chinese.
   String addr_cn;
   //address in English.
   String addr_en;
   //the restaurant would like to update its fax # with this. After it is
   //confirmed, it will be stored in Account. Before that, it is stored
   //here.
   String numb_newfax;
   boolean has_NewFax = false;
   //a list of holidays.
   Vector Holiday; // a holiday
   //id of the category this restaurant belongs to.
   String catId;
   //a list of business session. Each business session is an array
   //of two times. The first time is the start time. The second time
   //is the end time. The restaurant is open for business in each
   //session.
   Vector BsHour; //Business hour
   ...
   //y: year.
   //m: month.
   //d: date.
   void addHoliday(int y,int m,int d){
      if(y<1900) y+=1900;
      Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
      Holiday.add(aHoliday);
   }
   public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
      int fMin = fromHr*60 + fromMin;  //start time in minutes.
      int tMin = toHr*60 + toMin;   //end time in minutes.
      //make sure both times are valid and the start time is earlier
      //than the end time.
      if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
         GregorianCalendar bs[] = {
            new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            new GregorianCalendar(1900,1,1, toHr, toMin,0)
         };
         BsHour.add(bs);
         return true;
      } else {
         return false;
      }
   }
}
```

3. Turn the comments into code:

```
class Account {
   ...
   //check if the password is complex enough, i.e.,
   //contains letter and digit/symbol.
   boolean isComplexPassword(String password){
      //found a digit or symbol?
      boolean dg_sym_found=false;
      //found a letter?
      boolean letter_found=false;
      for(int i=0; i<password.length(); i++){
         char c=password.charAt(i);
         if(Character.isLowerCase(c)||Character.isUpperCase(c))
             letter_found=true;
         else dg_sym_found=true;
      }
      return (letter_found) && (dg_sym_found);
   }
}
```

4. Turn the comments into code:

```
class TokenStream {
   Vector v;  //a list of tokens parsed from br.
   int index; //index of the current token in v.
   BufferedReader br; //read the chars from here to parse the tokens.
   int currentChar;  //previous char read from br.

   //read the chars from the reader and parse the tokens.
   TokenStream(Reader read) {
      br = new BufferedReader(read);
      takeChar();
      v = parseFile();
      index=0;
   }
   //read the chars from br, parse the tokens and store them into a vector.
   Vector parseFile() {
      Vector v; //accumulate the tokens that have been parsed.
      ...
      return v;
   }
   ...
}
```

5. Turn the comments into code:

```
class FoodDB {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   //find all foods whose names contain both the keywords. returns
   //an iterator on these foods.
   public Iterator srchFood(String cName, String eName){
      //it contains all the foods to be returned.
      TreeMap foodTree = new TreeMap();
      Iterator foodList;
      Food food;
      foodList = getAllRecords();
      while (foodList.hasNext()){
         food = (Food) foodList.next();
         //do its names contain both keywords?
         if ((cName==null || //null or "" means no key is specified
            cName.equals("") ||
            food.getCName().indexOf(cName)!=-1)
            &&
            (eName==null || //null or "" means no key is specified
            eName.equals("") ||
            food.getEName().indexOf(eName)!=-1)){
            foodTree.put(food.getFoodKey(),food);
         }
      }
      return foodTree.values().iterator();
   }
}
```

6. Turn the comments into code:

```
//an order.
class Order {
   String orderId; //order ID.
   Restaurant r1;  //the order is placed for this restaurant.
   Customer c1;  //the order is created by this customer.
   //"H": deliver to home address of the customer.
   //"W": deliver to work address of the customer.
   //"O": deliver to the address specified here.
   String addressType;
   String otherAddress; //address if addressType is "O".
   HashMap orderItems;  //order items.

   //get the total amount of this order.
   public double getTotalAmt() {
      //total amount.
      BigDecimal amt= new BigDecimal("0.00");
      //for each order item do...
      Iterator iter=orderItems.values().iterator();
      while(iter.hasNext()){
         //add the amount of the next order item.
         OrderItem oi=(OrderItem)iter.next();
         amt = amt.add(new BigDecimal(String.valueOf(oi.getAmount())));
      }
      return amt.doubleValue();
   }
}
```

7. Come up with some code that contains comments and then turn the comments into code.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Hints

1. Why not just call the method "convertToPoints"? Also, introduce a variable with a good name to hold the value of 72.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Turn the comments into code:

```
class InchToPointConvertor {
   //convert the quantity in inches to points.
   static float parseInch(float inch) {
      return inch * 72; //one inch contains 72 points.
   }
}
```

   Turn the first comment into the method name and the second into a variable name:

```
class InchToPointConvertor {
   final static int POINTS_PER_INCH=72;
   static float convertToPoints(float inch) {
      return inch * POINTS_PER_INCH;
   }
}
```

2. Turn the comments into code:

```
class Restaurant extends Account {
   //the string "Rest" is concated with the restaurant ID to
   //form the key.
   final static String RestaurantIDText = "Rest";
   String website;
   //address in Chinese.
   String addr_cn;
   //address in English.
   String addr_en;
   //the restaurant would like to update its fax # with this. After it is
   //confirmed, it will be stored in Account. Before that, it is stored
   //here.
   String numb_newfax;
   boolean has_NewFax = false;
   //a list of holidays.
   Vector Holiday; // a holiday
   //id of the category this restaurant belongs to.
   String catId;
   //a list of business session. Each business session is an array
   //of two times. The first time is the start time. The second time
   //is the end time. The restaurant is open for business in each
   //session.
   Vector BsHour; //Business hour
   ...
   //y: year.
   //m: month.
   //d: date.
   void addHoliday(int y,int m,int d){
      if(y<1900) y+=1900;
      Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
      Holiday.add(aHoliday);
   }
   public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
      int fMin = fromHr*60 + fromMin;  //start time in minutes.
      int tMin = toHr*60 + toMin;   //end time in minutes.
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      //make sure both times are valid and the start time is earlier
      //than the end time.
      if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
         GregorianCalendar bs[] = {
            new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
            new GregorianCalendar(1900,1,1, toHr, toMin,0)
         };
         BsHour.add(bs);
         return true;
      } else {
         return false;
      }
   }
}
```

The most severe problem here is the long comment explaining the Vector named "BsHour". It is explaining how a business session is represented by an array of two times. We should turn this comment into a class definition for business session:

```
class BusinessSession {
   int minStart;
   int minEnd;
   BusinessSession(int fromHour, int fromMinute, int toHour, int toMinute) {
      minStart=getMinutesFromMidNight(fromHour, fromMinute);
      minEnd=getMinutesFromMidNight(toHour, toMinute);
   }
   int getMinutesFromMidNight(int hours, int minutes) {
      return hours*60+minutes;
   }
   boolean isMinutesWithinOneDay(int minutes) {
      return minutes>0 && minutes<=1440;
   }
   boolean isValid() {
      return isMinutesWithinOneDay(minStart) &&
            isMinutesWithinOneDay(minEnd) &&
            minStart<minEnd;
   }
}
class Restaurant extends Account {
   final static String keyPrefix="Rest";
   String website;
   String chineseAddr;
   String englishAddr;
   String faxNoUnderConfirmation;
   boolean hasFaxNoUnderConfirmation = false;
   Vector holidayList;
   String catIdForThisRestaurant;
   Vector businessSessionList;
   ...
   void addHoliday(int year, int month, int day){
      if(year<1900) year+=1900;
      Calendar aHoliday = (new GregorianCalendar(year,month,day,0,0,0));
      holidayList.add(aHoliday);
   }
   public boolean addBusinessSession(int fromHr, int fromMin, int toHr, int
toMin){
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      BusinessSession bs=new BusinessSession(fromHr, fromMin, toHr, toMin);
      if(bs.isValid()){
         businessSessionList.add(bs);
         return true;
      } else {
         return false;
      }
   }
}
```

3. Turn the comments into code:

```
class Account {
   ...
   //check if the password is complex enough, i.e.,
   //contains letter and digit/symbol.
   boolean isComplexPassword(String password){
      //found a digit or symbol?
      boolean dg_sym_found=false;
      //found a letter?
      boolean letter_found=false;
      for(int i=0; i<password.length(); i++){
         char c=password.charAt(i);
         if(Character.isLowerCase(c)||Character.isUpperCase(c))
             letter_found=true;
         else dg_sym_found=true;
      }
      return (letter_found) && (dg_sym_found);
   }
}
```

The first comment contains two sentences. The first sentence ("check if the password is complex enough") describes the purpose of the method, it should be the method name. As the method name (isComplexPassword) is exactly like that, so we can simply delete this sentence. The second sentence ("contains letter and digit/symbol") describes how the method works, so it should become the method body, not the method name.

```
class Account {
   ...
   boolean isComplexPassword(String password){
      return containsLetter(password) &&
            (containsDigit(password) || containsSymbol(password));
   }
   boolean containsLetter(String password) {
      ...
   }
   boolean containsDigit(String password) {
      ...
   }
   boolean containsSymbol(String password) {
      ...
   }
}
```

4. Turn the comments into code:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class TokenStream {
   Vector v;  //a list of tokens parsed from br.
   int index; //index of the current token in v.
   BufferedReader br; //read the chars from here to parse the tokens.
   int currentChar;  //previous char read from br.

   //read the chars from the reader and parse the tokens.
   TokenStream(Reader read) {
      br = new BufferedReader(read);
      takeChar();
      v = parseFile();
      index=0;
   }
   //read the chars from br, parse the tokens and store them into a vector.
   Vector parseFile() {
      Vector v; //accumulate the tokens that have been parsed.
      ...
      return v;
   }
   ...
}
```

The comments can be turned into names of variables, parameters and methods. The comment describing how the constructor works should be turned into the body of the constructor.

```
class TokenStream {
   Vector parsedTokenList;
   int currentTokenIdxInList;
   BufferedReader charInputSourceForParsing;
   int previousCharReadFromSource;

   TokenStream(Reader reader) {
      charInputSourceForParsing = new BufferedReader(reader);
      takeChar();
      parsedTokenList = parseTokensFromInputSource();
      currentTokenIdxInList = 0;
   }
   Vector parseTokensFromInputSource() {
      Vector tokensParsedSoFar;
      ...
      return tokensParsedSoFar;
   }
   ...
}
```

5. Turn the comments into code:

```
class FoodDB {
   //find all foods whose names contain both the keywords. returns
   //an iterator on these foods.
   public Iterator srchFood(String cName, String eName){
      //it contains all the foods to be returned.
      TreeMap foodTree = new TreeMap();
      Iterator foodList;
      Food food;
      foodList = getAllRecords();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      while (foodList.hasNext()){
         food = (Food) foodList.next();
         //do its names contain both keywords?
         if ((cName==null || //null or "" means no key is specified
            cName.equals("") ||
            food.getCName().indexOf(cName)!=-1)
            &&
            (eName==null || //null or "" means no key is specified
            eName.equals("") ||
            food.getEName().indexOf(eName)!=-1)){
            foodTree.put(food.getFoodKey(),food);
         }
      }
      return foodTree.values().iterator();
   }
}
```

The comments can be turned into names of variables, parameters and methods.

```
class FoodDB {
   public Iterator findFoodsWithKeywordsInNames(String cKeyword, String
eKeyword){
      TreeMap foodsFound = new TreeMap();
      for (Iterator foodIter=getAllRecords(); foodIter.hasNext(); ){
         Food food = (Food) foodIter.next();
         if (foodContainsKeyInNames(food, cKeyword, eKeyword)) {
            foodsFound.put(food.getFoodKey(),food);
         }
      }
      return foodsFound.values().iterator();
   }
   boolean foodContainsKeyInNames(
      Food food,
      String cKeyword,
      String eKeyword) {
      return nameContainsKeyword(food.getCName(), cKeyword) &&
            nameContainsKeyword(food.getEName(), eKeyword);
   }
   boolean nameContainsKeyword(String name, String keyword) {
      return keywordIsNotSpecified(keyword) || name.indexOf(keyword)!=-1;
   }
   boolean keywordIsNotSpecified(String keyword) {
      return keyword==null || keyword.equals("");
   }
}
```

6. Turn the comments into code:

```
//an order.
class Order {
   String orderId; //order ID.
   Restaurant r1;  //the order is placed for this restaurant.
   Customer c1;  //the order is created by this customer.
   //"H": deliver to home address of the customer.
   //"W": deliver to work address of the customer.
   //"O": deliver to the address specified here.
   String addressType;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   String otherAddress; //address if addressType is "O".
   HashMap orderItems;  //order items.

   //get the total amount of this order.
   public double getTotalAmt() {
      //total amount.
      BigDecimal amt= new BigDecimal("0.00");
      //for each order item do...
      Iterator iter=orderItems.values().iterator();
      while(iter.hasNext()){
         //add the amount of the next order item.
         OrderItem oi=(OrderItem)iter.next();
         amt = amt.add(new BigDecimal(String.valueOf(oi.getAmount())));
      }
      return amt.doubleValue();
   }
}
```

The comments can be turned into names of variables, parameters and methods.

```
class Order {
   String orderId;
   Restaurant restToReceiveOrder;
   Customer customerPlacingOrder;
   static final String DELIVER_ADDRESS_TYPE_HOME="H";
   static final String DELIVER_ADDRESS_TYPE_WORK="W";
   static final String DELIVER_ADDRESS_TYPE_OTHER="O";
   String addressType;
   String otherAddress; //address if addressType is DELIVER_ADDRESS_TYPE_OTHER.
   HashMap orderItems;

   Iterator getItemsIterator() {
      return orderItems.values().iterator();
   }
   public double getTotalAmount() {
      BigDecimal totalAmt= new BigDecimal("0.00");
      for (Iterator iter=getItemsIterator(); iter.hasNext(); ){
         OrderItem nextOrderItem=(OrderItem)iter.next();
         totalAmt =
            totalAmt.add(
               new BigDecimal(String.valueOf(nextOrderItem.getAmount())));
      }
      return totalAmt.doubleValue();
   }
}
```

Note that one comment is still left. It is very difficult to remove this comment, unless we remove the type code. To see how to remove type codes, see Chapter 3: Removing Code Smells.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 3

Removing Code Smells

Example

This is a CAD application. At the moment it can draw three types of shapes: lines, rectangles and circles. Please read the code below carefully:

```
class Shape {
   final static int TYPELINE = 0;
   final static int TYPERECTANGLE = 1;
   final static int TYPECIRCLE = 2;
   int shapeType;
   //starting point of the line.
   //lower left corner of the rectangle.
   //center of the circle.
   Point p1;
   //ending point of the line.
   //upper right corner of the rectangle.
   //not used for the circle.
   Point p2;
   int radius;
}
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         switch (shapes[i].getType()) {
            case Shape.TYPELINE:
               graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
               break;
            case Shape.TYPERECTANGLE:
               //draw the four edges.
               graphics.drawLine(...);
               graphics.drawLine(...);
               graphics.drawLine(...);
               graphics.drawLine(...);
               break;
            case Shape.TYPECIRCLE:
               graphics.drawCircle(shapes[i].getP1(), shapes[i].getRadius());
               break;
         }
      }
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Problem in the code above: The code will keep changing

There is a problem in the code above: if we need to support one more shape (e.g., triangles), the Shape class needs to change, and the drawShapes method in CADApp class also needs to change, e.g.:

```
class Shape {
   final static int TYPELINE = 0;
   final static int TYPERECTANGLE = 1;
   final static int TYPECIRCLE = 2;
   final static int TYPETRIANGLE = 3;
   int shapeType;
   Point p1;
   Point p2;
   //third point of the triangle.
   Point p3;
   int radius;
}
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         switch (shapes[i].getType()) {
            case Shape.TYPELINE:
               graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
               break;
            case Shape.TYPERECTANGLE:
               //draw the four edges.
               graphics.drawLine(...);
               graphics.drawLine(...);
               graphics.drawLine(...);
               graphics.drawLine(...);
               break;
            case Shape.TYPECIRCLE:
               graphics.drawCircle(shapes[i].getP1(), shapes[i].getRadius());
               break;
            case Shape.TYPETRIANGLE:
               graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
               graphics.drawLine(shapes[i].getP2(), shapes[i].getP3());
               graphics.drawLine(shapes[i].getP3(), shapes[i].getP1());
               break;
         }
      }
   }
}
```

If in the future one more shape is added, they will have to change again. This is no good thing. Ideally, we hope that a class, a method and etc. do not need to change after they are written. They should be readily available for reuse without change. But the current case is the opposite. When we keep adding new shapes, we will keep changing the Shape class and the drawShapes method in the CADApp class.

How to stabilize this code (no need to change again)? Before answering this question, let's consider another question first: Given a piece of code, how to check if it is stable?

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

How to check if some code is stable

To check if some given code is stable, we may do it like this: Assume if some certain situations arise or some certain new requirements come, then check if the code needs to change. However, it is very hard to do because there are too many possible situations that could arise.

A simpler method is, when we find that this is already the third time that we change this code, we consider it unstable. This is a "lazy" and "passive" method. We start to handle to problem only when we feel the pain. However, it is a very effective method.

In addition, there is another simple but more "active" method: If the code is unstable or contains some other potential problems, the code will often contain some obvious traces. We call these traces "code smells". Code smells are not always a bad thing, but most often they are indeed a bad thing. Therefore, whenever we find code smells in the code, we must become alerted and check the code carefully.

Now, let's see the code smells in the example code above.

Code smells in the example code

The first code smell: The code uses type code:

```
class Shape {
   final int TYPELINE = 0;
   final int TYPERECTANGLE = 1;
   final int TYPECIRCLE = 2;
   int shapeType;
   ...
}
```

This is a severe warning that our code may have problems.

The second code smell: The Shape class contains variables that are not always used. For example, the variable radius is used only when the Shape is a circle:

```
class Shape {
   ...
   Point p1;
   Point p2;
   int radius; //not always used!
}
```

The third code smell: We cannot give the variable p1 (or p2) a better name, because it has different meanings in different situations:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class Shape {
   ...
   Point p1; //should we call it startPoint, lowerLeftCorner or center?
   Point p2;
}
```

The fourth code smell: The switch statement in drawShapes. When we use switch (or a long list of if-then-else-if), get alerted. The switch statement commonly appears with type code at the same time.

Now, let's see how to remove the code smells in the example code above.

Removing code smells: How to remove a type code

Usually, in order to remove a type code, we can create a sub-class for each type code value, e.g.:

```
class Shape {
}
class Line extends Shape {
   Point startPoint;
   Point endPoint;
}
class Rectangle extends Shape {
   Point lowerLeftCorner;
   Point upperRightCorner;
}
class Circle extends Shape {
   Point center;
   int radius;
}
```

Because there is no type code, drawShapes has to use instanceof to check the type of the Shape object. Therefore, it can't use switch any more. It must change to use if-then-else such as:

```
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         if (shapes[i] instanceof Line) {
            Line line = (Line)shapes[i];
            graphics.drawLine(line.getStartPoint(),line.getEndPoint());
         } else if (shapes[i] instanceof Rectangle) {
            Rectangle rect = (Rectangle)shapes[i];
            graphics.drawLine(...);
            graphics.drawLine(...);
            graphics.drawLine(...);
            graphics.drawLine(...);
         } else if (shapes[i] instanceof Circle) {
            Circle circle = (Circle)shapes[i];
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            graphics.drawCircle(circle.getCenter(), circle.getRadius());
         }
      }
   }
}
```

Now, without the type code, the variables in each class (Shape, Line, Rectangle, Circle) are always useful. They also have much better names (p1 can now be called startPoint). The only code smell left is the long if-then-else-if in drawShapes. Our next step is to remove this long if-then-else-if.

Removing code smells: How to remove a long if-then-else-if

Usually, in order to remove a long if-then-else-if or a switch, we need to try to make the code in each conditional branch identical. For drawShapes, we will first write the code in each branch in a more abstract way:

```
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         if (shapes[i] instanceof Line) {
            draw the line;
         } else if (shapes[i] instanceof Rectangle) {
            draw the rectangle;
         } else if (shapes[i] instanceof Circle) {
            draw the circle;
         }
      }
   }
}
```

However, the code in these three branches is still different. So, make it even more abstract:

```
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         if (shapes[i] instanceof Line) {
            draw the shape;
         } else if (shapes[i] instanceof Rectangle) {
            draw the shape;
         } else if (shapes[i] instanceof Circle) {
            draw the shape;
         }
      }
   }
}
```

Now, the code in these three branches is identical. Therefore, we no longer need the different branches:

```
class CADApp {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         draw the shape;
      }
   }
}
```

Finally, turn "draw the shape" into a method of the Shape class:

```
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         shapes[i].draw(graphics);
      }
   }
}
```

Of course, we need to provide this draw method:

```
abstract class Shape {
   abstract void draw(Graphics graphics);
}
class Line extends Shape {
   Point startPoint;
   Point endPoint;
   void draw(Graphics graphics) {
      graphics.drawLine(getStartPoint(), getEndPoint());
   }
}
class Rectangle extends Shape {
   Point lowerLeftCorner;
   Point upperRightCorner;
   void draw(Graphics graphics) {
      graphics.drawLine(...);
      graphics.drawLine(...);
      graphics.drawLine(...);
      graphics.drawLine(...);
   }
}
class Circle extends Shape {
   Point center;
   int radius;
   void draw(Graphics graphics) {
      graphics.drawCircle(getCenter(), getRadius());
   }
}
```

Turning an abstract class into an interface

Now, Shape is an abstract class without any executable code. It seems more appropriate to turn it into an interface:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
interface Shape {
   void draw(Graphics graphics);
}
class Line implements Shape {
   ...
}
class Rectangle implements Shape {
   ...
}
class Circle implements Shape {
   ...
}
```

The improved code

The improved code is shown below:

```
interface Shape {
   void draw(Graphics graphics);
}
class Line implements Shape {
   Point startPoint;
   Point endPoint;
   void draw(Graphics graphics) {
      graphics.drawLine(getStartPoint(), getEndPoint());
   }
}
class Rectangle implements Shape {
   Point lowerLeftCorner;
   Point upperRightCorner;
   void draw(Graphics graphics) {
      graphics.drawLine(...);
      graphics.drawLine(...);
      graphics.drawLine(...);
      graphics.drawLine(...);
   }
}
class Circle implements Shape {
   Point center;
   int radius;
   void draw(Graphics graphics) {
      graphics.drawCircle(getCenter(), getRadius());
   }
}
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         shapes[i].draw(graphics);
      }
   }
}
```

If we need to support one more shape (e.g., triangle), none of classes needs to change. All it takes is to create a new Triangle class.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Another example

Let's see another example. In this system there are three types of users: regular users, administrators and guests. Regular users must change their password once every 90 days (or sooner). Administrators must change their password once every 30 days. Guests don't need to change passwords. Only regular users and administrators can print reports. Please read the current code:

```
class UserAccount {
   final static int USERTYPE_NORMAL = 0;
   final static int USERTYPE_ADMIN = 1;
   final static int USERTYPE_GUEST = 2;
   int userType;
   String id;
   String name;
   String password;
   Date dateOfLastPasswdChange;
   public boolean checkPassword(String password) {
      ...
   }
}
class InventoryApp {
   void login(UserAccount userLoggingIn, String password) {
      if (userLoggingIn.checkPassword(password)) {
         GregorianCalendar today = new GregorianCalendar();
         GregorianCalendar expiryDate = getAccountExpiryDate(userLoggingIn);
         if (today.after(expiryDate)) {
            //prompt the user to change password
            ...
         }
      }
   }
   GregorianCalendar getAccountExpiryDate(UserAccount account) {
      int passwordMaxAgeInDays = getPasswordMaxAgeInDays(account);
      GregorianCalendar expiryDate = new GregorianCalendar();
      expiryDate.setTime(account.dateOfLastPasswdChange);
      expiryDate.add(Calendar.DAY_OF_MONTH, passwordMaxAgeInDays);
      return expiryDate;
   }
   int getPasswordMaxAgeInDays(UserAccount account) {
      switch (account.getType()) {
         case UserAccount.USERTYPE_NORMAL:
            return 90;
         case UserAccount.USERTYPE_ADMIN:
            return 30;
         case UserAccount.USERTYPE_GUEST:
            return Integer.MAX_VALUE;
      }
   }
   void printReport(UserAccount currentUser) {
      boolean canPrint;
      switch (currentUser.getType()) {
         case UserAccount.USERTYPE_NORMAL:
            canPrint = true;
            break;
         case UserAccount.USERTYPE_ADMIN:
            canPrint = true;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            break;
         case UserAccount.USERTYPE_GUEST:
            canPrint = false;
      }
      if (!canPrint) {
         throw new SecurityException("You have no right");
      }
      //print the report.
   }
}
```

Use an object to represent a type code value

According to the method described before, to remove a type code, we only need to create a sub-class for each type code value, e.g.:

```
abstract class UserAccount {
   String id;
   String name;
   String password;
   Date dateOfLastPasswdChange;
   abstract int getPasswordMaxAgeInDays();
   abstract boolean canPrintReport();
}
class NormalUserAccount extends UserAccount {
   int getPasswordMaxAgeInDays() {
      return 90;
   }
   boolean canPrintReport() {
      return true;
   }
}
class AdminUserAccount extends UserAccount {
   int getPasswordMaxAgeInDays() {
      return 30;
   }
   boolean canPrintReport() {
      return true;
   }
}
class GuestUserAccount extends UserAccount {
   int getPasswordMaxAgeInDays() {
      return Integer.MAX_VALUE;
   }
   boolean canPrintReport() {
      return false;
   }
}
```

However, the behavior (code) of these three sub-classes is too similar: Their getPasswordMaxAgeInDays methods only differ in a value (30, 90 or Integer.MAX_VALUE). Their canPrintReport methods also only differ in a value (true or false). Therefore, these three

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

user types can be represented using three objects instead of three sub-classes:

```
class UserAccount {
   UserType userType;
   String id;
   String name;
   String password;
   Date dateOfLastPasswdChange;
   UserType getType() {
      return userType;
   }
}
class UserType {
   int passwordMaxAgeInDays;
   boolean allowedToPrintReport;
   UserType(int passwordMaxAgeInDays, boolean allowedToPrintReport) {
      this.passwordMaxAgeInDays = passwordMaxAgeInDays;
      this.allowedToPrintReport = allowedToPrintReport;
   }
   int getPasswordMaxAgeInDays() {
      return passwordMaxAgeInDays;
   }
   boolean canPrintReport() {
      return allowedToPrintReport;
   }
   static UserType normalUserType = new UserType(90, true);
   static UserType adminUserType = new UserType(30, true);
   static UserType guestUserType = new UserType(Integer.MAX_VALUE, false);
}
class InventoryApp {
   void login(UserAccount userLoggingIn, String password) {
      if (userLoggingIn.checkPassword(password)) {
         GregorianCalendar today = new GregorianCalendar();
         GregorianCalendar expiryDate = getAccountExpiryDate(userLoggingIn);
         if (today.after(expiryDate)) {
            //prompt the user to change password
            ...
         }
      }
   }
   GregorianCalendar getAccountExpiryDate(UserAccount account) {
      int passwordMaxAgeInDays = getPasswordMaxAgeInDays(account);
      GregorianCalendar expiryDate = new GregorianCalendar();
      expiryDate.setTime(account.dateOfLastPasswdChange);
      expiryDate.add(Calendar.DAY_OF_MONTH, passwordMaxAgeInDays);
      return expiryDate;
   }
   int getPasswordMaxAgeInDays(UserAccount account) {
      return account.getType().getPasswordMaxAgeInDays();
   }
   void printReport(UserAccount currentUser) {
      boolean canPrint;
      canPrint = currentUser.getType().canPrintReport();
      if (!canPrint) {
         throw new SecurityException("You have no right");
      }
      //print the report.
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
```

Note that using an object to represent a type code value can also remove the switch statement.

Summary on type code removal

To remove type codes and switch statements, there are two methods:

1. Use different sub-classes of a base class to represent the different type code values.
2. Use different objects of a class to represent the different type code values.

When different type code values need very different behaviors, use method 1. When their behaviors are mostly the same and only differ in values, use method 2.

Common code smells

Type codes and switch statements are common code smells. In addition, there are quite some other code smells that are also very common. Below is a summary list:

- Duplicate code.
- Too many comments.
- Type code.
- Switch or a long if-then-else-if.
- Cannot give a good name to a variable, method or class.
- Use names like XXXUtil, XXXManager, XXXController and etc.
- Use the words "And", "Or" and etc. in the name of a variable, method or class.
- Some instance variables are not always used.
- A method contains too much code.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- A class contains too much code.
- A method takes too many parameters.
- Two classes are using each other.

References

- http://c2.com/cgi/wiki?CodeSmell.
- The Open Closed Principle states: If we need to add more functionality, we should only need to add new code, but not change the existing code. Removing type codes and switch statements is a common way to achieve Open Closed Principle. For more information about Open Closed Principle, please see http://www.objectmentor.com/publications/ocp.pdf.
- Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2002.
- Bertrand Meyer, Object-Oriented Software Construction, Pearson Higher Education, 1988.
- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999. This book lists many code smells and the refactoring methods.
- Martin Fowler's web site about refactoring: http://www.refactoring.com/catalog/index.html.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```
class FoodSalesReport {
   int q0; //how many items of rice sold?
   int q1; //how many items of noodle sold?
   int q2; //how many items of drink sold?
   int q3; //how many items of dessert sold?
   void LoadData(Connection conn) {
      PreparedStatement st = conn.prepareStatement("select "+
         "sum(case when foodType=0 then qty else 0 end) as totalQty0,"+
         "sum(case when foodType=1 then qty else 0 end) as totalQty1,"+
         "sum(case when foodType=2 then qty else 0 end) as totalQty2,"+
         "sum(case when foodType=3 then qty else 0 end) as totalQty3 "+
         "from foodSalesTable group by foodType");
      try {
         ResultSet rs = st.executeQuery();
         try {
            rs.next();
            q0 = rs.getInt("totalQty0");
            q1 = rs.getInt("totalQty1");
            q2 = rs.getInt("totalQty2");
            q3 = rs.getInt("totalQty3");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
}
```

2. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```
class SurveyData {
   String path; //save the data to this file.
   boolean hidden; //should the file be hidden?
   //set the path to save the data according to the type of data (t).
   void setSavePath(int t) {
      if (t==0) { //raw data.
         path = "c:/application/data/raw.dat";
         hidden = true;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      } else if (t==1) {  //cleaned up data.
         path = "c:/application/data/cleanedUp.dat";
         hidden = true;
      } else if (t==2) {  //processed data.
         path = "c:/application/data/processed.dat";
         hidden = true;
      } else if (t==3) {  //data ready for publication.
         path = "c:/application/data/publication.dat";
         hidden = false;
      }
   }
}
```

3. Point out and remove the code smells in the code (this example was inspired by the one given by Eugen):

```
class CustomersInDB {
   Connection conn;
   Customer getCustomer(String IDNumber) {
      PreparedStatement st = conn.prepareStatement(
         "select * from customer where ID=?");
      try {
         st.setString(1,
               IDNumber.replace('-', ' ').
                  replace('(', ' ').
                  replace(')', ' ').
                  replace('/', ' '));
         ResultSet rs = st.executeQuery();
         ...
      } finally {
         st.close();
      }
   }
   void addCustomer(Customer customer) {
      PreparedStatement st = conn.prepareStatement(
         "insert into customer values(?,?,?,?)");
      try {
         st.setString(1,
               customer.getIDNumber().replace('-', ' ').
                       replace('(', ' ').
                       replace(')', ' ').
                       replace('/', ' '));
         st.setString(2, customer.getName());
         ...
         st.executeUpdate();
         ...
      } finally {
         st.close();
      }
   }
}
```

4. The code below contains duplicate code: the loop in printOverdueRentals and in countOverdueRentals. If you need to remove this duplication at any cost, how do you do it?

```
class BookRentals {
   Vector rentals;
   int countRentals() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      return rentals.size();
   }
   BookRental getRentalAt(int i) {
      return (BookRental)rentals.elementAt(i);
   }
   void printOverdueRentals() {
      int i;
      for (i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            System.out.println(rental.toString());
      }
   }
   int countOverdueRentals() {
      int i, count;
      count=0;
      for (i=0; i<countRentals(); i++)
         if (getRentalAt(i).isOverdue())
            count++;
      return count;
   }
}
```

5. Point out and remove the code smells in the code:

```
class Currency {
   final public int USD=0;
   final public int RMB=1;
   final public int ESCUDO=2;  //Portuguese currency
   private int currencyCode;
   public Currency(int currencyCode) {
      this.currencyCode=currencyCode;
   }
   public String format(int amount) {
      switch (currencyCode) {
         case USD:
            //return something like $1,200
         case RMB:
            //return something like RMB1,200
         case ESCUDO:
            //return something like $1.200
      }
   }
}
```

6. Point out and remove the code smells in the code:

```
class Payment {
   final static String FOC = "FOC";  //free of charge.
   final static String TT = "TT";  //paid by telegraphic transfer.
   final static String CHEQUE = "Cheque";  //paid by cheque.
   final static String CREDIT_CARD = "CreditCard"; //paid by credit card.
   final static String CASH = "Cash";  //paid by cash.
   //type of payment. Must be one of the above constant.
   String paymentType;
   Date paymentDate; //if FOC, the date the fee is waived.
   int actualPayment; //if FOC, it is 0.
   int discount;  //if FOC, the amount that is waived.
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   String bankName; //if it is by TT, cheque or credit card.
   String chequeNumber; //if it is by cheque.
   //if it is by credit card.
   String creditCardType;
   String creditCardHolderName;
   String creditCardNumber;
   Date creditCardExpiryDate;
   int getNominalPayment() {
      return actualPayment+discount;
   }
   String getBankName() {
      if (paymentType.equals(TT) ||
         paymentType.equals(CHEQUE) ||
         paymentType.equals(CREDIT_CARD)) {
         return bankName;
      }
      else {
         throw new Exception("bank name is undefined for this payment type");
      }
   }
}
```

7. In the above application, there is a dialog to allow the user edit a payment object. The user can choose the payment type in a combo box. Then the related components for that payment type will be displayed. This function is provided by the CardLayout in Java. The code is shown below. Update the code accordingly.

```
class EditPaymentDialog extends JDialog {
   Payment newPayment;  //new payment to be returned.
   JPanel sharedPaymentDetails;
   JPanel uniquePaymentDetails;
   JTextField paymentDate;
   JComboBox paymentType;
   JTextField discountForFOC;
   JTextField bankNameForTT;
   JTextField actualAmountForTT;
   JTextField discountForTT;
   JTextField bankNameForCheque;
   JTextField chequeNumberForCheque;
   JTextField actualAmountForCheque;
   JTextField discountForCheque;
   ...

   EditPaymentDialog() {
      //setup the components.
      Container contentPane = getContentPane();
      String comboBoxItems[] = { //available payment types.
         Payment.FOC,
         Payment.TT,
         Payment.CHEQUE,
         Payment.CREDIT_CARD,
         Payment.CASH
      };
      //setup the components for the details shared by all types of payment.
      sharedPaymentDetails = new JPanel();
      paymentDate = new JTextField();
      paymentType = new JComboBox(comboBoxItems);
      sharedPaymentDetails.add(paymentDate);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      sharedPaymentDetails.add(paymentType);
      contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
      //setup the unique components for each type of payment.
      uniquePaymentDetails = new JPanel();
      uniquePaymentDetails.setLayout(new CardLayout());
      //setup a panel for FOC payment.
      JPanel panelForFOC = new JPanel();
      discountForFOC = new JTextField();
      panelForFOC.add(discountForFOC);
      uniquePaymentDetails.add(panelForFOC, Payment.FOC);
      //setup a panel for TT payment.
      JPanel panelForTT = new JPanel();
      bankNameForTT = new JTextField();
      actualAmountForTT = new JTextField();
      discountForTT = new JTextField();
      panelForTT.add(bankNameForTT);
      panelForTT.add(actualAmountForTT);
      panelForTT.add(discountForTT);
      uniquePaymentDetails.add(panelForTT, Payment.TT);
      //setup a panel for cheque payment.
      JPanel panelForCheque = new JPanel();
      bankNameForCheque = new JTextField();
      chequeNumberForCheque = new JTextField();
      actualAmountForCheque = new JTextField();
      discountForCheque = new JTextField();
      panelForCheque.add(bankNameForCheque);
      panelForCheque.add(chequeNumberForCheque);
      panelForCheque.add(actualAmountForCheque);
      panelForCheque.add(discountForCheque);
      uniquePaymentDetails.add(panelForCheque, Payment.CHEQUE);
      //setup a panel for credit card payment.
      ...
      //setup a panel for cash payment.
      ...
      contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
   }
Payment editPayment(Payment payment) {
      displayPayment(payment);
      setVisible(true);
      return newPayment;
   }
   void displayPayment(Payment payment) {
      paymentDate.setText(payment.getDateAsString());
      paymentType.setSelectedItem(payment.getType());
      if (payment.getType().equals(Payment.FOC)) {
         discountForFOC.setText(Integer.toString(payment.getDiscount()));
      }
      else if (payment.getType().equals(Payment.TT)) {
         bankNameForTT.setText(payment.getBankName());
         actualAmountForTT.setText(
            Integer.toString(payment.getActualAmount()));
         discountForTT.setText(Integer.toString(payment.getDiscount()));
      }
      else if (payment.getType().equals(Payment.CHEQUE)) {
         bankNameForCheque.setText(payment.getBankName());
         chequeNumberForCheque.setText(payment.getChequeNumber());
         actualAmountForCheque.setText(
            Integer.toString(payment.getActualAmount()));
         discountForCheque.setText(Integer.toString(payment.getDiscount()));
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      else if (payment.getType().equals(Payment.CREDIT_CARD)) {
         //...
      }
      else if (payment.getType().equals(Payment.CASH)) {
         //...
      }
   }
   //when the user clicks OK.
   void onOK() {
      newPayment = makePayment();
      dispose();
   }
   //make a payment from the components.
   Payment makePayment() {
      String paymentTypeString = (String) paymentType.getSelectedItem();
      Payment payment = new Payment(paymentTypeString);
      payment.setDateAsText(paymentDate.getText());
      if (paymentTypeString.equals(Payment.FOC)) {
         payment.setDiscount(Integer.parseInt(discountForFOC.getText()));
      }
      else if (paymentTypeString.equals(Payment.TT)) {
         payment.setBankName(bankNameForTT.getText());
         payment.setActualAmount(
            Integer.parseInt(actualAmountForTT.getText()));
         payment.setDiscount(
            Integer.parseInt(discountForTT.getText()));
      }
      else if (paymentTypeString.equals(Payment.CHEQUE)) {
         payment.setBankName(bankNameForCheque.getText());
         payment.setChequeNumber(chequeNumberForCheque.getText());
         payment.setActualAmount(
            Integer.parseInt(actualAmountForCheque.getText()));
         payment.setDiscount(
            Integer.parseInt(discountForCheque.getText()));
      }
      else if (paymentTypeString.equals(Payment.CREDIT_CARD)) {
         //...
      }
      else if (paymentTypeString.equals(Payment.CASH)) {
         //...
      }
      return payment;
   }
}
```

8. This is an embedded application controlling a cooker. In every second, it will check if the cooker is over-heated (e.g., short-circuited). If yes it will cut itself off the power and make an alarm using its built-in speaker. It will also check if the moisture inside is lower than a certain threshold (e.g., the rice is cooked). If yes, it will turn its built-in heater to 50 degree Celsius just to keep the rice warm. In the future, you expect that some more things will need to be done in every second.

   Point out and remove the problem in the code.

```
class Scheduler extends Thread {
   Alarm alarm;
   HeatSensor heatSensor;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   PowerSupply powerSupply;
   MoistureSensor moistureSensor;
   Heater heater;
   public void run() {
      for (;;) {
         Thread.sleep(1000);
         //check if it is overheated.
         if (heatSensor.isOverHeated()) {
            powerSupply.turnOff();
            alarm.turnOn();
         }
         //check if the rice is cooked.
         if (moistureSensor.getMoisture()<60) {
            heater.setTemperature(50);
         }
      }
   }
}
```

9. This application is concerned with the training courses. The schedule of a course can be expressed in three ways (as of now): weekly, range or list. A weekly schedule is like "every Tuesday for 5 weeks starting from Oct. 22". A range schedule is like "Every day from Oct. 22 to Nov. 3". A list schedule is like "Oct. 22, Oct. 25, Nov. 3, Nov. 10". In this exercise we will ignore the time and just assume that it is always 7:00pm-10:00pm. It is expected that new ways to express the schedule may be added in the future.

   Point out and remove the code smells in the code:

```
class Course {
   static final int WEEKLY=0;
   static final int RANGE=1;
   static final int LIST=2;
   String courseTitle;
   int scheduleType;  // WEEKLY, RANGE, or LIST
   int noWeeks;       // For WEEKLY.
   Date fromDate;     // for WEEKLY and RANGE.
   Date toDate;       // for RANGE.
   Date dateList[];   // for LIST.

   int getDurationInDays() {
      switch (scheduleType) {
         case WEEKLY:
            return noWeeks;
         case RANGE:
            int msInOneDay = 24*60*60*1000;
            return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
         case LIST:
            return dateList.length;
         default:
            return 0;  // unknown schedule type!
      }
   }
   void printSchedule() {
      switch (scheduleType) {
         case WEEKLY:
            //...
         case RANGE:
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            //...
         case LIST:
            //...
      }
   }
}
```

10. This application is concerned with training courses. A course has a title, a fee and a list of sessions. However, sometimes a course can consist of several modules, each of which is a course. For example, there may be a compound course "Fast track to becoming a web developer" which consists of three modules: a course named "HTML", a course named "FrontPage" and a course named "Flash". It is possible for a module to consist of some other modules. If a course consists of modules, its fee and schedule are totally determined by that of its modules and thus it will not maintain its list of sessions.

    Point out and remove the code smells in the code:

```
class Session {
   Date date;
   int startHour;
   int endHour;
   int getDuration() {
      return endHour-startHour;
   }
}
class Course {
   String courseTitle;
   Session sessions[];
   double fee;
   Course modules[];

   Course(String courseTitle, double fee, Session sessions[]) {
      //...
   }
   Course(String courseTitle, Course modules[]) {
      //...
   }
   String getTitle() {
      return courseTitle;
   }
   double getDuration() {
      int duration=0;
      if (modules==null)
         for (int i=0; i<sessions.length; i++)
            duration += sessions[i].getDuration();
      else
         for (int i=0; i<modules.length; i++)
            duration += modules[i].getDuration();
      return duration;
   }
   double getFee() {
      if (modules==null)
         return fee;
      else {
         double totalFee = 0;
         for (int i=0; i<modules.length; i++)
            totalFee += modules[i].getFee();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
         return totalFee;
      }
   }
   void setFee(int fee) throws Exception {
      if (modules==null)
         this.fee = fee;
      else
         throw new Exception("Please set the fee of each module one by one");
   }
}
```

11. Point out and remove the code smells in the code:

```
class BookRental {
   String bookTitle;
   String author;
   Date rentDate;
   Date dueDate;
   double rentalFee;
   boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   double getTotalFee() {
      return isOverdue() ? 1.2*rentalFee : rentalFee;
   }
}
class MovieRental {
   String movieTitle;
   int classification;
   Date rentDate;
   Date dueDate;
   double rentalFee;
   boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   double getTotalFee() {
      return isOverdue() ? Math.max(1.3*rentalFee, rentalFee+20) : rentalFee;
   }
}
```

12. Point out and remove the code smells in the code:

```
class Customer {
   String homeAddress;
   String workAddress;
}
class Order {
   String orderId;
   Restaurant restaurantReceivingOrder;
   Customer customerPlacingOrder;
   //"H": deliver to home address of the customer.
   //"W": deliver to work address of the customer.
   //"O": deliver to the address specified here.
   String addressType;
   String otherAddress; //address if addressType is "O".
   HashMap orderItems;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public String getDeliveryAddress() {
      if (addressType.equals("H")) {
         return customerPlacingOrder.getHomeAddress();
      } else if (addressType.equals("W")) {
         return customerPlacingOrder.getWorkAddress();
      } else if (addressType.equals("O")) {
         return otherAddress;
      } else {
         return null;
      }
   }
}
```

13. Come up with some code that uses type code and remove it.

Hints

1. Rename q0-q3, LoadData and etc. The SQL statement contains duplicate code. The field names are duplicated.

2. The data folder is duplicated. The setting of hidden is duplicated. Consider removing the if-then-else-if. Consider renaming some names.

3. The code replacing the few special characters with a space is duplicated. Inside that code, the call to replace is duplicated.

4. You must make the code in the body in each loop look identical (in a certain abstraction level):

```
class BookRentals {
   ...
   void printOverdueRentals() {
      int i;
      for (i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            process the rental;
      }
   }
   int countOverdueRentals() {
      int i, count;
      count=0;
      for (i=0; i<countRentals(); i++)
         if (getRentalAt(i).isOverdue())
            process the rental;
      return count;
   }
}
Extract the common code into a separate method. Let the two existing methods call
this new method.
class BookRentals {
   ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void yyy() {
      for (i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            process the rental;
      }
   }
   void printOverdueRentals() {
      yyy();
   }
   int countOverdueRentals() {
      int count;
      yyy();
      return count;
   }
}
```

Because there are two implementations for the "process the rental" method, you should create an interface to allow for different implementations:

```
interface XXX {
   void process(Rental rental);
}
class BookRentals {
   ...
   void yyy(XXX obj) {
      for (i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            obj.process(rental);
      }
   }
   ...
}
```

Then, printOverdueRentals and countOverdueRentals need to provide their respective implementations:

```
class XXX1 implements XXX {
   ...
}
class XXX2 implements XXX {
   ...
}
class BookRentals {
   ...
   void yyy(XXX obj) {
      ...
   }
   void printOverdueRentals() {
      XXX1 obj=new XXX1();
      yyy(obj);
   }
   int countOverdueRentals() {
      int count;
      XXX2 obj=new XXX2();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        yyy(obj);
        return count;
    }
}
```

Think of a good names for XXX, XXX1, XXX2 and yyy by considering what they do. If possible, make XXX1 and XXX2 anonymous classes (possible in Java, not in Delphi or C++).

5. Create sub-classes like USDCurrency, RMBCurrency and ESCUDOCurrency. Each sub-class should have a format method.

6. Create sub-classes like FOCPayment, TTPayment and etc. Each sub-class should have a getNominalPayment method. Only some of them should have a getBankName method.

7. Some comments can be removed by making the code as clear as the comments.

   You should create a UI class for each payment class such as FOCPaymentUI, TTPaymentUI and etc. They should inherit/implement something like a PaymentUI. The PaymentUI have methods like tryToDisplayPayment(Payment payment) and makePayment(). The tryToDisplayPayment method tries to display the Payment object using the various components (text fields and etc.). The makePayment method uses the contents of those components to create a new Payment object. For example, TTPaymentUI checks if the Payment object is an TTPayment object. If yes, it will display the TTPayment's information in the components and return true. Otherwise, its tryToDisplayPayment method will return false. TTPaymentUI should create these components (including the panel holding the components) by itself. However, some components such as that for the payment date are shared by all UI objects and therefore should not be created by TTPaymentUI. Instead, create these shared components in the EditPaymentDialog and pass them into TTPaymentUI's constructor.

   Each UI class should implement the toString method so that we can put the UI objects into the paymentType combo box directly (combo box in Java can contain Objects, not just Strings. Java will call the toString method of each object for visual display).

   Using these classes, the long if-then-else-if in the displayPayment method and the makePayment method in the EditPaymentDialog class can be eliminated.

8. The run method will keep growing and growing. To stop it, you need to make the code checking for overheat look identical to the code checking for cooked rice (in a certain abstraction level). Any new code to be added to the run method in the future must also look identical. Consider the JButton and ActionListener in Java.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```
class FoodSalesReport {
   int q0; //how many items of rice sold?
   int q1; //how many items of noodle sold?
   int q2; //how many items of drink sold?
   int q3; //how many items of dessert sold?
   void LoadData(Connection conn) {
      PreparedStatement st = conn.prepareStatement("select "+
         "sum(case when foodType=0 then qty else 0 end) as totalQty0,"+
         "sum(case when foodType=1 then qty else 0 end) as totalQty1,"+
         "sum(case when foodType=2 then qty else 0 end) as totalQty2,"+
         "sum(case when foodType=3 then qty else 0 end) as totalQty3 "+
         "from foodSalesTable group by foodType");
      try {
         ResultSet rs = st.executeQuery();
         try {
            rs.next();
            q0 = rs.getInt("totalQty0");
            q1 = rs.getInt("totalQty1");
            q2 = rs.getInt("totalQty2");
            q3 = rs.getInt("totalQty3");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
}
```

The variable names q0-q3 can be made better to eliminate the comments. The sum(...) SQL expression is duplicated.

```
class FoodSalesReport {
   int qtyRiceSold;
   int qtyNoodleSold;
   int qtyDrinkSold;
   int qtyDessertSold;
   void LoadData(Connection conn) {
      String sqlExprList = "";
      for (int i = 0; i <= 3; i++) {
         String separator = (i==0) ? "" : ",";
         sqlExprList = sqlExprList+
                  separator+
                  getSQLForSoldQtyForFoodType(i);
      }
      PreparedStatement st = conn.prepareStatement("select "+
         sqlExprList+
         "from foodSalesTable group by foodType");
      try {
         ResultSet rs = st.executeQuery();
         try {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            rs.next();
            qtyRiceSold = rs.getInt("totalQty0");
            qtyNoodleSold = rs.getInt("totalQty1");
            qtyDrinkSold = rs.getInt("totalQty2");
            qtyDessertSold = rs.getInt("totalQty3");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
   String getSQLForSoldQtyForFoodType(int foodType) {
      return "sum(case when foodType="+foodType+
            " then qty else 0 end) as totalQty"+foodType;
   }
}
```

If you are still concerned with the type code and the similarity between these food types, do something like:

```
class FoodType {
   int typeCode;
   FoodType(int typeCode) {
      ...
   }
   static FoodType foodTypeRice = new FoodType(0);
   static FoodType foodTypeNoodle = new FoodType(1);
   static FoodType foodTypeDrink = new FoodType(2);
   static FoodType foodTypeDessert = new FoodType(3);
   static FoodType knownFoodTypes[] =
      { foodTypeRice, foodTypeNoodle, foodTypeDrink, foodTypeDessert };
}
class FoodSalesReport {
   HashMap foodTypeToQtySold;
   void LoadData(Connection conn) {
      FoodType knownFoodTypes[] = FoodType.knownFoodTypes;
      String sqlExprList = "";
      for (int i = 0; i < knownFoodTypes.length; i++) {
         String separator = (i==0) ? "" : ",";
         sqlExprList = sqlExprList+
                  separator+
                  getSQLForSoldQtyForFoodType(knownFoodTypes[i]);
      }
      PreparedStatement st = conn.prepareStatement("select "+
         sqlExprList+
         "from foodSalesTable group by foodType");
      try {
         ResultSet rs = st.executeQuery();
         try {
            rs.next();
            for (int i = 0; i < knownFoodTypes.length; i++) {
               FoodType foodType = knownFoodTypes[i];
               int qty = rs.getInt(getQtyFieldNameForFoodType(foodType));
               foodTypeToQtySold.put(foodType, new Integer(qty));
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
   static String getQtyFieldNameForFoodType(FoodType foodType) {
      return "totalQty"+foodType.getCode();
   }
   String getSQLForSoldQtyForFoodType(FoodType foodType) {
      return "sum(case when foodType="+foodType.getCode()+
            " then qty else 0 end) as "+
            getQtyFieldNameForFoodType(foodType);
   }
}
```

2. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```
class SurveyData {
   String path; //save the data to this file.
   boolean hidden; //should the file be hidden?
   //set the path to save the data according to the type of data (t).
   void setSavePath(int t) {
      if (t==0) { //raw data.
         path = "c:/application/data/raw.dat";
         hidden = true;
      } else if (t==1) {  //cleaned up data.
         path = "c:/application/data/cleanedUp.dat";
         hidden = true;
      } else if (t==2) {  //processed data.
         path = "c:/application/data/processed.dat";
         hidden = true;
      } else if (t==3) {  //data ready for publication.
         path = "c:/application/data/publication.dat";
         hidden = false;
      }
   }
}
```

The folder "c:/application/data" is duplicated. The ".dat" is duplicated. The way the path is set is duplicated. The setting of hidden is duplicated. The comments should be turned into code. The type code should be eliminated. Each type code value should be replaced by an object.

```
class SurveyDataType {
   String baseFileName;
   boolean hideDataFile;
   SurveyDataType(String baseFileName, boolean hideDataFile) {
      this.baseFileName = baseFileName;
      this.hideDataFile = hideDataFile;
   }
   String getSavePath() {
      return "c:/application/data/"+baseFileName+".dat";
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   static SurveyDataType rawDataType =
      new SurveyDataType("raw", true);
   static SurveyDataType cleanedUpDataType =
      new SurveyDataType("cleanedUp", true);
   static SurveyDataType processedDataType =
      new SurveyDataType("processed", true);
   static SurveyDataType publicationDataType =
      new SurveyDataType("publication", false);
}
```

3. Point out and remove the code smells in the code (this example was inspired by the one given by Eugen):

```
class CustomersInDB {
   Connection conn;
   Customer getCustomer(String IDNumber) {
      PreparedStatement st = conn.prepareStatement(
         "select * from customer where ID=?");
      try {
         st.setString(1,
               IDNumber.replace('-', ' ').
                  replace('(', ' ').
                  replace(')', ' ').
                  replace('/', ' '));
         ResultSet rs = st.executeQuery();
         ...
      } finally {
         st.close();
      }
   }
   void addCustomer(Customer customer) {
      PreparedStatement st = conn.prepareStatement(
         "insert into customer values(?,?,?,?)");
      try {
         st.setString(1,
               customer.getIDNumber().replace('-', ' ').
                      replace('(', ' ').
                      replace(')', ' ').
                      replace('/', ' '));
         st.setString(2, customer.getName());
         ...
         st.executeUpdate();
         ...
      } finally {
         st.close();
      }
   }
}
```

The processing of the ID number is duplicated. The calls to replace are similar.

```
class CustomersInDB {
   Connection conn;
   String replaceSymbolsInID(String idNumber) {
      String symbolsToReplace = "-()/";
      for (int i = 0; i < symbolsToReplace.length(); i++) {
         idNumber = idNumber.replace(symbolsToReplace.charAt(i), ' ');
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      }
      return idNumber;
   }
   Customer getCustomer(String IDNumber) {
      PreparedStatement st = conn.prepareStatement(
         "select * from customer where ID=?");
      try {
         st.setString(1, replaceSymbolsInID(IDNumber));
         ResultSet rs = st.executeQuery();
         ...
      } finally {
         st.close();
      }
   }
   void addCustomer(Customer customer) {
      PreparedStatement st = conn.prepareStatement(
         "insert into customer values(?,?,?,?)");
      try {
         st.setString(1, replaceSymbolsInID(customer.getIDNumber()));
         st.setString(2, customer.getName());
         ...
         st.executeUpdate();
         ...
      } finally {
         st.close();
      }
   }
}
```

4. The code below contains duplicate code: the loop in printOverdueRentals and in countOverdueRentals. If you need to remove this duplication at any cost, how do you do it?

```
class BookRentals {
   Vector rentals;
   int countRentals() {
      return rentals.size();
   }
   BookRental getRentalAt(int i) {
      return (BookRental)rentals.elementAt(i);
   }
   void printOverdueRentals() {
      int i;
      for (i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            System.out.println(rental.toString());
      }
   }
   int countOverdueRentals() {
      int i, count;
      count=0;
      for (i=0; i<countRentals(); i++)
         if (getRentalAt(i).isOverdue())
            count++;
      return count;
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

First, make the code in both loop bodies look identical:

```
class BookRentals {
   ...
   void printOverdueRentals() {
      int i;
      for (i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            process the rental;
      }
   }
   int countOverdueRentals() {
      int i, count;
      count=0;
      for (i=0; i<countRentals(); i++)
         if (getRentalAt(i).isOverdue())
            process the rental;
      return count;
   }
}
```

As the two loops are identical, extract the loop into a method. Let the two existing methods call this new method:

```
class BookRentals {
   ...
   void yyy() {
      for (int i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            process the rental;
      }
   }
   void printOverdueRentals() {
      yyy();
   }
   int countOverdueRentals() {
      int count;
      yyy();
      return count;
   }
}
```

To see what is a good name for this method, consider what this method does. From the code in this method, it can be seen that it loops through each rental and process each overdue rental. So, "processOverdueRentals" should be a good name for this method:

```
class BookRentals {
   ...
   void processOverdueRentals() {
      for (int i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            process the rental;
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
   void printOverdueRentals() {
      processOverdueRentals();
   }
   int countOverdueRentals() {
      int count;
      processOverdueRentals();
      return count;
   }
}
```

Because there are two implementations for the "process the rental" method, create an interface to allow for different implementations:

```
interface XXX {
   void process(BookRental rental);
}
class BookRentals {
   ...
   void processOverdueRentals(XXX obj) {
      for (int i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            obj.process(rental);
      }
   }
   ...
}
```

To see what is a good name for this interface, consider what this interface does. From the code in this interface, it is clear that it has only one method and that method processes a rental. So, "RentalProcessor" should be a good name for this interface:

```
interface RentalProcessor {
   void process(BookRental rental);
}
class BookRentals {
   ...
   void processOverdueRentals(RentalProcessor processor) {
      for (int i=0; i<countRentals(); i++) {
         BookRental rental = getRentalAt(i);
         if (rental.isOverdue())
            processor.process(rental);
      }
   }
   ...
}
```

printOverdueRentals and countOverdueRentals need to provide their respective implementations:

```
class RentalPrinter implements RentalProcessor {
   void process(BookRental rental) {
      System.out.println(rental.toString());
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class RentalCounter implements RentalProcessor {
   int count = 0;
   void process(BookRental rental) {
      count++;
   }
}
class BookRentals {
   ...
   void processOverdueRentals(RentalProcessor processor) {
      ...
   }
   void printOverdueRentals() {
      RentalPrinter rentalPrinter=new RentalPrinter();
      processOverdueRentals(rentalPrinter);
   }
   int countOverdueRentals() {
      RentalCounter rentalCounter=new RentalCounter();
      processOverdueRentals(rentalCounter);
      return rentalCounter.count;
   }
}
```

Use anonymous inner class or inner class if possible:

```
class BookRentals {
   ...
   void printOverdueRentals() {
      processOverdueRentals(new RentalProcessor() {
         void process(BookRental rental) {
            System.out.println(rental.toString());
         }
      });
   }
   int countOverdueRentals() {
      class RentalCounter implements RentalProcessor {
         int count = 0;
         void process(BookRental rental) {
            count++;
         }
      }
      RentalCounter rentalCounter = new RentalCounter();
      processOverdueRentals(rentalCounter);
      return rentalCounter.count;
   }
}
```

5. Point out and remove the code smells in the code:

```
class Currency {
   final public int USD=0;
   final public int RMB=1;
   final public int ESCUDO=2;  //Portuguese currency
   private int currencyCode;
   public Currency(int currencyCode) {
      this.currencyCode=currencyCode;
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public String format(int amount) {
      switch (currencyCode) {
         case USD:
            //return something like $1,200
         case RMB:
            //return something like RMB1,200
         case ESCUDO:
            //return something like $1.200
      }
   }
}
```

The use of type code is bad. Convert each type code into a class.

```
interface Currency {
   public String format(int amount);
}
class USDCurrency implements Currency {
   public String format(int amount) {
      //return something like $1,200
   }
}
class RMBCurrency implements Currency {
   public String format(int amount) {
      //return something like RMB1,200
   }
}
class ESCUDOCurrency implements Currency {
   public String format(int amount) {
      //return something like $1.200
   }
}
```

6. Point out and remove the code smells in the code:

```
class Payment {
   final static String FOC = "FOC";  //free of charge.
   final static String TT = "TT";  //paid by telegraphic transfer.
   final static String CHEQUE = "Cheque";  //paid by cheque.
   final static String CREDIT_CARD = "CreditCard"; //paid by credit card.
   final static String CASH = "Cash";  //paid by cash.
   //type of payment. Must be one of the above constant.
   String paymentType;
   Date paymentDate; //if FOC, the date the fee is waived.
   int actualPayment; //if FOC, it is 0.
   int discount;  //if FOC, the amount that is waived.
   String bankName; //if it is by TT, cheque or credit card.
   String chequeNumber; //if it is by cheque.
   //if it is by credit card.
   String creditCardType;
   String creditCardHolderName;
   String creditCardNumber;
   Date creditCardExpiryDate;
   int getNominalPayment() {
      return actualPayment+discount;
   }
   String getBankName() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      if (paymentType.equals(TT) ||
         paymentType.equals(CHEQUE) ||
         paymentType.equals(CREDIT_CARD)) {
         return bankName;
      }
      else {
         throw new Exception("bank name is undefined for this payment type");
      }
   }
}
```

The use of type code is bad. Most attributes are not always used. We should convert each type code into a class. The actualAmount and discount are duplicated in several payment types, so extract them to form a common parent class. Some comments can be turned into code.

```
abstract class Payment {
   Date paymentDate;
   abstract int getNominalPayment();
}
class FOCPayment extends Payment {
   int amountWaived;
   int getNominalPayment() {
      return amountWaived;
   }
}
class RealPayment extends Payment {
   int actualPayment;
   int discount;
   int getNominalPayment() {
      return actualPayment+discount;
   }
}
class TTPayment extends RealPayment {
   String bankName;
   String getBankName() {
      return bankName;
   }
}
class ChequePayment extends RealPayment {
   String bankName;
   String chequeNumber;
   String getBankName() {
      return bankName;
   }
}
class CreditCardPayment extends RealPayment {
   String bankName;
   String cardType;
   String cardHolderName;
   String cardNumber;
   Date expiryDate;
   String getBankName() {
      return bankName;
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class CashPayment extends RealPayment {
}
```

Note that the bankName is still duplicated in several payment types. We may extract it to create a parent class, but it is quite difficult to think of a good name for this parent class (BankPayment? PaymentThroughBank?). As it is just a single variable, the motivation to extract it to form a class is not very strong either. So it is up to you to decide to extract it or not.

7. In the above application, there is a dialog to allow the user edit a payment object. The user can choose the payment type in a combo box. Then the related components for that payment type will be displayed. This function is provided by the CardLayout in Java. The code is shown below. Update the code accordingly.

```
class EditPaymentDialog extends JDialog {
   Payment newPayment;  //new payment to be returned.
   JPanel sharedPaymentDetails;
   JPanel uniquePaymentDetails;
   JTextField paymentDate;
   JComboBox paymentType;
   JTextField discountForFOC;
   JTextField bankNameForTT;
   JTextField actualAmountForTT;
   JTextField discountForTT;
   JTextField bankNameForCheque;
   JTextField chequeNumberForCheque;
   JTextField actualAmountForCheque;
   JTextField discountForCheque;
   ...

   EditPaymentDialog() {
      //setup the components.
      Container contentPane = getContentPane();
      String comboBoxItems[] = { //available payment types.
         Payment.FOC,
         Payment.TT,
         Payment.CHEQUE,
         Payment.CREDIT_CARD,
         Payment.CASH
      };
      //setup the components for the details shared by all types of payment.
      sharedPaymentDetails = new JPanel();
      paymentDate = new JTextField();
      paymentType = new JComboBox(comboBoxItems);
      sharedPaymentDetails.add(paymentDate);
      sharedPaymentDetails.add(paymentType);
      contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
      //setup the unique components for each type of payment.
      uniquePaymentDetails = new JPanel();
      uniquePaymentDetails.setLayout(new CardLayout());
      //setup a panel for FOC payment.
      JPanel panelForFOC = new JPanel();
      discountForFOC = new JTextField();
      panelForFOC.add(discountForFOC);
      uniquePaymentDetails.add(panelForFOC, Payment.FOC);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      //setup a panel for TT payment.
      JPanel panelForTT = new JPanel();
      bankNameForTT = new JTextField();
      actualAmountForTT = new JTextField();
      discountForTT = new JTextField();
      panelForTT.add(bankNameForTT);
      panelForTT.add(actualAmountForTT);
      panelForTT.add(discountForTT);
      uniquePaymentDetails.add(panelForTT, Payment.TT);
      //setup a panel for cheque payment.
      JPanel panelForCheque = new JPanel();
      bankNameForCheque = new JTextField();
      chequeNumberForCheque = new JTextField();
      actualAmountForCheque = new JTextField();
      discountForCheque = new JTextField();
      panelForCheque.add(bankNameForCheque);
      panelForCheque.add(chequeNumberForCheque);
      panelForCheque.add(actualAmountForCheque);
      panelForCheque.add(discountForCheque);
      uniquePaymentDetails.add(panelForCheque, Payment.CHEQUE);
      //setup a panel for credit card payment.
      ...
      //setup a panel for cash payment.
      ...
      contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
   }
Payment editPayment(Payment payment) {
      displayPayment(payment);
      setVisible(true);
      return newPayment;
   }
   void displayPayment(Payment payment) {
      paymentDate.setText(payment.getDateAsString());
      paymentType.setSelectedItem(payment.getType());
      if (payment.getType().equals(Payment.FOC)) {
         discountForFOC.setText(Integer.toString(payment.getDiscount()));
      }
      else if (payment.getType().equals(Payment.TT)) {
         bankNameForTT.setText(payment.getBankName());
         actualAmountForTT.setText(
            Integer.toString(payment.getActualAmount()));
         discountForTT.setText(Integer.toString(payment.getDiscount()));
      }
      else if (payment.getType().equals(Payment.CHEQUE)) {
         bankNameForCheque.setText(payment.getBankName());
         chequeNumberForCheque.setText(payment.getChequeNumber());
         actualAmountForCheque.setText(
            Integer.toString(payment.getActualAmount()));
         discountForCheque.setText(Integer.toString(payment.getDiscount()));
      }
      else if (payment.getType().equals(Payment.CREDIT_CARD)) {
         //...
      }
      else if (payment.getType().equals(Payment.CASH)) {
         //...
      }
   }
   //when the user clicks OK.
   void onOK() {
      newPayment = makePayment();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      dispose();
   }
   //make a payment from the components.
   Payment makePayment() {
      String paymentTypeString = (String) paymentType.getSelectedItem();
      Payment payment = new Payment(paymentTypeString);
      payment.setDateAsText(paymentDate.getText());
      if (paymentTypeString.equals(Payment.FOC)) {
         payment.setDiscount(Integer.parseInt(discountForFOC.getText()));
      }
      else if (paymentTypeString.equals(Payment.TT)) {
         payment.setBankName(bankNameForTT.getText());
         payment.setActualAmount(
            Integer.parseInt(actualAmountForTT.getText()));
         payment.setDiscount(
            Integer.parseInt(discountForTT.getText()));
      }
      else if (paymentTypeString.equals(Payment.CHEQUE)) {
         payment.setBankName(bankNameForCheque.getText());
         payment.setChequeNumber(chequeNumberForCheque.getText());
         payment.setActualAmount(
            Integer.parseInt(actualAmountForCheque.getText()));
         payment.setDiscount(
            Integer.parseInt(discountForCheque.getText()));
      }
      else if (paymentTypeString.equals(Payment.CREDIT_CARD)) {
         //...
      }
      else if (paymentTypeString.equals(Payment.CASH)) {
         //...
      }
      return payment;
   }
}
```

Some comments can be turned into code. The long if-then-else-if in the displayPayment method and the makePayment method in the EditPaymentDialog class are no good.

To eliminate these if-then-else-ifs, we should create a UI class for each payment class such as FOCPaymentUI, TTPaymentUI and etc. They should inherit/implement something like a PaymentUI. The PaymentUI have methods like tryToDisplayPayment (Payment payment) and makePayment(). The PaymentUI construtor should take the paymentDate JTextField as an argument because all sub-classes need it. Each sub-class should implement the toString method so that we can replace the String array by a PaymentUI array for the paymentType combo box (combo box in java can contain Objects, not just Strings).

```
abstract class PaymentUI {
   JTextField paymentDate;
   PaymentUI(JTextField paymentDate) {
      ...
   }
   void displayPaymentDate(Payment payment) {
      paymentDate.setText(
         new SimpleDateFormat().format(payment.getPaymentDate()));
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
   Date makePaymentDate() {
      //parse the text in paymentDate and return a date;
   }
   abstract boolean tryToDisplayPayment(Payment payment);
   abstract Payment makePayment();
   abstract JPanel getPanel();
}
abstract class RealPaymentUI extends PaymentUI {
   JTextField actualPayment;
   JTextField discount;
   RealPaymentUI(JTextField paymentDate) {
      super(paymentDate);
      actualPayment = new JTextField();
      discount = new JTextField();
   }
   void displayActualPayment(RealPayment payment) {
      actualPayment.setText(Integer.toString(payment.getActualPayment()));
   }
   void displayDiscount(RealPayment payment) {
      discount.setText(Integer.toString(payment.getDiscount()));
   }
   int makeActualPayment() {
      //parse the text in actualPayment and return an int;
   }
   int makeDiscount() {
      //parse the text in discount and return an int;
   }
}
class TTPaymentUI extends RealPaymentUI {
   JPanel panel;
   JTextField bankName;
   TTPaymentUI(JTextField paymentDate) {
      super(paymentDate);
      panel = ...;
      bankName = ...;
      //add bankName, actualPayment, discount to panel.
   }
   boolean tryToDisplayPayment(Payment payment) {
      if (payment instanceof TTPayment) {
         TTPayment ttpayment = (TTPayment)payment;
         displayPaymentDate(payment);
         displayActualPayment(ttpayment);
         displayDiscount(ttpayment);
         bankName.setText(ttpayment.getBankName());
         return true;
      }
      return false;
   }
   Payment makePayment() {
      return new TTPayment(makePaymentDate(),
                  makeActualPayment(),
                  makeDiscount(),
                  bankName.getText());
   }
   String toString() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      return "TT";
   }
   JPanel getPanel() {
      return panel;
   }
}
class FOCPaymentUI extends PaymentUI {
   ...
}
class ChequePaymentUI extends RealPaymentUI {
   ...
}
class EditPaymentDialog extends JDialog {
   Payment newPaymentToReturn;
   JPanel sharedPaymentDetails;
   JTextField paymentDate;
   JComboBox paymentType;
   PaymentUI paymentUIs[];

   EditPaymentDialog() {
      setupComponents();
   }
   void setupComponents() {
      setupComponentsSharedByAllPaymentTypes();
      setupPaymentUIs();
      setupPaymentTypeIndicator();
      setupComponentsUniqueToEachPaymentType();
   }
   void setupComponentsSharedByAllPaymentTypes() {
      paymentDate = new JTextField();
      sharedPaymentDetails = new JPanel();
      sharedPaymentDetails.add(paymentDate);
      Container contentPane = getContentPane();
      contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
   }
   void setupPaymentUIs() {
      paymentUIs[0] = new TTPaymentUI(paymentDate);
      paymentUIs[1] = new FOCPaymentUI(paymentDate);
      paymentUIs[2] = new ChequeaymentUI(paymentDate);
      ...
   }
   void setupPaymentTypeIndicator() {
      paymentType = new JComboBox(paymentUIs);
      sharedPaymentDetails.add(paymentType);
   }
   void setupComponentsUniqueToEachPaymentType() {
      JPanel uniquePaymentDetails = new JPanel();
      uniquePaymentDetails.setLayout(new CardLayout());
      for (int i = 0; i < paymentUIs.length; i++) {
         PaymentUI UI = paymentUIs[i];
         uniquePaymentDetails.add(UI.getPanel(), UI.toString());
      }
      Container contentPane = getContentPane();
      contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
   }
   Payment editPayment(Payment payment) {
      displayPayment(payment);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      setVisible(true);
      return newPaymentToReturn;
   }
   void displayPayment(Payment payment) {
      for (int i = 0; i < paymentUIs.length; i++) {
         PaymentUI UI = paymentUIs[i];
         if (UI.tryToDisplayPayment(payment)) {
            paymentType.setSelectedItem(UI);
         }
      }
   }
   void onOK() {
      newPaymentToReturn = makePayment();
      dispose();
   }
   Payment makePayment() {
      PaymentUI UI = (PaymentUI)paymentType.getSelectedItem();
      return UI.makePayment();
   }
}
```

8. This is an embedded application controlling a cooker. In every second, it will check if the cooker is over-heated (e.g., short-circuited). If yes it will cut itself off the power and make an alarm using its built-in speaker. It will also check if the moisture inside is lower than a certain threshold (e.g., the rice is cooked). If yes, it will turn its built-in heater to 50 degree Celsius just to keep the rice warm. In the future, you expect that some more things will need to be done in every second.

   Point out and remove the problem in the code.

```
class Scheduler extends Thread {
   Alarm alarm;
   HeatSensor heatSensor;
   PowerSupply powerSupply;
   MoistureSensor moistureSensor;
   Heater heater;
   public void run() {
      for (;;) {
         Thread.sleep(1000);
         //check if it is overheated.
         if (heatSensor.isOverHeated()) {
            powerSupply.turnOff();
            alarm.turnOn();
         }
         //check if the rice is cooked.
         if (moistureSensor.getMoisture()<60) {
            heater.setTemperature(50);
         }
      }
   }
}
```

   The run method will keep growing and growing. To stop it, we should make the code checking for overheat look identical to the code checking for cooked rice (in a certain abstraction level):

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class Scheduler extends Thread {
   public void run() {
      for (;;) {
         Thread.sleep(1000);
         //check if it is overheated.
         do some task;
         //check if the rice is cooked.
         do some task;
      }
   }
}
```

Define a an interface such as Task and two implementation classes doing the task differently:

```
interface Task {
   void doIt();
}
class Scheduler extends Thread {
   Task tasks[];
   void registerTask(Task task) {
      //add task to tasks;
   }
   public void run() {
      for (;;) {
         Thread.sleep(1000);
         for (int i = 0; i < tasks.length; i++) {
            tasks[i].doIt();
         }
      }
   }
}
class OverHeatCheckTask implements Task {
   Alarm alarm;
   PowerSupply powerSupply;
   HeatSensor heatSensor;
   void doIt() {
      if (heatSensor.isOverHeated()) {
         powerSupply.turnOff();
         alarm.turnOn();
      }
   }
}
class RickCookedCheckTask implements Task {
   Heater heater;
   MoistureSensor moistureSensor;
   void doIt() {
      if (moistureSensor.getMoisture()<60) {
         heater.setTemperature(50);
      }
   }
}
class Cooker {
   public static void main(String args[]) {
      Scheduler scheduler = new Scheduler();
      scheduler.registerTask(new OverHeatCheckTask());
      scheduler.registerTask(new RickCookedCheckTask());
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
}
```

9. This application is concerned with the training courses. The schedule of a course can be expressed in three ways (as of now): weekly, range or list. A weekly schedule is like "every Tuesday for 5 weeks starting from Oct. 22". A range schedule is like "Every day from Oct. 22 to Nov. 3". A list schedule is like "Oct. 22, Oct. 25, Nov. 3, Nov. 10". In this exercise we will ignore the time and just assume that it is always 7:00pm-10:00pm. It is expected that new ways to express the schedule may be added in the future.

   Point out and remove the code smells in the code:

```
class Course {
   static final int WEEKLY=0;
   static final int RANGE=1;
   static final int LIST=2;
   String courseTitle;
   int scheduleType;  // WEEKLY, RANGE, or LIST
   int noWeeks;       // For WEEKLY.
   Date fromDate;     // for WEEKLY and RANGE.
   Date toDate;       // for RANGE.
   Date dateList[];   // for LIST.

   int getDurationInDays() {
      switch (scheduleType) {
         case WEEKLY:
            return noWeeks;
         case RANGE:
            int msInOneDay = 24*60*60*1000;
            return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
         case LIST:
            return dateList.length;
         default:
            return 0;  // unknown schedule type!
      }
   }
   void printSchedule() {
      switch (scheduleType) {
         case WEEKLY:
            //...
         case RANGE:
            //...
         case LIST:
            //...
      }
   }
}
```

   The use of type code is bad. Some instance variables are not used all the time. We should turn each type code value into a sub-class. We are talking about different types of schedules, not really different types of courses.

```
interface Schedule {
   int getDurationInDays();
   void print();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class WeeklySchedule implements Schedule {
   int noWeeks;
   Date fromDate;
   Date toDate;
   int getDurationInDays() {
      return noWeeks;
   }
   void print() {
      ...
   }
}
class RangeSchedule implements Schedule {
   Date fromDate;
   Date toDate;
   int getDurationInDays() {
      int msInOneDay = 24*60*60*1000;
      return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
   }
   void print() {
      ...
   }
}
class ListSchedule implements Schedule {
   Date dateList[];
   int getDurationInDays() {
      return dateList.length;
   }
   void print() {
      ...
   }
}
class Course {
   String courseTitle;
   Schedule schedule;
   int getDurationInDays() {
      return schedule.getDurationInDays();
   }
   void printSchedule() {
      schedule.print();
   }
}
```

10. This application is concerned with training courses. A course has a title, a fee and a list of sessions. However, sometimes a course can consist of several modules, each of which is a course. For example, there may be a compound course "Fast track to becoming a web developer" which consists of three modules: a course named "HTML", a course named "FrontPage" and a course named "Flash". It is possible for a module to consist of some other modules. If a course consists of modules, its fee and schedule are totally determined by that of its modules and thus it will not maintain its list of sessions.

    Point out and remove the code smells in the code:

```
class Session {
   Date date;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   int startHour;
   int endHour;
   int getDuration() {
      return endHour-startHour;
   }
}
class Course {
   String courseTitle;
   Session sessions[];
   double fee;
   Course modules[];

   Course(String courseTitle, double fee, Session sessions[]) {
      //...
   }
   Course(String courseTitle, Course modules[]) {
      //...
   }
   String getTitle() {
      return courseTitle;
   }
   double getDuration() {
      int duration=0;
      if (modules==null)
         for (int i=0; i<sessions.length; i++)
            duration += sessions[i].getDuration();
      else
         for (int i=0; i<modules.length; i++)
            duration += modules[i].getDuration();
      return duration;
   }
   double getFee() {
      if (modules==null)
         return fee;
      else {
         double totalFee = 0;
         for (int i=0; i<modules.length; i++)
            totalFee += modules[i].getFee();
         return totalFee;
      }
   }
   void setFee(int fee) throws Exception {
      if (modules==null)
         this.fee = fee;
      else
         throw new Exception("Please set the fee of each module one by one");
   }
}
```

Some instance variables are not used all the time. There are actually two types of courses: those containing modules and those not containing modules. The if-then-else type checking is duplicated in many methods. We should separate each type into a sub-class. Some methods like setFee is applicable to one type only and should be declared in that sub-class.

```
class Session {
   Date date;
   int startHour;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   int endHour;
   int getDuration() {
      return endHour-startHour;
   }
}
abstract class Course {
   String courseTitle;
   Course(String courseTitle) {
      ...
   }
   String getTitle() {
      return courseTitle;
   }
   abstract double getFee();
   abstract double getDuration();
}
class SimpleCourse extends Course {
   Session sessions[];
   double fee;
   SimpleCourse(String courseTitle, double fee, Session sessions[]) {
      ...
   }
   double getFee() {
      return fee;
   }
   double getDuration() {
      int duration=0;
      for (int i=0; i<sessions.length; i++) {
         duration += sessions[i].getDuration();
      }
      return duration;
   }
   void setFee(int fee) {
      this.fee = fee;
   }
}
class CompoundCourse extends Course {
   Course modules[];
   CompoundCourse(String courseTitle, Course modules[]) {
      ...
   }
   double getFee() {
      double totalFee = 0;
      for (int i=0; i<modules.length; i++) {
         totalFee += modules[i].getFee();
      }
      return totalFee;
   }
   double getDuration() {
      int duration=0;
      for (int i=0; i<modules.length; i++) {
         duration += modules[i].getDuration();
      }
      return duration;
   }
}
```

11. Point out and remove the code smells in the code:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class BookRental {
   String bookTitle;
   String author;
   Date rentDate;
   Date dueDate;
   double rentalFee;
   boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   double getTotalFee() {
      return isOverdue() ? 1.2*rentalFee : rentalFee;
   }
}
class MovieRental {
   String movieTitle;
   int classification;
   Date rentDate;
   Date dueDate;
   double rentalFee;
   boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   double getTotalFee() {
      return isOverdue() ? Math.max(1.3*rentalFee, rentalFee+20) : rentalFee;
   }
}
```

Some instance variables and some methods are duplicated in both classes. They should be extracted into a parent class like Rental. The getTotalFee method in both classes have the same structure. Make them look identical and then extract it into the parent class.

```
abstract class Rental {
   Date rentDate;
   Date dueDate;
   double rentalFee;
   boolean isOverdue() {
      Date now=new Date();
      return dueDate.before(now);
   }
   double getTotalFee() {
      return isOverdue() ? getFeeWhenOverdue() : rentalFee;
   }
   abstract double getFeeWhenOverdue();
}
class BookRental extends Rental {
   String bookTitle;
   String author;
   double getFeeWhenOverdue() {
      return 1.2*rentalFee;
   }
}
class MovieRental extends Rental {
   String movieTitle;
   int classification;
   double getFeeWhenOverdue() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      return Math.max(1.3*rentalFee, rentalFee+20);
   }
}
```

12. Point out and remove the code smells in the code:

```
class Customer {
   String homeAddress;
   String workAddress;
}
class Order {
   String orderId;
   Restaurant restaurantReceivingOrder;
   Customer customerPlacingOrder;
   //"H": deliver to home address of the customer.
   //"W": deliver to work address of the customer.
   //"O": deliver to the address specified here.
   String addressType;
   String otherAddress; //address if addressType is "O".
   HashMap orderItems;

   public String getDeliveryAddress() {
      if (addressType.equals("H")) {
         return customerPlacingOrder.getHomeAddress();
      } else if (addressType.equals("W")) {
         return customerPlacingOrder.getWorkAddress();
      } else if (addressType.equals("O")) {
         return otherAddress;
      } else {
         return null;
      }
   }
}
```

The Order class contains a type code. The instance variables otherAddress is not always used. We should turn each type code value into a class:

```
class Customer {
   String homeAddress;
   String workAddress;
}
interface DeliveryAddress {
}
class HomeAddress implements DeliveryAddress {
   Customer customer;
   HomeAddress(Customer customer) {
      ...
   }
   String toString() {
      return customer.getHomeAddress();
   }
}
class WorkAddress implements DeliveryAddress {
   Customer customer;
   WorkAddress(Customer customer) {
      ...
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   String toString() {
      return customer.getWorkAddress();
   }
}
class SpecifiedAddress implements DeliveryAddress {
   String addressSpecified;
   SpecifiedAddress(String addressSpecified) {
      ...
   }
   String toString() {
      return addressSpecified;
   }
}
class Order {
   String orderId;
   Restaurant restaurantReceivingOrder;
   Customer customerPlacingOrder;
   DeliveryAddress deliveryAddress;
   HashMap orderItems;

   public String getDeliveryAddress() {
      return deliveryAddress.toString();
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 4

Keeping Code Fit

Example

This is a conference management system. It is used to manage the information of the conference participants. At the beginning, we only need to record the ID (assigned by the conference organizer), name, telephone and address of each participant. For this, we write the following code:

```
class Participant {
   String id;
   String name;
   String telNo;
   String address;
}
class ConferenceSystem {
   Participant participants[];
}
```

Later, there comes a new requirement: We need to record whether a participant requests the conference organizer to book a hotel for him. If yes, we also need to record the name of the hotel he selected, the check-in date, check-out date, room type (single or double). For this, we expand the above Participant class:

```
class Participant {
   String id;
   String name;
   String telNo;
   String address;
   boolean bookHotelForHim;
   String hotelName;
   Date checkInDate;
   Date checkOutDate;
   boolean isSingleRoom;
   void setHotelBooking(String hotelName, Date checkInDate, ...) {
      ...
   }
}
```

Later, there comes yet another new requirement: Record which seminars each participant is going to attend. For each seminar that he is going to attend, we need to record the registration date and whether he needs a set of simultaneous interpretation device. For this, we expand the Participant class again:

```
class Participant {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    String id;
    String name;
    String telNo;
    String address;
    boolean bookHotelForHim;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
    String idOfSeminarsRegistered[];
    Date seminarRegistrationDates[];
    boolean needSIDeviceForEachSeminar[];
    void setHotelBooking(String hotelName, Date checkInDate, ...) {
       ...
    }
    void registerForSeminar(String seminarId, Date regDate, boolean needSIDevice) {
       //add seminarId to idOfSeminarsRegistered.
       //add regDate to seminarRegistrationDates.
       //add needSIDevice to needSIDeviceForEachSeminar.
    }
    boolean isRegisteredForSeminar(String seminarId) {
       ...
    }
    Date getSeminarRegistrationDate(String seminarId) {
       ...
    }
    boolean needSIDeviceForSeminar(String seminarId) {
       ...
    }
    String [] getAllSeminarsRegistered() {
       return idOfSeminarsRegistered;
    }
}
```

The code is getting bloated

Note that this is already the second time that we expand the Participant class. Every time we expand it, it includes more code (instance variables and methods) and more functionality. Originally it had only 4 instance variables. Now it has 12! In addition, the number of concepts it deals with has increased greatly. Originally it only dealt with the basic information of a participant (name, address and etc.), now it also deals with the concepts of hotel, hotel booking, seminar, simultaneous interpretation and etc. If in the future more requirements come, we will expand the Participant class again, then how complicated and bloated it will become!

How to trim the Participant class? Or how to keep it fit from day one? Before answering these two questions, let's consider another question that must be answered first: Given a class, how to check whether it needs trimming?

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

How to check if a class needs trimming

To check if a class needs trimming, a subjective method is: After reading its code, do we feel that it is "too long" (a common code smell), "too complicated" or involves "too many" concepts? If yes, it needs trimming.

A simpler and objective method is, when we find that we are already expanding a class for the second or third time, we consider it needs trimming. This is a "lazy", "passive" method, but it is effective.

Now let's see how to trim the Participant class.

Extract the functionality about hotel booking

First we consider how to extract the functionality about hotel booking. A possible method is:

```
class Participant {
   String id;
   String name;
   String telNo;
   String address;
}
class HotelBooking {
   String participantId;
   String hotelName;
   Date checkInDate;
   Date checkOutDate;
   boolean isSingleRoom;
}
class HotelBookings {
   HotelBooking hotelBookings[];
   void addBooking(HotelBooking booking) {
      ...
   }
}
class ConferenceSystem {
   Participant participants[];
   HotelBookings hotelBookings;
}
```

Now, the Participant class knows absolutely nothing about hotel booking. Of course, we don't have to use an array to store the hotel bookings. For example, we may as well use a Map:

```
class Participant {
   String id;
   String name;
   String telNo;
   String address;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class HotelBooking {
   String participantId;
   String hotelName;
   Date checkInDate;
   Date checkOutDate;
   boolean isSingleRoom;
}
class HotelBookings {
   HashMap mapFromPartIdToHotelBooking;
   //Must provide participant's ID.
   void addBooking(String participantId, HotelBooking booking) {
      ...
   }
}
class ConferenceSystem {
   Participant participants[];
   HotelBookings hotelBookings;
}
```

The advantage of this method is that we can quickly find the hotel booking of a participant. Another possible method is:

```
class Participant {
   String id;
   String name;
   String telNo;
   String address;
   HotelBooking hotelBooking;
}
class HotelBooking {
   String hotelName;
   Date checkInDate;
   Date checkOutDate;
   boolean isSingleRoom;
}
class ConferenceSystem {
   Participant participants[];
}
```

Note that in this method the Participant class still has the concept of hotel booking. Just that the actual functionality has been extracted into the HotelBooking class. It is more straight forward to find the hotel booking of a participant. The cost we pay is that Participant still has the concept of hotel booking.

Of course, in addition to the methods above, there are many other possible methods.

Extract the functionality about seminar

Now we consider how to extract the functionality about seminar. A possible method is:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class Participant {
   String id;
   String name;
   String telNo;
   String address;
}
class SeminarRegistration {
   String participantId;
   String seminarId;
   Date registrationDate;
   boolean needSIDevice;
}
class SeminarRegistry {
   SeminarRegistration registrations[];
   void registerForSeminar(SeminarRegistration registration) {
      //add registration to registrations.
   }
   boolean isRegisteredForSeminar(String participantId, String seminarId) {
      ...
   }
   Date getSeminarRegistrationDate(String participantId, String seminarId) {
      ...
   }
   boolean needSIDeviceForSeminar(String participantId, String seminarId) {
      ...
   }
   SeminarRegistration[] getAllRegistrations(String participantId) {
      ...
   }
}
class ConferenceSystem {
   Participant participants[];
   SeminarRegistry seminarRegistry;
}
```

Of course, in addition to the methods above, there are many other possible methods.

The improved code

The improved code is shown below:

```
class Participant {
   String id;
   String name;
   String telNo;
   String address;
}
class HotelBooking {
   String participantId;
   String hotelName;
   Date checkInDate;
   Date checkOutDate;
   boolean isSingleRoom;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class HotelBookings {
   HotelBooking hotelBookings[];
   void addBooking(HotelBooking booking) {
      ...
   }
}
class SeminarRegistration {
   String participantId;
   String seminarId;
   Date registrationDate;
   boolean needSIDevice;
}
class SeminarRegistry {
   SeminarRegistration registrations[];
   void registerForSeminar(SeminarRegistration registration) {
      //add registration to registrations.
   }
   boolean isRegistered (String participantId, String seminarId) {
      ...
   }
   Date getRegistrationDate(String participantId, String seminarId) {
      ...
   }
   boolean needSIDevice(String participantId, String seminarId) {
      ...
   }
   SeminarRegistration[] getAllRegistrations(String participantId) {
      ...
   }
}
class ConferenceSystem {
   Participant participants[];
   HotelBookings hotelBookings;
   SeminarRegistry seminarRegistry;
}
```

References

The Single Responsibility Principle states: We should make a class that will change for one reason only. When a class includes many different types of functionality, we are obviously violating the Single Responsibility Principle. For more details, please see:

- http://www.objectmentor.com/resources/articles/srp.
- http://c2.com/cgi/wiki?OneResponsibilityRule.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. The code below still contains duplicate code: we create a prepared statement, set it up, execute it and then close it at several places. If you need to remove this duplication at any cost, how do you do it?

```
class ParticipantsInDB {
   Connection conn;
   final String tableName="participants";
   void addParticipant(Participant part){
      PreparedStatement st = conn.prepareStatement("INSERT INTO "+tableName+"
VALUES (?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         st.setString(2, part.getEFirstName());
         st.setString(3, part.getELastName());
         //...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteAllParticipants(){
      PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
      try {
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteParticipant(String participantId){
      PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName+"
WHERE id=?");
      try {
         st.setString(1, participantId);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

2. This application lets the network administrators document their server configurations. There are several types of servers as shown below. Design the code that implements a Swing GUI allowing the user to edit a server object. It must allow the user to change the

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

type of the server. It must use a CardLayout to display the components just right for the type of the server.

```
class Server {
   String id;
   String CPUModel;
}
class DNSServer extends Server {
   String domainName;
}
class WINSServer extends Server {
   String replicationPartner;
   int replicationInterval;
}
class DomainController extends Server {
   boolean remainNT4Compatible;
}
```

3. Point out and remove the code smells in the code below (this example is adpated from the one contributed by Eugen):

```
public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
   // mode value --> ADD, MOD, DEL
   String outResult = "";
   final String connName = "tmp";
   DBConnectionManager connMgr = DBConnectionManager.getInstance();
   Connection conP  = connMgr.getConnection(connName);
   Statement stmt = conP.createStatement();
   int upd = stmt.executeUpdate(sqlStmt);
   switch(upd) {
   case 1:
      if (mode.equalsIgnoreCase("ADD")) { outResult = "SUC Add"; }
      else if (mode.equalsIgnoreCase("MOD")) { outResult = "SUC Mod"; }
      else if (mode.equalsIgnoreCase("DEL")) { outResult = "SUC Del"; }
      break;
   case 0:
      if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
      else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
      else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
      break;
   default:
      if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
      else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
      else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
      break;
   }
   stmt.close();
   conP.close();
   return outResult;
}
```

4. Point out and remove the code smells in the code below (this example is adapted from the one contributed by Carol):

```
class Account {
   final public int SAVING=0;
   final public int CHEQUE=1;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   final public int FIXED=2;  //Portuguese currency
   private int accountType;
   private double balance;
   public double getInterestRate(...) { // Some method;
      ...
   }
   public Account(int accountType) {
      this.accountType=accountType;
   }
   public double calcInterest() {
      switch (accountType) {
         case SAVING:
            return balance*getInterestRate();
         case CHEQUE:
            return 0;
         case FIXED:
            return balance*(getInterestRate()+0.02);
      }
   }
}
```

5. Point out and remove the code smells in the code below (this example is contributed by Antonio):

```
class Department{
   final public int Account =0;
   final public int Marketing = 1;
   final public int CustomerServices = 2;
   protected int departmentCode;
   public Department(int departmentCode){
      this.departmentCode = departmentCode;
   }
   public String getDepartmentName(){
      switch (departmentCode){
         case Account:
            return "Account";
         case Marketing:
            return "Marketing";
         case CustomerServices:
            return "Customer Services";
      }
   }
}
```

6. Point out and remove the code smells in the code below (this example is contributed by Carita):

```
class NormalPayment {
   int units;
   double rate;
   final double TAX_RATE = 0.1;
   double getBillableAmount() {
      double baseAmt = units * rate;
      double tax = baseAmt * TAX_RATE;
      return baseAmt + tax;
   }
}
class PaymentForSeniorCitizen {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   int units;
   double rate;
   final double TAX_RATE = 0.1;
   double getBillableAmount() {
      double baseAmt = units * rate * 0.8;
      double tax = baseAmt * (TAX_RATE - 0.5) ;
      return baseAmt + tax;
   }
}
```

7. Point out and remove the code smells in the code below (this example is contributed by Malaquias):

```
class PianoKey {
   final static int key0 = 0;
   final static int key1 = 1;
   final static int key2 = 2;
   int keyNumber;
   public void playSound() {
      if (keyNumber == 0) {
         //play the frequency for key0
      }
      else if (keyNumber == 1) {
         //play the frequency for key1
      }
      else if (keyNumber == 2) {
         //play the frequency for key2
      }
   }
}
class Piano {
   Vector rythmn;
   public void play() {
      for(int i=0;i<rythmn.size();i++) {
         rythmn.getElementAt(i).playSound();
      }
   }
}
```

8. Point out and remove the code smells in the code below (this example is contributed by Frankie):

```
class Account {
   final static int LEVEL_USER = 1;
   final static int LEVEL_ADMIN = 2;
   int accountLevel;
   Date expiredDate; // for user account only
   boolean hasLogin; // for admin account only
}
class ERPApp {
   public boolean checkLoginIssue(Account account) {
      if (account.getLevel() == Account.LEVEL_USER) {
         // Check the account expired date
         Date now = new Date();
         if (account.getExpiredDate().before(now))
            return false;
         return true;
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      else if (account.getLevel() == Account.LEVEL_ADMIN) {
         // No expired date for admin account
         // Check multilogin
         if (account.hasLogin())
            return false;
         return true;
      }
      return false;
   }
}
```

9. Point out and remove the code smells in the code below (this example is adapted from the one contributed by YK):

```
class Form1 extends JDialog {
   JComboBox comboBoxReportType;
   Form1() {
      comboBoxReportType = new JComboBox();
      comboBoxReportType.addItem("r1");
      comboBoxReportType.addItem("r2");
      ...
      comboBoxReportType.addItem("r31c");
   }
   void processReport1() {
      //print some fancy report...
   }
   void processReport2() {
      //print another totally different fancy report...
   }
   ...
   void processReport31c() {
      //print yet another totally different fancy report...
   }
   void printReport(String repNo) {
      if (repNo.equals("r1"))
         processReport1();
      else if (repNo.equals("r2"))
         processReport2();
      ...
      else if (repNo.equals("r31c"))
         processReport31c();
   }
   void onPrintClick() {
      printReport((String) comboBoxReportType.getSelectedItem());
   }
}
```

10. This application is about restaurants. Initially we created a Restaurant class to represent a restaurant account and includes information such as its name, access password, tel and fax number, address. The class is like:

```
class Restaurant {
   String name;
   String password;
   String telNo;
   String faxNo;
   String address;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Later, the following requirements are added in sequence:

1. After initial registration, the restaurant account is assigned an activation code by the system. Only after the user enters the activation code, the account will become activated. Until then, the account is inactive and login is not allowed.

2. If the user would like to change the fax number of the account, the new fax number will not take effect immediately (the existing fax number will remain in effect). Instead, the account is assigned an activation code by the system. Only after the user enters the activation code, the new fax number will take effect.

3. A restaurant can be marked as in a certain category (e.g., Chinese restaurant, Portuguese restaurant and etc.). A category is identified by a category ID.

4. The user can input the holidays for the restaurant.

5. The user can input the business hours for the restaurant.

After implementing all these five requirements, the class has grown significantly and become quite complicated as shown below:

```
class Restaurant {
   String name;
   String password;
   String telNo;
   String faxNo;
   String address;
   String verificationCode;
   boolean isActivated;
   String faxNoToBeConfirmed;
   boolean isThereFaxNoToBeConfirmed = false;
   String catId;
   Vector holidays;
   Vector businessSessions;

   void activate(String verificationCode) {
      isActivated =(this.verificationCode.equals(verificationCode));
      if (isActivated && isThereFaxNoToBeConfirmed){
         faxNo = faxNoToBeConfirmed;
         isThereFaxNoToBeConfirmed = false;
      }
   }
   void setFaxNo(String newFaxNo) {
      faxNoToBeConfirmed = newFaxNo;
      isThereFaxNoToBeConfirmed = true;
      isActivated = false;
   }
   boolean isInCategory(String catId) {
      return this.catId.equals(catId);
   }
   void addHoliday(int year, int month, int day) {
      ...
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void removeHoliday(int year, int month, int day) {
      ...
   }
   boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin)
{
      ...
   }
   boolean isInBusinessHour(Calendar time) {
      ...
   }
   Vector getAllHolidays() {
      return holidays;
   }
   Vector getAllBusinessSessions() {
      return businessSessions;
   }
}
```

Your task is to implement the five requirements above in separate classes, leaving the original simple Restaurant class unchanged.

11. This application is about students. Originally we had a simple Student class as shown in the code:

```
class StudentManagementSystem {
   Student students[];
}
class Student {
   String studentId;
   String name;
   Date dateOfBirth;
}
```

Later, in order to record what courses the student has enrolled in, on which dates he enrolled and how he paid for them, we modified the code as:

```
class StudentManagementSystem {
   Student students[];
}
class Student {
   String studentId;
   String name;
   Date dateOfBirth;
   String courseCodes[]; //the student has enrolled in these courses.
   Date enrollDates[];  //for each enrolled course, the date he enrolled.
   Payment payments[];  //for each enrolled course, how he paid.

   void enroll(String courseCode, Date enrollDate, Payment payment) {
      //add courseCode to courseCodes
      //add enrollDate to enrollDates
      //add payment to Payments
   }
   void unenroll(String courseCode) {
      ...
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Your task is to implement this requirement without modifying the Student class.

12. This application lets the network administrators document their server configurations. Originally we had a simple Server class as shown in the code:

```
class Server {
   String name;
   String CPUModel;
   int RAMSizeInMB;
   int diskSizeInMB;
   InetAddress ipAddress;
}
class ServerConfigSystem {
   Server servers[];
}
```

Now, there are four new requirements. You are required to implement these requirements without modifying the Server class:

1. An administrator (identified by an admin ID) can be assigned to be responsible for a server.

2. We can check if a server is a DHCP server or not. If yes, we can record the address scope it manages (e.g., from 192.168.0.21 to 192.168.0.254).

3. We can check if a server is a file server or not. If yes, we can set and check the disk space quota allocated for each user (identified by a user ID).

4. A server can be a DHCP server and a file server at the same time.

13. At the moment the code of a system is shown below:

```
class Customer {
   String id;
   String name;
   String address;
}
class Supplier {
   String id;
   String name;
   String telNo;
   String address;
}
class SalesSystem {
   Vector customers;
   Vector suppliers;
}
```

Implement the following new requirements while keeping the code fit:

1. Customers can place orders. Each order has a unique ID. It records the date of the order, the ID of the customer and the name and quantity of each item being

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

ordered.

2. The system can list all the orders placed by a particular customer.

3. The system can list all the orders placed for a particular supplier.

4. A supplier can provide a certain discount to some of the customers it selects (valid for some selected items only). The discount for different customers or different items may be different.

14. Come up with a class that was originally slim but becomes bloated as new requirements are implemented. Then separate the code into different classes and restore the class to its original slim form.

Hints

1. It is similar to the RentalProcessor example. First, make the code look identical:

```
class ParticipantsInDB
   ...
   void addParticipant(Participant part){
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         set the parameters of st;
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteAllParticipants(){
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         set the parameters of st;
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

2. Refer to the EditPaymentDialog.

3. The different values of "mode" do NOT result in significantly different behaviors. Therefore you should NOT create a sub-class to represent each value (if you do, you will find these classes are pretty much the same). The same is true for "upd".

4. No hint for this one.

5. The difference values of "departmentCode" do NOT result in significantly different

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

behaviors. Therefore you should NOT create a sub-class to represent each value.

6. No hint for this one.

7. The difference between the keys is just the frequency. Therefore, use an object containing a int (the frequency) to represent each key. Do NOT create a class to represent each key.

8. No hint for this one.

9. No hint for this one.

10. The system should keep a list of restaurant objects. They have all been activated. If a new restaurant account is created and waiting to be activated, do not put a new restaurant object onto the list yet. Instead, create another object in the system. This object should contain all the information of a restaurant and have a method like activate(restaurant ID, verification code). When the user enters the verification code, the system should call this method (directly or indirectly). If everything is correct, this object should put the restaurant object (create it if required) onto the list. It should also remove itself from the system because it has finished its duty.

    To implement the new fax number activation, do something similar. Create an object in the system. When it is activated, it should set the fax number of the restaurant object and remove itself from the system.

    You will note that there is quite some duplicate code after implementing the two requirements above. Remove the duplication.

    The requirements about categories, holidays and business hours should be easy to implement.

11. No hint for this one.

12. To meet the 4th requirement, you can't use inheritance. Instead of a DHCPServer class, try creating a DHCPConfiguration class.

13. Create classes like Orders and Discounts.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. The code below still contains duplicate code: they all create a prepared statement, set it up, execute it and then close it. If you need to remove this duplication at any cost, how do you do it?

```
class ParticipantsInDB {
   Connection conn;
   final String tableName="participants";
   void addParticipant(Participant part){
      PreparedStatement st = conn.prepareStatement("INSERT INTO "+tableName+"
VALUES (?,?,?,?,?)");
      try {
         st.setString(1, part.getId());
         st.setString(2, part.getEFirstName());
         st.setString(3, part.getELastName());
         //...
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteAllParticipants(){
      PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
      try {
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteParticipant(String participantId){
      PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName+"
WHERE id=?");
      try {
         st.setString(1, participantId);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

First, make the code look identical:

```
class ParticipantsInDB {
   Connection conn;
   final String tableName="participants";
   void addParticipant(Participant part){
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         set the parameters of st;
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteAllParticipants(){
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         set the parameters of st;
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void deleteParticipant(String participantId){
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         set the parameters of st;
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

Then extract the duplicate code into a method:

```
class ParticipantsInDB {
   ...
   void ???() {
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         set the parameters of st;
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

To find a good name for this method, see what the code of the method actually does. In this case, It creates, executes and closes an SQL statement. So "executeSQL" should be a good name:

```
class ParticipantsInDB {
   ...
   void executeSQL() {
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         set the parameters of st;
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

As there are several ways to implement "set the parameters of st", we need to create an interface so that later we can create one implementation class for each implementation:

```
interface ??? {
   void setParametersOf(PreparedStatement st);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class ParticipantsInDB {
   ...
   void executeSQL(??? someObject) {
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         someObject.setParametersOf(st);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

To find a good name for this interface, see what this interface does. As it has only one method (setParametersOf) which sets the parameters of a prepared statement, "StatementParamsFiller" should be a good name:

```
interface StatementParamsSetter{
   void setParametersOf(PreparedStatement st);
}
class ParticipantsInDB {
   ...
   void executeSQL(StatementParamsSetter paramsSetter) {
      PreparedStatement st = conn.prepareStatement(some SQL);
      try {
         paramsSetter.setParametersOf(st);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

The "some SQL" can be represented by different data values (strings):

```
interface StatementParamsSetter{
   void setParametersOf(PreparedStatement st);
}
class ParticipantsInDB {
   ...
   void executeSQL(String SQL, StatementParamsSetter paramsSetter) {
      PreparedStatement st = conn.prepareStatement(SQL);
      try {
         paramsSetter.setParametersOf(st);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

Finally, provide different implementation classes. Make them inner classes if possible:

```
interface StatementParamsSetter{
   void setParametersOf(PreparedStatement st);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class ParticipantsInDB {
   ...
   void executeSQL(String SQL, StatementParamsSetter paramsSetter) {
      PreparedStatement st = conn.prepareStatement(SQL);
      try {
         paramsSetter.setParametersOf(st);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   void addParticipant(final Participant part){
      executeSQL("INSERT INTO "+tableName+" VALUES (?,?,?,?,?)",
         new StatementParamsSetter() {
            void setParametersOf(PreparedStatement st) {
               st.setString(1, part.getId());
               st.setString(2, part.getEFirstName());
               st.setString(3, part.getELastName());
            }
         });
   }
   void deleteAllParticipants(){
      executeSQL("DELETE FROM "+tableName,
         new StatementParamsSetter() {
            void setParametersOf(PreparedStatement st) {
            }
         });
   }
   void deleteParticipant(final String participantId){
      executeSQL("DELETE FROM "+tableName+" WHERE id=?",
         new StatementParamsSetter() {
            void setParametersOf(PreparedStatement st) {
               st.setString(1, participantId);
            }
         });
   }
}
```

2. This application lets the network administrators document their server configurations. There are several types of servers as shown below. Design the code that implements a Swing GUI allowing the user to edit a server object. It must allow the user to change the type of the server. It must use a CardLayout to display the components just right for the type of the server.

```
class Server {
   String id;
   String CPUModel;
}
class DNSServer extends Server {
   String domainName;
}
class WINSServer extends Server {
   String replicationPartner;
   int replicationInterval;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class DomainController extends Server {
   boolean remainNT4Compatible;
}
```

Create a UI class for each server class:

```
abstract class ServerUI {
   JPanel panelForThisServerType;
   JTextField id;
   JTextField CPUModel;
   ServerUI(JTextField id, JTextField CPUModel) {
      this.id = id;
      this.CPUModel = CPUModel;
      this.panelForThisServerType = new JPanel(...);
   }
   void showServerCommonInfo(Server server) {
      id.setText(server.getId());
      CPUModel.setText(server.getCPUModel());
   }
   String getIdFromUI() {
      return id.getText();
   }
   String getCPUModelFromUI() {
      return CPUModel.getText();
   }
   JPanel getPanel() {
      return panelForThisServerType;
   }
   abstract boolean tryToDisplayServer(Server server);
   abstract Server makeServer();
}
class DNSServerUI extends ServerUI {
   JTextField domainName;
   DNSServerUI(JTextField id, JTextField CPUModel) {
      super(id, CPUModel);
      domainName = new JTextField(...);
      panelForThisServerType.add(domainName);
   }
   String toString() {
      return "DNS";
   }
   boolean tryToDisplayServer(Server server) {
      if (server instanceof DNSServer) {
         DNSServer dnsServer = (DNSServer)server;
         showServerCommonInfo(server);
         domainName.setText(dnsServer.getDomainName());
         return true;
      }
      return false;
   }
   Server makeServer() {
      return new DNSServer(
            getIdFromUI(),
            getCPUModelFromUI(),
            domainName.getText());
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class WINSServerUI extends ServerUI {
   JTextField replicationPartner;
   JTextField replicationInterval;
   WINSServerUI(JTextField id, JTextField CPUModel) {
      super(id, CPUModel);
      replicationPartner = new JTextField(...);
      replicationInterval = new JTextField(...);
      panelForThisServerType.add(replicationPartner);
      panelForThisServerType.add(replicationInterval);
   }
   String toString() {
      return "WINS";
   }
   boolean tryToDisplayServer(Server server) {
      ...
   }
   Server makeServer() {
      ...
   }
}
class DomainControllerUI extends ServerUI {
   ...
}
class ServerDialog extends JDialog {
   Server newServerToReturn;
   JPanel panelForCommonComponents;
   JPanel panelForServerTypeDependentComponents;
   JTextField id;
   JTextField CPUModel;
   JComboBox serverType;
   ServerUI serverUIs[];
   ServerDialog() {
      setupCommonComponents();
      setupServerUIs();
      setupServerTypeCombo();
      setupServerTypeDependentComponents();
   }
   void setupCommonComponents() {
      panelForCommonComponents = new JPanel(...);
      id = new JTextField(...);
      CPUModel = new JTextField(...);
      panelForCommonComponents.add(id);
      panelForCommonComponents.add(CPUModel);
      Container contentPane = getContentPane();
      contentPane.add(panelForCommonComponents, BorderLayout.NORTH);
   }
   void setupServerUIs() {
      ServerUI serverUIs[] = {
         new DNSServerUI(id, CPUModel),
         new WINSServerUI(id, CPUModel),
         new DomainControllerUI(id, CPUModel)
      };
      this.serverUIs = serverUIs;
   }
   void setupServerTypeCombo() {
      serverType = new JComboBox(serverUIs);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      panelForCommonComponents.add(serverType);
   }
   void setupServerTypeDependentComponents() {
      panelForServerTypeDependentComponents = new JPanel(...);
      panelForServerTypeDependentComponents.setLayout(new CardLayout());
      for (int i = 0; i < serverUIs.length; i++) {
         ServerUI UI = serverUIs[i];
          panelForServerTypeDependentComponents.add(
            UI.getPanel(),
            UI.toString());
      }
      Container contentPane = getContentPane();
      contentPane.add(
         panelForServerTypeDependentComponents,
         BorderLayout.CENTER);
   }
   Server editServer(Server server) {
      displayServer(server);
      setVisible(true);
      return newServerToReturn;
   }
   void displayServer(Server server) {
      for (int i = 0; i < serverUIs.length; i++) {
         ServerUI serverUI = serverUIs[i];
         if (serverUI.tryToDisplayServer(server)) {
            serverType.setSelectedItem(serverUI);
         }
      }
   }
   void onOK() {
      newServerToReturn = makeServer();
      dispose();
   }
   Server makeServer() {
      ServerUI serverUI = (ServerUI)serverType.getSelectedItem();
      return serverUI.makeServer();
   }
}
```

3. Point out and remove the code smells in the code below (this example is adpated from the one contributed by Eugen):

```
public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
   // mode value --> ADD, MOD, DEL
   String outResult = "";
   final String connName = "tmp";
   DBConnectionManager connMgr = DBConnectionManager.getInstance();
   Connection conP  = connMgr.getConnection(connName);
   Statement stmt = conP.createStatement();
   int upd = stmt.executeUpdate(sqlStmt);
   switch(upd) {
   case 1:
      if (mode.equalsIgnoreCase("ADD")) { outResult = "SUC Add"; }
      else if (mode.equalsIgnoreCase("MOD")) { outResult = "SUC Mod"; }
      else if (mode.equalsIgnoreCase("DEL")) { outResult = "SUC Del"; }
      break;
   case 0:
      if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
      else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
      break;
   default:
      if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
      else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
      else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
      break;
   }
   stmt.close();
   conP.close();
   return outResult;
}
```

The three branches contain a lot of duplication. They are not much different and their differences can be represented using different data values (no need for different classes).

```
public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
   // mode value --> ADD, MOD, DEL
   String outResult = "";
   final String connName = "tmp";
   DBConnectionManager connMgr = DBConnectionManager.getInstance();
   Connection conP  = connMgr.getConnection(connName);
   Statement stmt = conP.createStatement();
   int upd = stmt.executeUpdate(sqlStmt);
   outResult = (upd==0 ? "SUC" : "Err")+
            " "+
            capitalizeString(mode);
   stmt.close();
   conP.close();
   return outResult;
}
public static String capitalizeString(String string) {
   ...
}
```

4. Point out and remove the code smells in the code below (this example is adapted from the one contributed by Carol):

```
class Account {
   final public int SAVING=0;
   final public int CHEQUE=1;
   final public int FIXED=2;  //Portuguese currency
   private int accountType;
   private double balance;
   public double getInterestRate(...) { // Some method;
      ...
   }
   public Account(int accountType) {
      this.accountType=accountType;
   }
   public double calcInterest() {
      switch (accountType) {
         case SAVING:
            return balance*getInterestRate();
         case CHEQUE:
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            return 0;
         case FIXED:
            return balance*(getInterestRate()+0.02);
      }
   }
}
```

Remove the type code and switch using different classes. There is a wrong comment ("Portuguese currency") that should be removed.

```
abstract class Account {
   private double balance;
   abstract public double calcInterest();
   public double getInterestRate(...) { // Some method;
      ...
   }
}
class SavingAccount extends Account {
   public double calcInterest() {
      return getBalance()*getInterestRate();
   }
}
class ChequeAccount extends Account {
   public double calcInterest() {
      return 0;
   }
}
class FixedAccount extends Account {
   public double calcInterest() {
      return getBalance()*(getInterestRate()+0.02);
   }
}
```

5. Point out and remove the code smells in the code below (this example is contributed by Antonio):

```
class Department{
   final public int Account =0;
   final public int Marketing = 1;
   final public int CustomerServices = 2;
   protected int departmentCode;
   public Department(int departmentCode){
      this.departmentCode = departmentCode;
   }
   public String getDepartmentName(){
      switch (departmentCode){
         case Account:
            return "Account";
         case Marketing:
            return "Marketing";
         case CustomerServices:
            return "Customer Services";
      }
   }
}
```

Remove the type code and the switch. The differences between the types can be

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

represented using different data values (no need for different classes).

```
class Department {
   final public Department Account = new Department("Account");
   final public Department Marketing = new Department("Marketing");
   final public Department CustomerServices =
      new Department("Customer Services ");
   private String departmentName;
   private Department(String departmentName){
      this.departmentName = departmentName;
   }
   public String getDepartmentName(){
      return departmentName;
   }
}
```

6. Point out and remove the code smells in the code below (this example is contributed by Carita):

```
class NormalPayment {
   int units;
   double rate;
   final double TAX_RATE = 0.1;
   double getBillableAmount() {
      double baseAmt = units * rate;
      double tax = baseAmt * TAX_RATE;
      return baseAmt + tax;
   }
}
class PaymentForSeniorCitizen {
   int units;
   double rate;
   final double TAX_RATE = 0.1;
   double getBillableAmount() {
      double baseAmt = units * rate * 0.8;
      double tax = baseAmt * (TAX_RATE - 0.5) ;
      return baseAmt + tax;
   }
}
```

The two classes contain quite some duplicate code. Extract the code to form a parent class:

```
abstract class Payment {
   int units;
   double rate;
   final double TAX_RATE = 0.1;
   abstract double getPreTaxedAmount();
   abstract double getTaxRate();
   double getBillableAmount() {
      return getPreTaxedAmount() * (1 + getTaxRate());
   }
   double getNormalAmount() {
      return units * rate;
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class NormalPayment extends Payment {
   double getPreTaxedAmount() {
      return getNormalAmount();
   }
   double getTaxRate() {
      return TAX_RATE;
   }
}
class PaymentForSeniorCitizen extends Payment {
   double getPreTaxedAmount() {
      return getNormalAmount()*0.8;
   }
   double getTaxRate() {
      return TAX_RATE - 0.5;
   }
}
```

7. Point out and remove the code smells in the code below (this example is contributed by Malaquias):

```
class PianoKey {
   final static int key0 = 0;
   final static int key1 = 1;
   final static int key2 = 2;
   int keyNumber;
   public void playSound() {
      if (keyNumber == 0) {
         //play the frequency for key0
      }
      else if (keyNumber == 1) {
         //play the frequency for key1
      }
      else if (keyNumber == 2) {
         //play the frequency for key2
      }
   }
}
class Piano {
   Vector rythmn;
   public void play() {
      for(int i=0;i<rythmn.size();i++) {
         ((PianoKey) rythmn.elementAt(i)).playSound();
      }
   }
}
```

Remove the type code and the long if-then-else-if. The differences between the types can be represented using different data values (no need for different classes).

```
class PianoKey {
   final static PianoKey key0 = new PianoKey(frequency for key0);
   final static PianoKey key1 = new PianoKey(frequency for key1);
   final static PianoKey key2 = new PianoKey(frequency for key2);
   ...
   int frequency;
   private PianoKey(int frequency) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      this.frequency = frequency;
   }
   public void playSound() {
      //play the frequency.
   }
}
class Piano {
   Vector rythmn;
   public void play() {
      for(int i=0;i<rythmn.size();i++) {
         ((PianoKey) rythmn.elementAt(i)).playSound();
      }
   }
}
```

8. Point out and remove the code smells in the code below (this example is contributed by Frankie):

```
class Account {
   final static int LEVEL_USER = 1;
   final static int LEVEL_ADMIN = 2;
   int accountLevel;
   Date expiredDate; // for user account only
   boolean hasLogin; // for admin account only
}
class ERPApp {
   public boolean checkLoginIssue(Account account) {
      if (account.getLevel() == Account.LEVEL_USER) {
         // Check the account expired date
         Date now = new Date();
         if (account.getExpiredDate().before(now))
            return false;
         return true;
      }
      else if (account.getLevel() == Account.LEVEL_ADMIN) {
         // No expired date for admin account
         // Check multilogin
         if (account.hasLogin())
            return false;
         return true;
      }
      return false;
   }
}
```

Remove the type code and switch using different classes. Remove the comments into code ("Check the account expired date", "Check multilogin" and etc.):

```
interface Account {
   boolean canLogin();
}
class UserAccount implements Account {
   Date expiredDate;
   boolean canLogin() {
      return isAccountExpired();
   }
   boolean isAccountExpired() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      Date now = new Date();
      return !getExpiredDate().before(now);
   }
}
class AdminAccount implements Account {
   boolean hasLogin;
   boolean canLogin() {
      return !isTryingMultiLogin();
   }
   boolean isTryingMultiLogin() {
      return hasLogin;
   }
}
class ERPApp {
   public boolean checkLoginIssue(Account account) {
      return account.canLogin();
   }
}
```

9. Point out and remove the code smells in the code below (this example is adapted from the one contributed by YK):

```
class Form1 extends JDialog {
   JComboBox comboBoxReportType;
   Form1() {
      comboBoxReportType = new JComboBox();
      comboBoxReportType.addItem("r1");
      comboBoxReportType.addItem("r2");
      ...
      comboBoxReportType.addItem("r31c");
   }
   void processReport1() {
      //print some fancy report...
   }
   void processReport2() {
      //print another totally different fancy report...
   }
   ...
   void processReport31c() {
      //print yet another totally different fancy report...
   }
   void printReport(String repNo) {
      if (repNo.equals("r1"))
         processReport1();
      else if (repNo.equals("r2"))
         processReport2();
      ...
      else if (repNo.equals("r31c"))
         processReport31c();
   }
   void onPrintClick() {
      printReport((String) comboBoxReportType.getSelectedItem());
   }
}
```

Remove the type code and switch using different classes:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
interface Report {
   void print();
}
class Report1 implements Report {
   String toString() {
      return "r1";
   }
   void print() {
      //print some fancy report...
   }
}
class Report2 implements Report {
   String toString() {
      return "r2";
   }
   void print() {
      //print another totally different fancy report...
   }
}
...
class Report31c implements Report {
   String toString() {
      return "31c";
   }
   void print() {
      //print yet another totally different fancy report...
   }
}
class Form1 extends JDialog {
   JComboBox comboBoxReportType;
   Report reports[] = {
      new Report1(),
      new Report2(),
      ...
      new Report31c()
   };
   Form1() {
      comboBoxReportType = new JComboBox();
      for (int i = 0; i < reports.length; i++) {
         comboBoxReportType.addItem(reports[i]);
      }
   }
   void onPrintClick() {
      Report report = (Report)comboBoxReportType.getSelectedItem();
      report.print();
   }
}
```

10. This application is about restaurants. Initially we created a Restaurant class to represent a restaurant account and includes information such as its name, access password, tel and fax number, address. The class is like:

```
class Restaurant {
   String name;
   String password;
   String telNo;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   String faxNo;
   String address;
}
```

Later, the following requirements are added in sequence:

1. After initial registration, the restaurant account is assigned an activation code by the system. Only after the user enters the activation code, the account will become activated. Until then, the account is inactive and login is not allowed.

2. If the user would like to change the fax number of the account, the new fax number will not take effect immediately (the existing fax number will remain in effect). Instead, the account is assigned an activation code by the system. Only after the user enters the activation code, the new fax number will take effect.

3. A restaurant can be marked as in a certain category (e.g., Chinese restaurant, Portuguese restaurant and etc.). A category is identified by a category ID.

4. The user can input the holidays for the restaurant.

5. The user can input the business hours for the restaurant.

After implementing all these five requirements, the class has grown significantly and become quite complicated as shown below:

```
class Restaurant {
   String name;
   String password;
   String telNo;
   String faxNo;
   String address;
   String verificationCode;
   boolean isActivated;
   String faxNoToBeConfirmed;
   boolean isThereFaxNoToBeConfirmed = false;
   String catId;
   Vector holidays;
   Vector businessSessions;

   void activate(String verificationCode) {
      isActivated =(this.verificationCode.equals(verificationCode));
      if (isActivated && isThereFaxNoToBeConfirmed){
         faxNo = faxNoToBeConfirmed;
         isThereFaxNoToBeConfirmed = false;
      }
   }
   void setFaxNo(String newFaxNo) {
      faxNoToBeConfirmed = newFaxNo;
      isThereFaxNoToBeConfirmed = true;
      isActivated = false;
   }
   boolean isInCategory(String catId) {
      return this.catId.equals(catId);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
   void addHoliday(int year, int month, int day) {
      ...
   }
   void removeHoliday(int year, int month, int day) {
      ...
   }
   boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin) {
      ...
   }
   boolean isInBusinessHour(Calendar time) {
      ...
   }
   Vector getAllHolidays() {
      return holidays;
   }
   Vector getAllBusinessSessions() {
      return businessSessions;
   }
}
```

Your task is to implement the five requirements above in separate classes, leaving the original simple Restaurant class unchanged.

First, restore the Restaurant class to the original simple form:

```
class Restaurant {
   String name;
   String password;
   String telNo;
   String faxNo;
   String address;
}
```

Implement the initial account activation:

```
class RestaurantActivator {
   String verificationCode;
   Restaurant restaurantToAdd;
   boolean tryToActivate(String restName, String verificationCode) {
      if (restName.equals(restaurantToAdd.getName()) &&
         this.verificationCode.equals(verificationCode)) {
         //add restaurantToAdd to the system;
         return true;
      }
      return false;
   }
}
class RestaurantActivators {
   RestaurantActivator activators[];
   void activate(String restName, String verificationCode) {
      for (int i = 0; i < activators.length; i++) {
         if (activators[i].tryToActivate(restName, verificationCode)) {
            //remove activator[i] from activators;
            return;
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      }
   }
}
class RestaurantSystem {
   Restaurant restaurants[];
   RestaurantActivators restaurantActivators;
}
```

Implement the set fax number activation:

```
class FaxNoActivator {
   String verificationCode;
   String newFaxNo;
   Restaurant restaurantToEdit;
   boolean tryToActivate(String restName, String verificationCode) {
      if (restName.equals(restaurantToEdit.getName()) &&
         this.verificationCode.equals(verificationCode)) {
         restaurantToEdit.setFaxNo(newFaxNo);
         return true;
      }
      return false;
   }
}
class FaxNoActivators {
   FaxNoActivator activators[];
   void activate(String restName, String verificationCode) {
      for (int i = 0; i < activators.length; i++) {
         if (activators[i].tryToActivate(restName, verificationCode)) {
            //remove activator[i] from activators;
            return;
         }
      }
   }
}
class RestaurantSystem {
   Restaurant restaurants[];
   RestaurantActivators restaurantActivators;
   FaxNoActivators faxNoActivators;
}
```

However, there is quite some duplicate code between the RestaurantActivator class and the FaxNoActivator class. The RestaurantActivators class is almost identical to the FaxNoActivators class. So, we make RestaurantActivator look identical to FaxNoActivator:

```
class RestaurantActivator {
   String verificationCode;
   Restaurant restaurant;
   boolean tryToActivate(String restName, String verificationCode) {
      if (restName.equals(restaurant.getName()) &&
         this.verificationCode.equals(verificationCode)) {
         //do something to the restaurant;
         return true;
      }
      return false;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
}
class FaxNoActivator {
   String verificationCode;
   String newFaxNo;
   Restaurant restaurant;
   boolean tryToActivate(String restName, String verificationCode) {
      if (restName.equals(restaurant.getName()) &&
         this.verificationCode.equals(verificationCode)) {
         //do something to the restaurant;
         return true;
      }
      return false;
   }
}
```

Now, extract the common code to form a parent class and let the two classes extend it:

```
abstract class RestaurantTaskActivator {
   String verificationCode;
   Restaurant restaurant;
   boolean tryToActivate(String restName, String verificationCode) {
      if (restName.equals(restaurant.getName()) &&
         this.verificationCode.equals(verificationCode)) {
         doSomethingToRestaurant();
         return true;
      }
      return false;
   }
   abstract void doSomethingToRestaurant();
}
class RestaurantActivator extends RestaurantTaskActivator {
   void doSomethingToRestaurant() {
      //add restaurant to the system;
   }
}
class FaxNoActivator extends RestaurantTaskActivator {
   String newFaxNo;
   void doSomethingToRestaurant() {
      restaurant.setFaxNo(newFaxNo);
   }
}
class RestaurantTaskActivators {
   RestaurantTaskActivator activators[];
   void activate(String restName, String verificationCode) {
      for (int i = 0; i < activators.length; i++) {
         if (activators[i].tryToActivate(restName, verificationCode)) {
            //remove activator[i] from activators;
            return;
         }
      }
   }
}
class RestaurantSystem {
   Restaurant restaurants[];
   RestaurantTaskActivators restaurantTaskActivators;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Implement the category ID:

```
class Category {
   String catId;
   String IdOfRestaurantsInThisCat[];
   boolean isInCategory(String restName) {
      ...
   }
}
class RestaurantSystem {
   Restaurant restaurants[];
   RestaurantTaskActivators restaurantTaskActivators;
   Category categories[];
}
```

Implement the holidays:

```
class Holidays {
   Vector holidays;

   void addHoliday(int year, int month, int day) {
      ...
   }
   void removeHoliday(int year, int month, int day) {
      ...
   }
   Vector getAllHolidays() {
      return holidays;
   }
}
class RestaurantSystem {
   Restaurant restaurants[];
   RestaurantTaskActivators restaurantTaskActivators;
   Category categories[];
   HashMap mapRestIdToHolidays;
}
```

Implement the business sessions:

```
class BusinessSessions {
   Vector businessSessions;
   boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin)
{
      ...
   }
   boolean isInBusinessHour(Calendar time) {
      ...
   }
   Vector getAllBusinessSessions() {
      return businessSessions;
   }
}
class RestaurantSystem {
   Restaurant restaurants[];
   RestaurantTaskActivators restaurantTaskActivators;
   Category categories[];
   HashMap mapRestIdToHolidays;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   HashMap mapRestIdToBusinessSessions;
}
```

11. This application is about students. Originally we had a simple Student class as shown in the code:

```
class StudentManagementSystem {
   Student students[];
}
class Student {
   String studentId;
   String name;
   Date dateOfBirth;
}
```

Later, in order to record what courses the student has enrolled in, on which dates he enrolled and how he paid for them, we modified the code as:

```
class StudentManagementSystem {
   Student students[];
}
class Student {
   String studentId;
   String name;
   Date dateOfBirth;
   String courseCodes[]; //the student has enrolled in these courses.
   Date enrollDates[];  //for each enrolled course, the date he enrolled.
   Payment payments[];  //for each enrolled course, how he paid.

   void enroll(String courseCode, Date enrollDate, Payment payment) {
      //add courseCode to courseCodes
      //add enrollDate to enrollDates
      //add payment to Payments
   }
   void unenroll(String courseCode) {
      ...
   }
}
```

Your task is to implement this requirement without modifying the Student class.

First, restore the Student class to the original simple form:

```
class Student {
   String studentId;
   String name;
   Date dateOfBirth;
}
```

Implement the enrollment requirement:

```
class Enrollment {
   String studentId;
   String courseCode;
   Date enrollDate;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   Payment payment;
}
class Enrollments {
   Enrollment enrollments[];
   void enroll(String studentId, String courseCode, Date enrollDate, Payment
payment) {
      Enrollment enrollment = new Enrollment(studentId, courseCode, ...);
      //add enrollment to enrollments;
   }
   void unenroll(String studentId, String courseCode) {
      ...
   }
}
class StudentManagementSystem {
   Student students[];
   Enrollments enrollments;
}
```

12. This application lets the network administrators document their server configurations. Originally we had a simple Server class as shown in the code:

```
class Server {
   String name;
   String CPUModel;
   int RAMSizeInMB;
   int diskSizeInMB;
   InetAddress ipAddress;
}
class ServerConfigSystem {
   Server servers[];
}
```

Now, there are four new requirements. You are required to implement these requirements without modifying the Server class:

1. An administrator (identified by an admin ID) can be assigned to be responsible for a server.

2. We can check if a server is a DHCP server or not. If yes, we can record the address scope it manages (e.g., from 192.168.0.21 to 192.168.0.254).

3. We can check if a server is a file server or not. If yes, we can set and check the disk space quota allocated for each user (identified by a user ID).

4. A server can be a DHCP server and a file server at the same time.

To allow the assignment of an administrator:

```
class Administrator {
   String adminId;
   Server serversAdminedByHim[];
}
class ServerConfigSystem {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   Server servers[];
   Administrator admins[];
}
```

To support DHCP servers:

```
class DHCPConfig {
   InetAddress startIP;
   InetAddress endIP;
}
class ServerConfigSystem {
   Server servers[];
   Administrator admins[];
   HashMap serverToDHCPConfig;
}
```

To support file servers:

```
class FileServerConfig {
   HashMap userIdToQuota;
   int getQuotaForUser(String userId) {
      ...
   }
   void setQuotaForUser(String userId, int quota) {
      ...
   }
}
class ServerConfigSystem {
   Server servers[];
   Administrator admins[];
   HashMap serverToDHCPConfig;
   HashMap serverToFileServerConfig;
}
```

Nothing needs to be done to allow a server to be a DHCP server and a file server at the same time.

13. At the moment the code of a system is shown below:

```
class Customer {
   String id;
   String name;
   String address;
}
class Supplier {
   String id;
   String name;
   String telNo;
   String address;
}
class SalesSystem {
   Vector customers;
   Vector suppliers;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Implement the following new requirements while keeping the code fit:

1. Customers can place orders. Each order has a unique ID. It records the date of the order, the ID of the customer and the name and quantity of each item being ordered.

2. The system can list all the orders placed by a particular customer.

3. The system can list all the orders placed for a particular supplier.

4. A supplier can provide a certain discount to some of the customers it selects (valid for some selected items only). The discount for different customers or different items may be different.

Implement the requirement of placing orders:

```
class Order {
   String orderId;
   String customerId;
   String supplierId;
   Date orderDate;
   OrderLine orderLines;
}
class OrderLine {
   String productName;
   int quantity;
}
class Orders {
   Order orders[];
   void placeOrder(String customerId, String supplierId, ...) {
      ...
   }
}
class SalesSystem {
   Vector customers;
   Vector suppliers;
   Orders orders;
}
```

Implement the requirement of printing orders placed by a customer:

```
class Orders {
   Order orders[];
   void placeOrder(String customerId, String supplierId, ...) {
      ...
   }
   void printOrdersByCustomer(String customerId) {
      ...
   }
}
```

Implement the requirement of printing orders for a supplier:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class Orders {
   Order orders[];
   void placeOrder(String customerId, String supplierId, ...) {
      ...
   }
   void printOrdersByCustomer(String customerId) {
      ...
   }
   void printOrdersForSupplier(String supplierId) {
      ...
   }
}
```

Implement the requirement of providing discounts:

```
class Discount {
   String supplierId;
   String customerId;
   String productName;
   double discountRate;
}
class Discounts {
   Discount discounts[];
   void addDiscount(
         String supplierId,
         String customerId,
         String productName,
         double discountRate) {
      ...
   }
   double findDiscount(
         String supplierId,
         String customerId,
         String productName){
      ...
   }
}
class SalesSystem {
   Vector customers;
   Vector suppliers;
   Orders orders;
   Discounts discounts;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 5

Take Care to Inherit

Example

This is a conference management system. It is used to manage various types of information of conference participants. Each participant corresponds to a record in the Participants table in the database. Sometimes the users delete a participant by mistake. Therefore, when deleting a participant, the system simply sets a delete flag in the record to true without actually deleting it. The system will delete all the records whose delete flag is true within 24 hours. If the users change their mind before 24 hours, the system can bring back the participant by setting the delete flag to false.

Please read the current code carefully:

```
public class DBTable {
   protected Connection conn;
   protected tableName;
   public DBTable(String tableName) {
      this.tableName = tableName;
      this.conn = ...;
   }
   public void clear() {
      PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
      try {
         st.executeUpdate();
      }finally{
         st.close();
      }
   }
   public int getCount() {
      PreparedStatement st = conn.prepareStatement("SELECT COUNT(*) FROM
"+tableName);
      try {
         ResultSet rs = st.executeQuery();
         rs.next();
         return rs.getInt(1);
      }finally{
         st.close();
      }
   }
}
public class ParticipantsInDB extends DBTable {
   public ParticipantsInDB() {
      super("participants");
   }
   public void addParticipant(Participant part) {
      ...
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    public void deleteParticipant(String participantId) {
       setDeleteFlag(participantId, true);
    }
    public void restoreParticipant(String participantId) {
       setDeleteFlag(participantId, false);
    }
    private void setDeleteFlag(String participantId, boolean b) {
       ...
    }
    public void reallyDelete() {
       PreparedStatement st = conn.prepareStatement(
                     "DELETE FROM "+
                     tableName+
                     " WHERE deleteFlag=true");
       try {
          st.executeUpdate();
       }finally{
          st.close();
       }
    }
    public int countParticipants() {
       PreparedStatement st = conn.prepareStatement(
                     "SELECT COUNT(*) FROM "+
                     tableName+
                     " WHERE deleteFlag=false");
       try {
          ResultSet rs = st.executeQuery();
          rs.next();
          return rs.getInt(1);
       }finally{
          st.close();
       }
    }
}
```

Note that countParticipants only counts those records whose deleteFlags are false. That is, it does not count the deleted participants.

The above code looks fine but it contains a severe problem. What is the problem? Read the code below:

```
ParticipantsInDB partsInDB = ...;
Participant kent = new Participant(...);
Participant paul = new Participant(...);
partsInDB.clear();
partsInDB.addParticipant(kent);
partsInDB.addParticipant(paul);
partsInDB.deleteParticipant(kent.getId());
System.out.println("There are "+partsInDB.getCount()+ "participants");
```

The last line should print "There are 1 participants", right? No! It will print "There are 2 participants"! This is because the last line calls the getCount method in DBTable, not the countParticipants method in ParticipantsInDB. Because getCount knows nothing about the delete flag and simply counts records, it doesn't know how to count the effective (not deleted) participants.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Having inherited inappropriate (or useless) features

ParticipantsInDB has inherited the methods of DBTable such as clear and getCount. For ParticipantsInDB, clear should be useful: It deletes all the participants. But getCount makes little sense: For ParticipantsInDB, what getCount does is to count the total number of participants including those effective and those deleted. Generally speaking, nobody should need to know this number. Even if someone does, this method really should not be called getCount, because this name can easily be associated with "counting the number of (effective) participants".

Therefore, ParticipantsInDB should really not inherit this getCount method. What should we do?

Is there really an inheritance relationship?

When we have inherited something that we don't want, we need to think twice: There is really an inheritance relationship between them? Must ParticipantsInDB be a DBTable? Would ParticipantsInDB like the others to know that it is a DBTable?

In fact, ParticipantsInDB represents a set of participants. It can be represented by one, two or three tables in a database. It can also be represented by two tables in two different databases. Therefore, it has no fixed relationship between a database table. Therefore, there is not really an inheritance relationship between them. It means that we should make ParticipantsInDB not to inherit DBTable. As a result, it no longer inherits the getCount method. However, ParticipantsInDB still needs to use the other functions of DBTable, so we let ParticipantsInDB use a DBTable instead:

```
public class DBTable {
   private Connection conn;
   private String tableName;
   public DBTable(String tableName) {
      this.tableName = tableName;
      this.conn = ...;
   }
   public void clear() {
      PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
      try {
         st.executeUpdate();
      }finally{
         st.close();
      }
   }
   public int getCount() {
      PreparedStatement st = conn.prepareStatement("SELECT COUNT(*) FROM
"+tableName);
      try {
         ResultSet rs = st.executeQuery();
         rs.next();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            return rs.getInt(1);
         }finally{
            st.close();
         }
      }
      public String getTableName() {
         return tableName;
      }
      public Connection getConn() {
         return conn;
      }
   }
   public class ParticipantsInDB {
      private DBTable table;
      public ParticipantsInDB() {
         table = new DBTable("participants");
      }
      public void addParticipant(Participant part) {
         ...
      }
      public void deleteParticipant(String participantId) {
         setDeleteFlag(participantId, true);
      }
      public void restoreParticipant(String participantId) {
         setDeleteFlag(participantId, false);
      }
      private void setDeleteFlag(String participantId, boolean b) {
         ...
      }
      public void reallyDelete() {
         PreparedStatement st = table.getConn().prepareStatement(
                         "DELETE FROM "+
                         table.getTableName()+
                         " WHERE deleteFlag=true");
         try {
            st.executeUpdate();
         }finally{
            st.close();
         }
      }
      public void clear() {
         table.clear();
      }
      public int countParticipants() {
         PreparedStatement st = table.getConn().prepareStatement(
                         "SELECT COUNT(*) FROM "+
                         table.getTableName()+
                         " WHERE deleteFlag=false");
         try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
         }finally{
            st.close();
         }
      }
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

ParticipantsInDB no longer inherits DBTable. Instead, it uses a variable to refer to a DBTable object and calls its methods such as clear, getConn, getTableName and etc. In particular, the clear method of ParticipantsInDB consists of only one statement that simply calls the clear method of DBTable. That is, a ParticipantsInDB object passes on its own work to a DBTable object to finish. This kind of passing is called "delegation".

Now, the original code which contained the bug will not compile:

```
ParticipantsInDB partsInDB = ...;
Participant kent = new Participant(...);
Participant paul = new Participant(...);
partsInDB.clear();
partsInDB.addParticipant(kent);
partsInDB.addParticipant(paul);
partsInDB.deleteParticipant(kent.getId());
//Compile error: there is no getCount method in ParticipantsInDB!
System.out.println("There are "+partsInDB.getCount()+ "participants");
```

In summary, we find that there is not really an inheritance relationship between ParticipantsInDB and DBTable. Then we replace delegation for inheritance to solve the problem. The advantage of delegation is that we can selectively "make public" the methods of DBTable (e.g., the clear method). If we use inheritance, we have no choice but to accept all the public methods of DBTable (clear and getCount) as our own public methods.

Take out non-essential features

Now let's see another example. Suppose that a Component represents a GUI object such as a button or text field. Please read the code below carefully:

```
abstract class Component {
   boolean isVisible;
   int posXInContainer;
   int posYInContainer;
   int width;
   int height;
   ...
   abstract void paint(Graphics graphics);
   void setWidth(int newWidth) {
      ...
   }
   void setHeight(int newHeight) {
      ...
   }
}
class Button extends Component {
   ActionListener listeners[];
   ...
   void paint(Graphics graphics) {
      ...
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class Container {
   Component components[];
   void add(Component component) {
      ...
   }
}
```

Suppose that now you would like to write a clock component. It looks like a round clock and will update its hour-hand and minute-hand to display the current time. As it is a component, it should inherit the Component class:

```
class ClockComponent extends Component {
   ...
   void paint(Graphics graphics) {
      //draw the clock showing the time.
   }
}
```

Now we have a problem: It should be round, but it inherits the attributes width and height from Component and the setWidth and setHeight methods. These things are meaningless to a round component.

When we have inherited something that we don't want, we need to think twice: Is there really an inheritance relationship between them? Must ClockComponent be a Component? Would ClockComponent like the others to know that it is a Component?

In contrast to the case of ParticipantsInDB, ClockComponent should indeed be a Component, otherwise it cannot be put into a Container along with some other Components. Therefore, we should indeed use inheritance here (can't use delegation).

As it must inherit Component but doesn't want width, height, setWidth and setHeight, we must take out these four features from Component. In fact, the current situation already proves that these features are not something that all Components must have (at least our ClockComponent doesn't need them). Taking them out is a very reasonable thing to do.

However, if we take out these features from Component, the existing classes like Button will lose the width and height, right? For the Button classes, these features are indeed needed (assuming Buttons must be rectangular).

A possible solution is to create a RectangularComponent to hold these features and let Button inherit RectangularComponent:

```
abstract class Component {
   boolean isVisible;
   int posXInContainer;
   int posYInContainer;
   ...
   abstract void paint(Graphics graphics);
}
abstract class RectangularComponent extends Component {
   int width;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   int height;
   void setWidth(int newWidth) {
      ...
   }
   void setHeight(int newHeight) {
      ...
   }
}
class Button extends RectangularComponent {
   ActionListener listeners[];
   ...
   void paint(Graphics graphics) {
      ...
   }
}
class ClockComponent extends Component {
   ...
   void paint(Graphics graphics) {
      //draw the clock showing the time.
   }
}
```

This is not the only possible solution. Another possible solution is to create a RectangularDimension class to hold features and let Button delegates to it:

```
abstract class Component {
   boolean isVisible;
   int posXInContainer;
   int posYInContainer;
   ...
   abstract void paint(Graphics graphics);
}
class RectangularDimension {
   int width;
   int height;
   void setWidth(int newWidth) {
      ...
   }
   void setHeight(int newHeight) {
      ...
   }
}
class Button extends Component {
   ActionListener listeners[];
   RectangularDimension dim;
   ...
   void paint(Graphics graphics) {
      ...
   }
   void setWidth(int newWidth) {
      dim.setWidth(newWidth);
   }
   void setHeight(int newHeight) {
      dim.setHeight(newHeight);
   }
}
class ClockComponent extends Component {
   ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    void paint(Graphics graphics) {
       //draw the clock showing the time.
    }
}
```

Summary

When we would like to let one class inherit another, we need to check carefully: Will the sub-class inherit some features (attributes or methods) that it doesn't want? If yes, we need to think carefully: Is there really an inheritance relationship between them? If no, use delegation. If yes, take out those unwanted features and put them into an appropriate location.

References

- The Liskov Substitution Principle states: We should be able to use an object of a sub-class to replace an object of a base class, without causing any difference to the client code using the base class. Although the main point of this chapter is different from that of the Liskov Substitution Principle, it is still a very good reference:
    - http://www.objectmentor.com/resources/articles/lsp.pdf.
    - http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple.
- Design By Contract is something related to the Liskov Substitution Principle. It states that we should test our assumption about our code. For the details, see:
    - http://c2.com/cgi/wiki?DesignByContract.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. The following code contains duplication. If you need to remove the duplication at any cost, how do you do that?

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
   GridListener listeners[];
   void appendRow() {
      //append a row at the end of the grid.
      for (int i = 0; i < listeners.length; i++) {
         listeners[i].onRowAppended();
      }
   }
   void moveRow(int existingIdx, int newIdx) {
      //move the row.
      for (int i = 0; i < listeners.length; i++) {
         listeners[i].onRowMoved(existingIdx, newIdx);
      }
   }
}
```

2. Java provides a class called "HashMap". You can add a key-value pair to such a map by the "put" method. Later, you can retrieve the value by calling the "get" method and providing the key. The key and value can be any type. It has a "size" method that returns the number of pairs in the map. See the sample code below.

```
HashMap m=new HashMap();
m.put("x", new Integer(123));
m.put("y", "hello");
m.put(new Double(120.33), "hi");
System.out.println(m.get("x"));  // print 123
System.out.println(m.get("y"));  // print hello
System.out.println(m.size());    // print 3
```

   Point out and remove the problem in the code below:

```
public class CourseCatalog extends HashMap {
   public void addCourse(Course c) {
      put(c.getTitle(), c);
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public Course findCourse(String title) {
      return (Course)get(title);
   }
   public int countCourses() {
      return size();
   }
}
```

3. Point out and remove the problem in the code below. You are NOT allowed to use the collection classes in Java.

```
public class Node {
   private Node nextNode;
   public Node getNextNode() {
      return nextNode;
   }
   public void setNextNode(Node nextNode) {
      this.nextNode = nextNode;
   }
}
public class LinkList {
   private Node firstNode;
   public void addNode(Node newNode) {
      ...
   }
   public Node getFirstNode() {
      return firstNode;
   }
}
public class Employee extends Node {
   String employeeId;
   String name;
   ...
}
public class EmployeeList extends LinkList {
   public void addEmployee(Employee employee) {
      addNode(employee);
   }
   public Employee getFirstEmployee() {
      return (Employee)getFirstNode();
   }
   ...
}
```

4. Suppose that in general a teacher can teach many students. However, a graduate student can be taught by a graduate teacher only. Point out and remove the problem in the code:

```
class Student {
   String studentId;
   ...
}
class Teacher {
   String teacherId;
   Vector studentsTaught;
   public String getId() {
      return teacherId;
   }
   public void addStudent(Student student) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      studentsTaught.add(student);
   }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
}
```

5. Point out and remove the problem in the code:

```
public class Button {
   private Font labelFont;
   private String labelText;
   ...
   public void addActionListener(ActionListener listener) {
      ...
   }
   public void paint(Graphics graphics) {
      //draw the label text on the graphics using the label's font.
   }
}
public class BitmapButton extends Button {
   private Bitmap bitmap;
   public void paint(Graphics graphics) {
      //draw the bitmap on the graphics.
   }
}
```

6. A property file is a regular text file, but its content has a particular format. For example, it may look like:

```
java.security.enforcePolicy=true
java.system.lang=en
conference.abc=10
xyz=hello
```

That is, every line in a property file has a key (a string), then an equal sign and finally a value.

In order to make it easier to create this kind of property file, you have written the following code, using the FileWriter and its write method in Java:

```
class PropertyFileWriter extends FileWriter {
   PropertyFileWriter(File file) {
      super(file);
   }
   void writeEntry(String key, String value) {
      super.write(key+"="+value);
   }
}
class App {
   void makePropertyFile() {
      PropertyFileWriter fw = new PropertyFileWriter("f1.properties");
      try {
         fw.writeEntry("conference.abc", "10");
         fw.writeEntry("xyz", "hello");
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      } finally {
         fw.close();
      }
   }
}
```

Point out and remove the problem in the code.

7. Come up with some code that misuses inheritance and correct the problem.

Hints

1. Refer to the RentalProcessor example.

2. You don't want to inherit methods like put.

3. You don't want to inherit methods like getNextNode and addNode. Stop Employee from inheriting Node. Create a new class like EmployeeNode. Let EmployeeList contains a LinkedList instead. When adding an Employee, it should create an EmployeeNode object.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. The following code contains duplication. If you need to remove the duplication at any cost, how do you do that?

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
   GridListener listeners[];
   void appendRow() {
      //append a row at the end of the grid.
      for (int i = 0; i < listeners.length; i++) {
         listeners[i].onRowAppended();
      }
   }
   void moveRow(int existingIdx, int newIdx) {
      //move the row.
      for (int i = 0; i < listeners.length; i++) {
         listeners[i].onRowMoved(existingIdx, newIdx);
      }
   }
}
```

First, make the code look identical:

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
   GridListener listeners[];
   void appendRow() {
      //append a row at the end of the grid.
      for (int i = 0; i < listeners.length; i++) {
         notify listeners[i];
      }
   }
   void moveRow(int existingIdx, int newIdx) {
      //move the row.
      for (int i = 0; i < listeners.length; i++) {
         notify listeners[i];
      }
   }
}
```

Then extract the duplicate code into a method:

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
   ...
   void appendRow() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      //append a row at the end of the grid.
      ???();
   }
   void moveRow(int existingIdx, int newIdx) {
      //move the row.
      ???();
   }
   void ???() {
      for (int i = 0; i < listeners.length; i++) {
         notify listeners[i];
      }
   }
}
```

To find a good name for the method, see what the code of the method actually does. In this case, It notifies all the listeners one by one. So "notifyListeners" should be a good name:

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
   ...
   void appendRow() {
      //append a row at the end of the grid.
      notifyListeners();
   }
   void moveRow(int existingIdx, int newIdx) {
      //move the row.
      notifyListeners();
   }
   void notifyListeners() {
      for (int i = 0; i < listeners.length; i++) {
         notify listeners[i];
      }
   }
}
```

As there are two ways to implement "notify listeners[i]", we need to create an interface so that later we can create one implementation class for each implementation:

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
interface ??? {
   void notify(GridListener listener);
}
class Grid {
   ...
   void appendRow() {
      //append a row at the end of the grid.
      notifyListeners();
   }
   void moveRow(int existingIdx, int newIdx) {
      //move the row.
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      notifyListeners();
   }
   void notifyListeners(??? someObject) {
      for (int i = 0; i < listeners.length; i++) {
         someObject.notify(listeners[i]);
      }
   }
}
```

To find a good name for this interface, see what this interface does. As it has only one method (notify) which notifies the specified GridListener, "GridListenerNotifier" should be a good name:

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
interface GridListenerNotifier {
   void notify(GridListener listener);
}
class Grid {
   ...
   void appendRow() {
      //append a row at the end of the grid.
      notifyListeners();
   }
   void moveRow(int existingIdx, int newIdx) {
      //move the row.
      notifyListeners();
   }
   void notifyListeners(GridListenerNotifier notifier) {
      for (int i = 0; i < listeners.length; i++) {
         notifier.notify(listeners[i]);
      }
   }
}
```

Provide a different implementation class for each implementation. Make them inner classes if possible:

```
interface GridListener {
   void onRowAppended();
   void onRowMoved(int existingIdx, int newIdx);
}
interface GridListenerNotifier {
   void notify(GridListener listener);
}
class Grid {
   ...
   void appendRow() {
      //append a row at the end of the grid.
      notifyListeners(new GridListenerNotifier() {
         void notify(GridListener listener) {
            listener.onRowAppended();
         }
      });
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void moveRow(final int existingIdx, final int newIdx) {
      //move the row.
      notifyListeners(new GridListenerNotifier() {
         void notify(GridListener listener) {
            listener.onRowMoved(existingIdx, newIdx);
         }
      });
   }
   void notifyListeners(GridListenerNotifier notifier) {
      for (int i = 0; i < listeners.length; i++) {
         notifier.notify(listeners[i]);
      }
   }
}
```

2. Java provides a class called "HashMap". You can add a key-value pair to such a map by the "put" method. Later, you can retrieve the value by calling the "get" method and providing the key. The key and value can be any type. It has a "size" method that returns the number of pairs in the map. See the sample code below.

```
HashMap m=new HashMap();
m.put("x", new Integer(123));
m.put("y", "hello");
m.put(new Double(120.33), "hi");
System.out.println(m.get("x"));  // print 123
System.out.println(m.get("y"));  // print hello
System.out.println(m.size());    // print 3
```

   Point out and remove the problem in the code below:

```
public class CourseCatalog extends HashMap {
   public void addCourse(Course c) {
      put(c.getTitle(), c);
   }
   public Course findCourse(String title) {
      return (Course)get(title);
   }
   public int countCourses() {
      return size();
   }
}
```

   CourseCatalog does not want to inherit the put, get and size methods. Yes, it needs to use these methods, but it does not want others to call these methods on it. For example, the following code will cause trouble:

```
CourseCatalog courseCatalog = new CourseCatalog();
courseCatalog.put("Hello", "World");
courseCatalog.findCourse("Hello"); //trouble: "World" is a string, not Course
```

   As there is no code that needs to use a CourseCatalog as a HashMap, we should use delegation instead of inheritance:

```
public class CourseCatalog {
   HashMap map;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public void addCourse(Course c) {
      map.put(c.getTitle(), c);
   }
   public Course findCourse(String title) {
      return (Course)map.get(title);
   }
   public int countCourses() {
      return map.size();
   }
}
```

3. Point out and remove the problem in the code below. You are NOT allowed to use the collection classes in Java.

```
public class Node {
   private Node nextNode;
   public Node getNextNode() {
      return nextNode;
   }
   public void setNextNode(Node nextNode) {
      this.nextNode = nextNode;
   }
}
public class LinkList {
   private Node firstNode;
   public void addNode(Node newNode) {
      ...
   }
   public Node getFirstNode() {
      return firstNode;
   }
}
public class Employee extends Node {
   String employeeId;
   String name;
   ...
}
public class EmployeeList extends LinkList {
   public void addEmployee(Employee employee) {
      addNode(employee);
   }
   public Employee getFirstEmployee() {
      return (Employee)getFirstNode();
   }
   ...
}
```

EmployeeList does not want to inherit the addNode method. Yes, it needs to use this method, but it does not want others to call this method on it. For example, the following code will cause trouble:

```
EmployeeList employeeList = new EmployeeList();
Node someNode = new Node();
employeeList.addNode(someNode);
employeeList.getFirstEmployee();//trouble: someNode is a Node, not Employee
```

Employee does not want to inherit getNextNode either.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

As there is no code that needs to use an EmployeeList as a List, we should use delegation instead of inheritance:

```
public class Node {
   ...
}
public class LinkList {
   ...
}
public class Employee {
   String employeeId;
   String name;
   ...
}
public class EmployeeNode extends Node {
   Employee employee;
}
public class EmployeeList {
   LinkList list;
   public void addEmployee(Employee employee) {
      list.addNode(new EmployeeNode(employee));
   }
   public Employee getFirstEmployee() {
      return ((EmployeeNode)list.getFirstNode()).getEmployee();
   }
   ...
}
```

4. Suppose that in general a teacher can teach many students. However, a graduate student can be taught by a graduate teacher only. Point out and remove the problem in the code:

```
class Student {
   String studentId;
   ...
}
class Teacher {
   String teacherId;
   Vector studentsTaught;
   public String getId() {
      return teacherId;
   }
   public void addStudent(Student student) {
      studentsTaught.add(student);
   }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
}
```

Currently, it is possible to let a non-graduate teacher teach a graduate student. The problem is caused by the addStudent method in the Teacher class. This method accepts any Student. Is it true that a Teacher can teach any Student? It is not true. Therefore, this method should not exist here. Instead, we should move it GraduateTeacher:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class Student {
   String studentId;
   ...
}
class Teacher {
   String teacherId;
   Vector studentsTaught;
   public String getId() {
      return teacherId;
   }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
   public void addStudent(Student student) {
      studentsTaught.add(student);
   }
}
```

However, as the AddStudent is no longer in the Teacher class, how can the other (non-graduate) teachers teach the non-graduate students? We can create a NonGraduateTeacher class and a NonGraduateStudent class for that:

```
class Student {
   String studentId;
   ...
}
class Teacher {
   String teacherId;
   Vector studentsTaught;
   public String getId() {
      return teacherId;
   }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
   public void addStudent(Student student) {
      studentsTaught.add(student);
   }
}
class NonGraduateStudent extends Student {
}
class NonGraduateTeacher extends Teacher {
   public void addStudent(NonGraduateStudent student) {
      studentsTaught.add(student);
   }
}
```

5. Point out and remove the problem in the code:

```
public class Button {
   private Font labelFont;
   private String labelText;
   ...
   public void addActionListener(ActionListener listener) {
      ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
   public void paint(Graphics graphics) {
      //draw the label text on the graphics using the label's font.
   }
}
public class BitmapButton extends Button {
   private Bitmap bitmap;
   public void paint(Graphics graphics) {
      //draw the bitmap on the graphics.
   }
}
```

The labelFont and labelText in the Button class make no sense in the BitmapButton class. As every button needs a font or some text, they should not be in the Button class. Extract them into a new sub-class such as LabelButton. The paint method in the Button class thus should become abstract because there is no more text to draw:

```
public abstract class Button {
   ...
   public void addActionListener(ActionListener listener) {
      ...
   }
   abstract public void paint(Graphics graphics);
}
public class LabelButton extends Button {
   private Font labelFont;
   private String labelText;
   public void paint(Graphics graphics) {
      //draw the label text on the graphics using the label's font.
   }
}
public class BitmapButton extends Button {
   private Bitmap bitmap;
   public void paint(Graphics graphics) {
      //draw the bitmap on the graphics.
   }
}
```

6. A property file is a regular text file, but its content has a particular format. For example, it may look like:

```
java.security.enforcePolicy=true
java.system.lang=en
conference.abc=10
xyz=hello
```

That is, every line in a property file has a key (a string), then an equal sign and finally a value.

In order to make it easier to create this kind of property file, you have written the following code, using the FileWriter and its write method in Java:

```
class PropertyFileWriter extends FileWriter {
   PropertyFileWriter(String path) {
      super(new File(path));
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
   void writeEntry(String key, String value) {
      super.write(key+"="+value);
   }
}
class App {
   void makePropertyFile() {
      PropertyFileWriter fw = new PropertyFileWriter("f1.properties");
      try {
         fw.writeEntry("conference.abc", "10");
         fw.writeEntry("xyz", "hello");
      } finally {
         fw.close();
      }
   }
}
```

Point out and remove the problem in the code.

PropertyFileWrite does not want to inherit the write method. Yes, it needs to use this method, but it does not want others to call this method on it. For example, the following code will cause trouble:

```
PropertyFileWriter propertyFileWriter = new PropertyFileWriter(...);
propertyFileWriter.write("this is not a valid entry line");
```

As there is no code that needs to use a PropertyFileWriter as a FileWriter, we should use delegation instead of inheritance:

```
class PropertyFileWriter {
   FileWriter fileWriter;
   PropertyFileWriter(String path) {
      fileWriter = new FileWriter(new File(path));
   }
   void writeEntry(String key, String value) {
      fileWriter.write(key+"="+value);
   }
   void close() {
      fileWriter.close();
   }
}
class App {
   void makePropertyFile() {
      PropertyFileWriter fw = new PropertyFileWriter("f1.properties");
      try {
         fw.writeEntry("conference.abc", "10");
         fw.writeEntry("xyz", "hello");
      } finally {
         fw.close();
      }
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 6

Handling Inappropriate References

Example

This is a small program handling zip files. The user can input the path to the zip file such as c:\f.zip and the paths to the files that he would like to add to the zip file such as c:\f1.doc, c:\f2.doc and etc. in the main frame, then the program will compress f1.doc and f2.doc and create f.zip. When it is compressing each file, the status bar in the main frame will display the related message. For example, when it is compressing c:\f2.doc, the status bar will display "zipping c:\f2.zip".

The current code is:

```
class ZipMainFrame extends Frame {
   StatusBar sb;
   void makeZip() {
      String zipFilePath;
      String srcFilePaths[];
      //setup zipFilePath and srcFilePaths according to the UI.
      ...
      ZipEngine ze = new ZipEngine();
      ze.makeZip(zipFilePath, srcFilePaths, this);
   }
   void setStatusBarText(String statusText) {
      sb.setText(statusText);
   }
}
class ZipEngine {
   void makeZip(String zipFilePath, String srcFilePaths[], ZipMainFrame f) {
      //create zip file at the path.
      ...
      for (int i = 0; i < srcFilePaths.length; i++) {
         //add the file srcFilePaths[i] into the zip file.
         ...
         f.setStatusBarText("Zipping "+srcFilePaths[i]);
      }
   }
}
```

Now, suppose that we would like to reuse the ZipEngine class in another system (e.g., an inventory system) to perform backups. The files it needs to compress (backup) are the internal data files used by the system. Therefore, the paths are hard-coded and do not need to be input by the user. Suppose that it has a main frame too, and it also needs to display in its

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

status bar the path to each file as it is compressed.

However, there is a problem. We hope to reuse ZipEngine without rewriting the code. However, please read the code of ZipEngine above: It needs a ZipMainFrame to operate. Obviously, any other systems will not have a ZipMainFrame class. Therefore, this ZipEngine class can be used only in this small zip file handling program, but not in all the other systems (including this inventory system).

"Inappropriate" references make code hard to reuse

Because ZipEngine references ZipMainFrame, when we try to reuse ZipEngine, it will drag in ZipMainFrame. Because in the new environment it is impossible to have ZipMainFrame, ZipEngine cannot be reused.

Generally speaking, if a class A references another class B, when we would like to reuse A, A will drag in B. If B references another class C, B will in turn drag in C. When B or C has no meaning in the new environment or cannot exist in the new environment, we cannot reuse A. Therefore, "Inappropriate" references make the code hard to reuse.

In order to reuse ZipEngine, we must first make ZipEngine no longer reference ZipMainFrame. How to do that? Before answering this question, we need to answer another question first: Given any code fragment, how to check if the code contains "inappropriate" references? What is "inappropriate"?

How to check for "inappropriate" references

Method 1

If there are "inappropriate" references in the code, a simplest method to check is: We check if the code contains mutual references or circular references. For example, ZipMainFrame references ZipEngine, and ZipEngine references ZipMainFrame. Circular references is a code smell, suggesting that the references are "inappropriate".

This method is very simple. However, sometimes even there is an "inappropriate" reference, there are still no circular references. Therefore, this method may miss.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Method 2

Another method is subjective: We check the code and ask ourselves: Does it really need what it references? For ZipEngine, does it really need ZipMainFrame? The code of ZipEngine shows that it needs to display a message in the status bar of ZipMainFrame. Does it absolutely need a ZipMainFrame and cannot use anything else? Is there anything that can replace the ZipMainFrame? In fact, it doesn't really absolutely need a ZipMainFrame. All it needs is just a status bar to let it display the message. Therefore, we can change the code to:

```
class ZipEngine {
   void makeZip(String zipFilePath, String srcFilePaths[], StatusBar statusBar) {
      //create zip file at the path.
      ...
      for (int i = 0; i < srcFilePaths.length; i++) {
         //add the file srcFilePaths[i] into the zip file.
         ...
         statusbar.setText("Zipping "+srcFilePaths[i]);
      }
   }
}
```

Now, ZipEngine only references a StatusBar, not a ZipMainFrame. Because StatusBar is much more general (can exist in many more environments) than ZipMainFrame, the reusability of ZipEngine has been increased significantly. However, this method is too subjective. There is no fixed criterion, therefore it is very hard to apply. For example, we may say that ZipEngine doesn't really need a status, all it needs is a service to display messages (not necessarily a status bar).

Method 3

The third method is also subjective: We check the code and ask ourselves: Can it be reused in another environment? If no, we consider it contains "inappropriate" references. For example, we ask: Can ZipEngine be reused in another environment? Because it references ZipMainFrame, it obviously cannot. Therefore, it really should not reference ZipMainFrame. This method is also subjective and is very hard to apply. For example: What is the new environment like? It is totally subject to our imagination without limits.

Method 4

The fourth method is a simpler and objective method. When we find that we need to reuse the code, but cannot do it because it references something, then we consider it contains "inappropriate" references. For example, when we really need to reuse ZipEngine in an inventory system, we find that it cannot be reused. So we consider it contains "inappropriate" references (to ZipMainFrame).

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

This is a "lazy", "passive" method, but it is effective.

Summary on the checking methods

To check if the code contains "inappropriate" references, there are four methods:

1. Look: Are there circular references?
2. Think: What it really needs?
3. Think: Can it be reused in a new environment?
4. Look: Now, I need to reuse it in this environment, can it be done?

Method 1 and 4 are the easiest to use and therefore recommended for beginners. With more design experience, method 2 and 3 will be easier.

How to make ZipEngine no longer reference ZipMainFrame

Now we will see how to make ZipEngine no longer reference ZipMainFrame. In fact, when we introduced method 2, through contemplation we have found the real need of ZipEngine and found a solution. Because method 2 is relatively hard to apply, we do not use it here. Assume that we use method 4 and find that ZipEngine cannot be reused in a text mode system (there is no status bar, but we would like to display the messages on the screen). How to solve this problem?

Because we cannot reuse ZipEngine, we first use Copy and Paste, and then modify the resulting code:

```
class TextModeApp {
   void makeZip() {
      String zipFilePath;
      String srcFilePaths[];
      ...
      ZipEngine ze = new ZipEngine();
      ze.makeZip(zipFilePath, srcFilePaths);
   }
}
class ZipEngine {
   void makeZip(String zipFilePath, String srcFilePaths[]) {
      //create zip file at the path.
      ...
      for (int i = 0; i < srcFilePaths.length; i++) {
         //add the file srcFilePaths[i] into the zip file.
         ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            System.out.println("Zipping "+srcFilePaths[i]);
         }
      }
   }
```

Compare this code to the original ZipEngine:

```
class ZipEngine {
   void makeZip(String zipFilePath, String srcFilePaths[], ZipMainFrame f) {
      //create zip file at the path.
      ...
      for (int i = 0; i < srcFilePaths.length; i++) {
         //add the file srcFilePaths[i] into the zip file.
         ...
         f.setStatusBarText("Zipping "+srcFilePaths[i]);
      }
   }
}
```

Obviously there is a lot of duplicate code (code smell). To remove the code smell, we need to make these two code fragments identical (on a certain abstraction level). For example, we may change them to:

```
class ZipEngine {
   void makeZip(String zipFilePath, String srcFilePaths[]) {
      //create zip file at the path.
      ...
      for (int i = 0; i < srcFilePaths.length; i++) {
         //add the file srcFilePaths[i] into the zip file.
         ...
         show the message;
      }
   }
}
```

Because this "show the message" method need two different implementations, we need to create an interface declaring this method and then create two implementation classes to implement this interface, each of which will provide one method implementation. Because this method is called "show the message", this interface may be called "MessageDisplay" or "MessageSink" and etc.:

```
interface MessageDisplay {
   void showMessage(String msg);
}
```

ZipEngine is changed to:

```
class ZipEngine {
   void makeZip(String zipFilePath, String srcFilePaths[], MessageDisplay
msgDisplay) {
      //create zip file at the path.
      ...
      for (int i = 0; i < srcFilePaths.length; i++) {
         //add the file srcFilePaths[i] into the zip file.
         ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            msgDisplay.showMessage("Zipping "+srcFilePaths[i]);
        }
    }
}
```

The two implementation classes implementing MessageDisplay may be:

```
class ZipMainFrameMessageDisplay implements MessageDisplay {
   ZipMainFrame f;
   ZipMainFrameMessageDisplay(ZipMainFrame f) {
      this.f = f;
   }
   void showMessage(String msg) {
      f.setStatusBarText(msg);
   }
}
class SystemOutMessageDisplay implements MessageDisplay {
   void showMessage(String msg) {
      System.out.println(msg);
   }
}
```

The two systems need to change accordingly:

```
class ZipMainFrame extends Frame {
   StatusBar sb;
   void makeZip() {
      String zipFilePath;
      String srcFilePaths[];
      //setup zipFilePath and srcFilePaths according to the UI.
      ...
      ZipEngine ze = new ZipEngine();
      ze.makeZip(zipFilePath, srcFilePaths, new ZipMainFrameMessageDisplay(this));
   }
   void setStatusBarText(String statusText) {
      sb.setText(statusText);
   }
}
class TextModeApp {
   void makeZip() {
      String zipFilePath;
      String srcFilePaths[];
      ...
      ZipEngine ze = new ZipEngine();
      ze.makeZip(zipFilePath, srcFilePaths, new SystemOutMessageDisplay());
   }
}
```

The improved code

The improved code is shown below. To make the code clearer, we use inner classes in Java:

```
interface MessageDisplay {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void showMessage(String msg);
}
class ZipEngine {
   void makeZip(String zipFilePath, String srcFilePaths[], MessageDisplay
msgDisplay) {
      //create zip file at the path.
      ...
      for (int i = 0; i < srcFilePaths.length; i++) {
         //add the file srcFilePaths[i] into the zip file.
         ...
         msgDisplay.showMessage("Zipping "+srcFilePaths[i]);
      }
   }
}
class ZipMainFrame extends Frame {
   StatusBar sb;
   void makeZip() {
      String zipFilePath;
      String srcFilePaths[];
      //setup zipFilePath and srcFilePaths according to the UI.
      ...
      ZipEngine ze = new ZipEngine();
      ze.makeZip(zipFilePath, srcFilePaths, new MessageDisplay() {
         void showMessage(String msg) {
            setStatusBarText(msg);
         }
      });
   }
   void setStatusBarText(String statusText) {
      sb.setText(statusText);
   }
}
class TextModeApp {
   void makeZip() {
      String zipFilePath;
      String srcFilePaths[];
      ...
      ZipEngine ze = new ZipEngine();
      ze.makeZip(zipFilePath, srcFilePaths, new MessageDisplay() {
         void showMessage(String msg) {
            System.out.println(msg);
         }
      });
   }
}
```

References

- The Dependency Inversion Principle states: A high level module shouldn't depend on a low level module. If it really happens, we should extract an abstract concept and let them both depend on that concept. For more details, please see:
  - http://www.objectmentor.com/resources/articles/dip.pdf.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- http://c2.com/cgi/wiki?DependencyInversionPrinciple.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. Point out the problem in the code below. Further suppose that you need to reuse the file copier in a text mode file copying application that will display the progress in its text console as an integer. What should you do?

```
class MainApp extends JFrame {
   ProgressBar progressBar;
   void main() {
      FileCopier fileCopier = new FileCopier(this);
      fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));        }
   void updateProgressBar(int noBytesCopied, int sizeOfSource) {
      progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
   }
}
class FileCopier {
   MainApp app;
   FileCopier(MainApp app) {
      this.app = app;
   }
   void copyFile(File source, File target) {
      int sizeOfSource = (int)source.length();
      for (int i = 0; i < sizeOfSource; ) {
         //read n (<= 512) bytes from source.
         //write n bytes to target.
         i += n;
         app.updateProgressBar(i, sizeOfSource);
      }
   }
}
```

2. In the above case, suppose that you need to develop another text mode file copying application that will display a "*" for each 10% of progress in its text console. What should you do?

3. Point out the problem in the code below. Further suppose that you need to reuse the fax machine code in another application. What should you do?

```
class MainApp {
   String faxNo;
   void main() {
      FaxMachine faxMachine = new FaxMachine(this);
      faxMachine.sendFax("783675", "hello");
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class FaxMachine {
   MainApp app;
   FaxMachine(MainApp app) {
      this.app = app;
   }
   void sendFax(String toFaxNo, String msg) {
      FaxMachineHardware hardware = ...;
      hardware.setStationId(app.getFaxNo());
      hardware.setRecipientFaxNo(toFaxNo);
      hardware.start();
      try {
         do {
            Graphics graphics = hardware.newPage();
            //draw the msg into the graphics.
         } while (more page is needed);
      } finally {
         hardware.done();
      }
   }
}
```

4. Point out the problem in the code below. Further suppose that you need to reuse the heat sensor code in another application. What should you do?

```
class Cooker {
   HeatSensor heatSensor;
   Speaker speaker;
   void alarm() {
      speaker.setFrequency(Speaker.HIGH_FREQUENCY);
      speaker.turnOn();
   }
}
class HeatSensor {
   Cooker cooker;
   HeatSensor(Cooker cooker) {
      this.cooker = cooker;
   }
   void checkOverHeated() {
      if (isOverHeated()) {
         cooker.alarm();
      }
   }
   boolean isOverHeated() {
      ...
   }
}
```

5. This is a word processor application. It can let the user select the font. Before the user confirms to change the font, it will let the user preview the effect of the change. The current code is shown below. Point out the code smell. If we need to reuse the ChooseFontDialog in a GUI application that doesn't support this preview functionality, how should you change the code?

```
class WordProcessorMainFrame extends JFrame {
   void onChangeFont() {
      ChooseFontDialog chooseFontDialog = new ChooseFontDialog(this);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      if (chooseFontDialog.showOnScreen()) {
         Font newFont = chooseFontDialog.getSelectedFont();
         //show the contents using this new font.
      } else {
         //show the contents using the existing font.
      }
   }
   void previewWithFont(Font font) {
      //show the contents using this preview font.
   }
}
class ChooseFontDialog extends JDialog {
   WordProcessorMainFrame mainFrame;
   Font selectedFont;
   ChooseFontDialog(WordProcessorMainFrame mainFrame) {
      this.mainFrame = mainFrame;
   }
   boolean showOnScreen() {
      ...
   }
   void onSelectedFontChange() {
      selectedFont = getSelectedFontFromUI();
      mainFrame.previewWithFont(selectedFont);
   }
   Font getSelectedFontFromUI() {
      ...
   }
   Font getSelectedFont() {
      return selectedFont;
   }
}
```

6. Come up with some code that contains inappropriate references and correct the problem.

Hints

1. Circular references.

2. No hint for this one.

3. Circular references. No need to create an interface and implementation classes. The differences should be represented using different objects, not classes.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Point out the problem in the code below. Further suppose that you need to reuse the file copier in a text mode file copying application that will display the progress in its text console as an integer. What should you do?

```
class MainApp extends JFrame {
   ProgressBar progressBar;
   void main() {
      FileCopier fileCopier = new FileCopier(this);
      fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
   }
   void updateProgressBar(int noBytesCopied, int sizeOfSource) {
      progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
   }
}
class FileCopier {
   MainApp app;
   FileCopier(MainApp app) {
      this.app = app;
   }
   void copyFile(File source, File target) {
      int sizeOfSource = (int)source.length();
      for (int i = 0; i < sizeOfSource; ) {
         //read n (<= 512) bytes from source.
         //write n bytes to target.
         i += n;
         app.updateProgressBar(i, sizeOfSource);
      }
   }
}
```

Currently FileCopier depends on MainApp, making it unreusable in this text mode application. To solve this problem, we can make a copy of FileCopier in the text mode application, modify it and then make its code look identical to the FileCopier in MainApp:

```
interface CopyMonitor {
   void updateProgress(int noBytesCopied, int sizeOfSource);
}
class FileCopier {
   CopyMonitor copyMonitor;
   FileCopier(CopyMonitor copyMonitor) {
      this.copyMonitor = copyMonitor;
   }
   void copyFile(File source, File target) {
      int sizeOfSource = (int)source.length();
      for (int i = 0; i < sizeOfSource; ) {
         //read n (<= 512) bytes from source.
         //write n bytes to target.
         i += n;
         copyMonitor.updateProgress(i, sizeOfSource);
      }
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

In each of the two applications, we need to create a class to implement the CopyMonitor interface:

```
class MainApp extends JFrame {
   ProgressBar progressBar;
   void main() {
      FileCopier fileCopier = new FileCopier(new CopyMonitor() {
         void updateProgress(int noBytesCopied, int sizeOfSource) {
                  updateProgressBar(noBytesCopied, sizeOfSource);
         }
      });
      fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
   }
   void updateProgressBar(int noBytesCopied, int sizeOfSource) {
      progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
   }
}
class TextApp {
   void main() {
      FileCopier fileCopier = new FileCopier(new CopyMonitor() {
         void updateProgress(int noBytesCopied, int sizeOfSource) {
                  System.out.println(noBytesCopied*100/sizeOfSource);
         }
      });
      fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
   }
}
```

Since both implementation classes need to calculate the percentage, we may just pass the percentage instead:

```
class MainApp extends JFrame {
   ProgressBar progressBar;
   void main() {
      FileCopier fileCopier = new FileCopier(new CopyMonitor() {
         void updateProgress(int completionPercentage) {
            updateProgressBar(completionPercentage);
         }
      });
      fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
   }
   void updateProgressBar(int percentage) {
      progressBar.setPercentage(percentage);
   }
}
class TextApp {
   void main() {
      FileCopier fileCopier = new FileCopier(new CopyMonitor() {
         void updateProgress(int completionPercentage) {
            System.out.println(completionPercentage);
         }
      });
      fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
   }
}
interface CopyMonitor {
   void updateProgress(int completionPercentage);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class FileCopier {
   CopyMonitor copyMonitor;
   FileCopier(CopyMonitor copyMonitor) {
      this.copyMonitor = copyMonitor;
   }
   void copyFile(File source, File target) {
      int sizeOfSource = (int)source.length();
      for (int i = 0; i < sizeOfSource; ) {
         //read n (<= 512) bytes from source.
         //write n bytes to target.
         i += n;
         copyMonitor.updateProgress(i*100/sizeOfSource);
      }
   }
}
```

2. In the above case, suppose that you need to develop another text mode file copying application that will display a "*" for each 10% of progress in its text console. What should you do?

   Just create another class to implement CopyMonitor:

```
class AnotherTextApp {
   void main() {
      FileCopier fileCopier = new FileCopier(new CopyMonitor() {
         int noStarsPrinted = 0;
         void updateProgress(int completionPercentage) {
            int noStarsToPrint = completionPercentage/10;
            while (noStarsPrinted<noStarsToPrint) {
               System.out.println("*");
               noStarsToPrint++;
            }
         }
      });
      fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
   }
}
```

3. Point out the problem in the code below. Further suppose that you need to reuse the fax machine code in another application. What should you do?

```
class MainApp {
   String faxNo;
   void main() {
      FaxMachine faxMachine = new FaxMachine(this);
      faxMachine.sendFax("783675", "hello");
   }
}
class FaxMachine {
   MainApp app;
   FaxMachine(MainApp app) {
      this.app = app;
   }
   void sendFax(String toFaxNo, String msg) {
      FaxMachineHardware hardware = ...;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      hardware.setStationId(app.getFaxNo());
      hardware.setRecipientFaxNo(toFaxNo);
      hardware.start();
      try {
         do {
            Graphics graphics = hardware.newPage();
            //draw the msg into the graphics.
         } while (more page is needed);
      } finally {
         hardware.done();
      }
   }
}
```

Currently FaxMachine depends on MainApp, making it unreusable in another application. Actually, what FaxMachine really needs is just the fax number to serve as the station ID. To solve this problem, we can just pass the fax number instead of the MainApp into the FaxMachine:

```
class MainApp {
   String faxNo;
   void main() {
      FaxMachine faxMachine = new FaxMachine(faxNo);
      faxMachine.sendFax("783675", "hello");
   }
}
class FaxMachine {
   String stationId;
   FaxMachine(String stationId) {
      this.stationId = stationId;
   }
   void sendFax(String toFaxNo, String msg) {
      FaxMachineHardware hardware = ...;
      hardware.setStationId(stationId);
      hardware.setRecipientFaxNo(toFaxNo);
      hardware.start();
      try {
         do {
            Graphics graphics = hardware.newPage();
            //draw the msg into the graphics.
         } while (more page is needed);
      } finally {
         hardware.done();
      }
   }
}
```

4. Point out the problem in the code below. Further suppose that you need to reuse the heat sensor code in another application. What should you do?

```
class Cooker {
   HeatSensor heatSensor;
   Speaker speaker;
   void alarm() {
      speaker.setFrequency(Speaker.HIGH_FREQUENCY);
      speaker.turnOn();
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
class HeatSensor {
   Cooker cooker;
   HeatSensor(Cooker cooker) {
      this.cooker = cooker;
   }
   void checkOverHeated() {
      if (isOverHeated()) {
         cooker.alarm();
      }
   }
   boolean isOverHeated() {
      ...
   }
}
```

Currently HeatSensor depends on Cooker, making it unreusable in another application. Actually, what HeatSenor really needs is just an alarm, not the Cooker. To solve this problem, we can just pass the alarm instead of the Cooker into the HeatSensor:

```
interface Alarm {
   void turnOn();
}
class Cooker {
   HeatSensor heatSensor;
   Speaker speaker;
   Alarm getAlarm() {
      return new Alarm() {
         void turnOn() {
            speaker.setFrequency(Speaker.HIGH_FREQUENCY);
            speaker.turnOn();
         }
      };
   }
}
class HeatSensor {
   Alarm alarm;
   HeatSensor(Alarm alarm) {
      this.alarm = alarm;
   }
   void checkOverHeated() {
      if (isOverHeated()) {
         alarm.turnOn();
      }
   }
   ...
}
```

5. This is a word processor application. It can let the user select the font. Before the user confirms to change the font, it will let the user preview the effect of the change. The current code is shown below. Point out the code smell. If we need to reuse the ChooseFontDialog in a GUI application that doesn't support this preview functionality, how should you change the code?

```
class WordProcessorMainFrame extends JFrame {
   void onChangeFont() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      ChooseFontDialog chooseFontDialog = new ChooseFontDialog(this);
      if (chooseFontDialog.showOnScreen()) {
         Font newFont = chooseFontDialog.getSelectedFont();
         //show the contents using this new font.
      } else {
         //show the contents using the existing font.
      }
   }
   void previewWithFont(Font font) {
      //show the contents using this preview font.
   }
}
class ChooseFontDialog extends JDialog {
   WordProcessorMainFrame mainFrame;
   Font selectedFont;
   ChooseFontDialog(WordProcessorMainFrame mainFrame) {
      this.mainFrame = mainFrame;
   }
   booleawn showOnScreen() {
      ...
   }
   void onSelectedFontChange() {
      selectedFont = getSelectedFontFromUI();
      mainFrame.previewWithFont(selectedFont);
   }
   Font getSelectedFontFromUI() {
      ...
   }
   Font getSelectedFont() {
      return selectedFont;
   }
}
```

The code smell is the circular references between WordProcessorMainFrame and ChooseFontDialog. In order to use ChooseFontDialog in a GUI application that does not support font change preview, we need to stop ChooseFontDialog from referencing WordProcessorMainFrame. Actually, what ChooseFontDialog really needs is to notify another object of the temporary font change, so that the other object may let the user preview the effect of the font change.

```
interface FontChangeListener {
   void onFontChanged(Font newFont);
}
class WordProcessorMainFrame extends JFrame {
   void onChangeFont() {
      ChooseFontDialog chooseFontDialog = new ChooseFontDialog(
         new FontChangeListener() {
            void onFontChanged(Font newFont) {
               previewWithFont(newFont);
            }
         });
      if (chooseFontDialog.showOnScreen()) {
         Font newFont = chooseFontDialog.getSelectedFont();
         //show the contents using this new font.
      } else {
         //show the contents using the existing font.
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
    void previewWithFont(Font font) {
       //show the contents using this preview font.
    }
}
class ChooseFontDialog extends JDialog {
    FontChangeListener fontChangeListener;
    Font selectedFont;
    ChooseFontDialog(FontChangeListener fontChangeListener) {
       this.fontChangeListener = fontChangeListener;
    }
    boolean showOnScreen() {
       ...
    }
    void onSelectedFontChange() {
       selectedFont = getSelectedFontFromUI();
       fontChangeListener.onFontChanged(selectedFont);
    }
    Font getSelectedFontFromUI() {
       ...
    }
    Font getSelectedFont() {
       return selectedFont;
    }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 7

Separate Database, User Interface and Domain Logic

Example

This is a conference management system. The system needs to record the ID, name, telephone of each participant and the region he comes from. The participant ID is a unique integer assigned by the conference organizer. The name must be specified. Telephone can be omitted. All the participants must come from China, US or Europe.

We have created a database and a table to store the information of the participants. The schema is:

```
create table Participants (
   id int primary key,
   name varchar(20) not null,
   telNo varchar(20),
   region varchar(20)
);
```

We have also written the code below to let an operator add a new participant. The system will check all the existing participants to find the maximum ID and then adds one to it and use it as the default value for the ID of this new participant. The operator may use this ID or use any other value. Please read the code carefully:

```
class AddParticipantDialog extends JDialog {
   Connection dbConn;
   JTextField id;
   JTextField name;
   JTextField telNo;
   JTextField region;
   AddParticipantDialog() {
      setupComponents();
      dbConn = ...;
   }
   void setupComponents() {
      ...
   }
   void show() {
      showDefaultValues();
      setVisible(true);
   }
   void showDefaultValues() {
      int nextId;
      PreparedStatement st =
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            dbConn.prepareStatement("select max(id) from participants");
         try {
            ResultSet rs = st.executeQuery();
            try {
               rs.next();
               nextId = rs.getInt(1)+1;
            } finally {
               rs.close();
            }
         } finally {
            st.close();
         }
         id.setText(new Integer(nextId).toString());
         name.setText("");
         region.setText("China");
      }
      void onOK() {
         if (name.getText().equals("")) {
            JOptionPane.showMessageDialog(this, "Invalid name");
            return;
         }
         if (!region.equals("China") &&
            !region.equals("US") &&
            !region.equals("Europe")) {
            JOptionPane.showMessageDialog(this, "Region is unknown");
            return;
         }
         PreparedStatement st =
            dbConn.prepareStatement("insert into from participants values(?,?,?,?)");
         try {
            st.setInt(1, Integer.parseInt(id.getText()));
            st.setString(2, name.getText());
            st.setString(3, telNo.getText());
            st.setString(4, region.getText());
            st.executeUpdate();
         } finally {
            st.close();
         }
         dispose();
      }
   }
```

This code looks normal, but it has mixed three different types of code together:

1. User interface: JDialog, JTextField, event handling.

2. Database access: Connection, PreparedStatement, SQL statements, ResultSet and etc.

3. Domain logic: Default value of the participant, that the name must be specified, the restriction on the region and etc. Domain logic is also called "domain model" or "business logic".

That these different types of code are mixed together causes the following problems:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- The code is complicated.
- The code is hard to reuse. If we need to create an EditParticipantDialog for the operator to edit a participant, we would like to reuse some domain logic (e.g., the restriction on the region). But currently the code implementing the domain logic is mixed with AddParticipantDialog and cannot be reused. It will be more difficult to reuse the domain logic in a web application.
- The code is hard to test. To test it, we have to setup a database and test it through the user interface.
- If the database schema is changed, AddParticipantDialog and many other places will have to be changed accordingly.
- It leads us to think in terms of low level concepts such as fields and records in database, instead of classes, objects, methods and attributes.

Therefore, we should separate these three types of code (user interface, database access and domain logic).

Extract the database access code

We will first extract the database access code. We can consider all the participants in the database as a set, i.e., there is no duplicate elements (participants) and there is no particular ordering between them. The set should support the operations of add, delete, update and enumeration:

```
class Participant {
   int id;
   String name;
   String telNo;
   String region;
   ...
}
interface ParticipantIterator {
   boolean next();
   Participant getParticipant();
}
class Participants {
   Connection dbConn;
   Participants() {
      dbConn = ...;
   }
   void addParticipant(Participant part) {
      PreparedStatement st =
         dbConn.prepareStatement("insert into from participants values(?,?,?,?)");
      try {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            st.setInt(1, part.getId());
            st.setString(2, part.getName());
            st.setString(3, part.getTelNo());
            st.setString(4, part.getRegion());
            st.executeUpdate();
         } finally {
            st.close();
         }
      }
      void removeParticipant(int partId) {
         ...
      }
      void updateParticipant(Participant part) {
         ...
      }
      ParticipantIterator getAllParticipantsById() {
         ...
      }
      ParticipantIterator getParticipantsWithNameById(String name) {
         ...
      }
   }
```

In AddParticipantDialog in addition to adding a new participant, we also need to find the maximum participant ID so far. Therefore, we need to define a getMaxId method:

```
   class Participants {
      Connection dbConn;
      void addParticipant(Participant part) {
         ...
      }
      int getMaxId() {
         PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
         try {
            ResultSet rs = st.executeQuery();
            try {
               rs.next();
               return rs.getInt(1);
            } finally {
               rs.close();
            }
         } finally {
            st.close();
         }
      }
      ...
   }
```

Now, AddParticipantDialog can be simplified as:

```
   class AddParticipantDialog extends JDialog {
      Participants participants;
      JTextField id;
      JTextField name;
      JTextField telNo;
      JTextField region;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    AddParticipantDialog(Participants participants) {
       this.participants = participants;
       setupComponents();
    }
    void setupComponents() {
       ...
    }
    void show() {
       showDefaultValues();
       setVisible(true);
    }
    void showDefaultValues() {
       int nextId = participants.getMaxId()+1;
       id.setText(new Integer(nextId).toString());
       name.setText("");
       telNo.setText("");
       region.setText("China");
    }
    void onOK() {
       if (name.getText().equals("")) {
          JOptionPane.showMessageDialog(this, "Invalid name");
          return;
       }
       if (!region.equals("China") &&
          !region.equals("US") &&
          !region.equals("Europe")) {
          JOptionPane.showMessageDialog(this, "Region is unknown");
          return;
       }
       Participant part = new Participant(
                     Integer.parseInt(id.getText()),
                     name.getText(),
                     telNo.getText(),
                     region.getText());
       participants.addParticipant(part);
       dispose();
    }
}
```

Now, AddParticipantDialog is much simpler. From its code we can't see that it is accessing the database at all!

Flexibility created by extracting the database access code

Because AddParticipantDialog now only deals with a set of participants instead of a database table, you may well use an XML file or a regular text file to store these participants. You only need to modify the Participants class, without changing AddParticipantDialog:

```
class Participants {
   void addParticipant(Participant part) {
      //save the participant into an XML file.
   }
   void getMaxId() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
         //find the maximum Id from the XML file.
      }
   }
```

Even more, you may in some cases use a database and in some other cases use an XML file. All you need is to turn the Participants class into an interface and use the database in one implementation class and use an XML file in another:

```
interface Participants {
   void addParticipant(Participant part);
   int getMaxId();
   ...
}
class ParticipantsInDB implements Participants {
   void addParticipant(Participant part) {
      ...
   }
   int getMaxId() {
      ...
   }
}
class ParticipantsInXMLFile implements Participants {
   void addParticipant(Participant part) {
      //save the participant into an XML file.
   }
   int getMaxId() {
      //find the maximum Id from the XML file.
   }
}
class AddParticipantDialog extends JDialog {
   AddParticipantDialog(Participants participants) {
      ...
   }
   ...
}
```

Separate domain logic and user interface

Now, we will separate the domain logic and the user interface:

```
class Participant {
   int id;
   String name;
   String telNo;
   String region;
   ...
   static Participant makeDefaultParticipant() {
      return new Participant(0, "", "", "China");
   }
   void assertValid() throws ParticipantException {
      if (name.equals("")) {
         throw new ParticipantException("Invalid name");
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        }
        if (!region.equals("China") &&
            !region.equals("US") &&
            !region.equals("Europe")) {
            throw new ParticipantException("Region is unknown");
        }
    }
}
class ParticipantException extends Exception {
    ParticipantException(String msg) {
        super(msg);
    }
}
```

Now, AddParticipantDialog can be simplified as:

```
class AddParticipantDialog extends JDialog {
    Participants participants;
    JTextField id;
    JTextField name;
    JTextField telNo;
    JTextField region;
    AddParticipantDialog(Participants participants) {
        this.participants = participants;
        setupComponents();
    }
    void setupComponents() {
        ...
    }
    void show() {
        showDefaultValues();
        setVisible(true);
    }
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        newPart.setId(participants.getMaxId()+1);
        showParticipant(newPart);
    }
    void showParticipant(Participant part) {
        id.setText(new Integer(part.getId()).toString());
        name.setText(part.getName());
        telNo.setText(part.getTelNo());
        region.setText(part.getRegion());
    }
    Participant makeParticipant() throws ParticipantException {
        Participant part = new Participant(
                        Integer.parseInt(id.getText()),
                        name.getText(),
                        telNo.getText(),
                        region.getText());
        part.assertValid();
        return part;
    }
    void onOK() {
        try {
            participants.addParticipant(makeParticipant());
            dispose();
        } catch (Exception e) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            JOptionPane.showMessageDialog(this, e.getMessage());
        }
    }
}
```

Now, AddParticipantDialog is even simpler. In particular, what it basically does is: show the data in a Participant object through the various UI components (showParticipant) and use the data in the UI components to make a Participant object (makeParticipant). Right, these are exactly the things that a user interface should do.

The improved code

The improved code is shown below:

```
class Participant {
   int id;
   String name;
   String telNo;
   String region;
   ...
   static Participant makeDefaultParticipant() {
      return new Participant(0, "", "", "China");
   }
   void assertValid() throws ParticipantException {
      if (name.equals("")) {
         throw new ParticipantException("Invalid name");
      }
      if (!region.equals("China") &&
         !region.equals("US") &&
         !region.equals("Europe")) {
         throw new ParticipantException("Region is unknown");
      }
   }
}
class ParticipantException extends Exception {
   ParticipantException(String msg) {
      super(msg);
   }
}
class Participants {
   Connection dbConn;
   Participants() {
      dbConn = ...;
   }
   void addParticipant(Participant part) {
      PreparedStatement st =
         dbConn.prepareStatement("insert into from participants values(?,?,?,?)");
      try {
         st.setInt(1, part.getId());
         st.setString(2, part.getName());
         st.setString(3, part.getTelNo());
         st.setString(4, part.getRegion());
         st.executeUpdate();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        } finally {
           st.close();
        }
     }
     int getMaxId() {
        PreparedStatement st =
           dbConn.prepareStatement("select max(id) from participants");
        try {
           ResultSet rs = st.executeQuery();
           try {
              rs.next();
              return rs.getInt(1);
           } finally {
              rs.close();
           }
        } finally {
           st.close();
        }
     }
  }
  class AddParticipantDialog extends JDialog {
     Participants participants;
     JTextField id;
     JTextField name;
     JTextField telNo;
     JTextField region;
     AddParticipantDialog(Participants participants) {
        this.participants = participants;
        setupComponents();
     }
     void setupComponents() {
        ...
     }
     void show() {
        showDefaultValues();
        setVisible(true);
     }
     void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        newPart.setId(participants.getMaxId()+1);
        showParticipant(newPart);
     }
     void showParticipant(Participant part) {
        id.setText(new Integer(part.getId()).toString());
        name.setText(part.getName());
        telNo.setText(part.getTelNo());
        region.setText(part.getRegion());
     }
     Participant makeParticipant() throws ParticipantException {
        Participant part = new Participant(
                       Integer.parseInt(id.getText()),
                       name.getText(),
                       telNo.getText(),
                       region.getText());
        part.assertValid();
        return part;
     }
     void onOK() {
        try {
           participants.addParticipant(makeParticipant());
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            dispose();
         } catch (Exception e) {
            JOptionPane.showMessageDialog(this, e.getMessage());
         }
      }
   }
```

SQLException is giving us away

In fact, there is still one problem in the code. Let's read the code the Participants class carefully:

```
class Participants {
   ...
   int getMaxId() {
      PreparedStatement st =
         dbConn.prepareStatement("select max(id) from participants");
      try {
         ResultSet rs = st.executeQuery();
         try {
            rs.next();
            return rs.getInt(1);
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
}
```

When calling methods like prepareStatement, executeQuery, next and close, JDBC may throw an SQLException. Because getMaxId by itself doesn't know how to handle this exception, we can only let the caller decide how to handle it. So, we need to change the code to:

```
class Participants {
   ...
   void getMaxId() throws SQLException {
      PreparedStatement st =
         dbConn.prepareStatement("select max(id) from participants");
      try {
         ResultSet rs = st.executeQuery();
         try {
            rs.next();
            return st.getInt(1);
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Here comes the problem: When AddParticipantDialog calls the getMaxId method, it must be prepared to handle an SQLException (catch it or re-throw it), e.g.:

```
class AddParticipantDialog extends JDialog {
   ...
   void showDefaultValues() {
      Participant newPart = Participant.makeDefaultParticipant();
      try {
         newPart.setId(participants.getMaxId()+1);
      } catch (SQLException e) {
         ...
      }
      showParticipant(newPart);
   }
}
```

Now, because AddParticipantDialog needs to handle SQLException, it is obviously dealing with a database instead of a simple set of participants or an XML file. That is, SQLException is giving us away, exposing the fact that the Participants class uses a database. It forces all the code using the Participants class such as AddParticipantDialog to be used along with a database. All this client code cannot be reused in any environment where there is no database.

To solve this problem, when the Participants class encounters an error, it shouldn't throw an SQLException (which is related to a database). Instead, it should throw an exception related to a set of participants:

```
class ParticipantsException extends Exception {
   ParticipantsException(Throwable cause) {
      super(cause);
   }
}
class Participants {
   ...
   int getMaxId() throws ParticipantsException {
      try {
         PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
         try {
            ResultSet rs = st.executeQuery();
            try {
               rs.next();
               return rs.getInt(1);
            } finally {
               rs.close();
            }
         } finally {
            st.close();
         }
      } catch (SQLException e) {
         //use e as the cause. It means the SQLException is the cause
         //for this ParticipantsException.
         throw new ParticipantsException(e);
      }
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
```

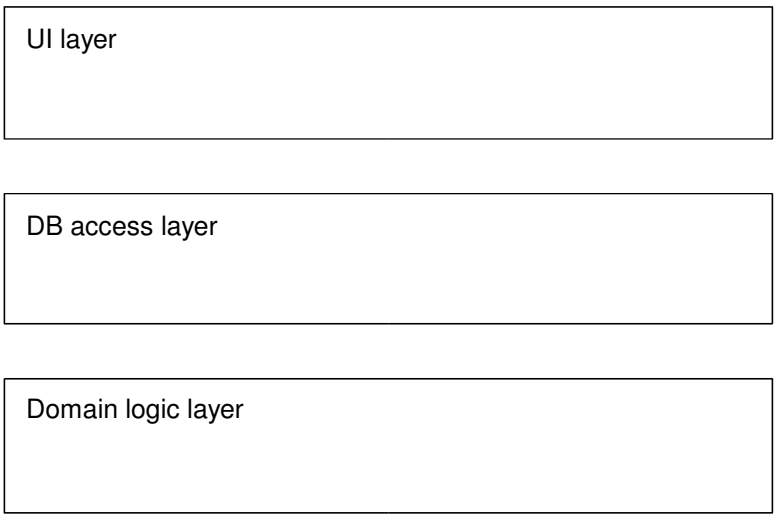
Now, AddParticipantDialog only needs to handle a ParticipantsException, not an SQLException, e.g.:

```
class AddParticipantDialog extends JDialog {
   ...
   void showDefaultValues() {
      Participant newPart = Participant.makeDefaultParticipant();
      try {
         newPart.setId(participants.getMaxId()+1);
      } catch (ParticipantsException e) {
         ...
      }
      showParticipant(newPart);
   }
}
```

Divide the system into layers

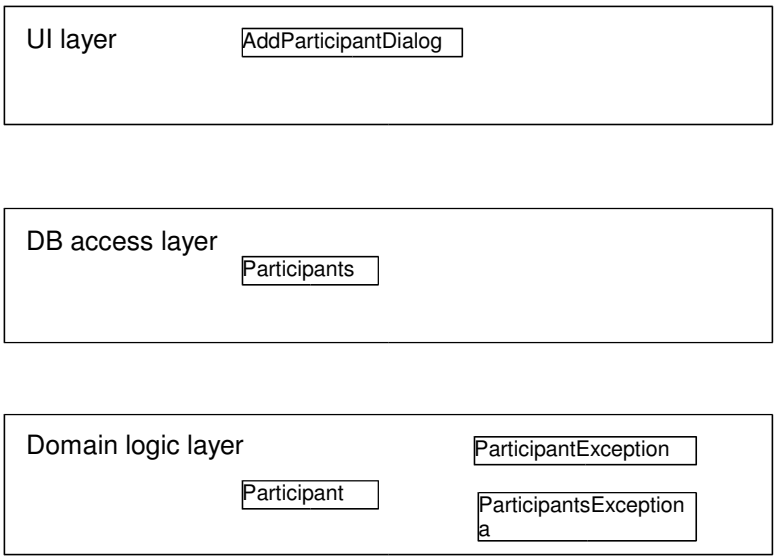
We can divide the code in the above conference system into three layers: "database access layer", "domain layer" and "user interface layer":

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

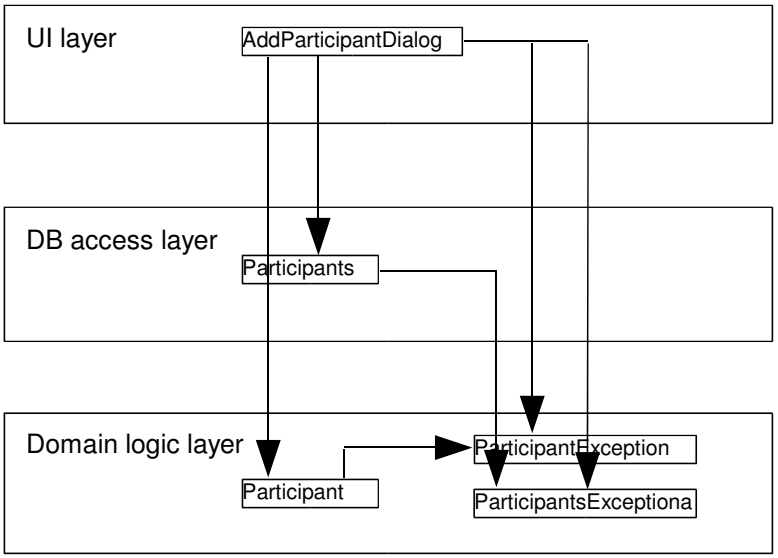


Because Participants accesses the database, we put it into the database access layer. Because Participant, ParticipantException, ParticipantsException deals with the domain logic, we put them into the domain logic layer. Because AddParticipantDialog interfaces with the user, we put it into the user interface layer:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

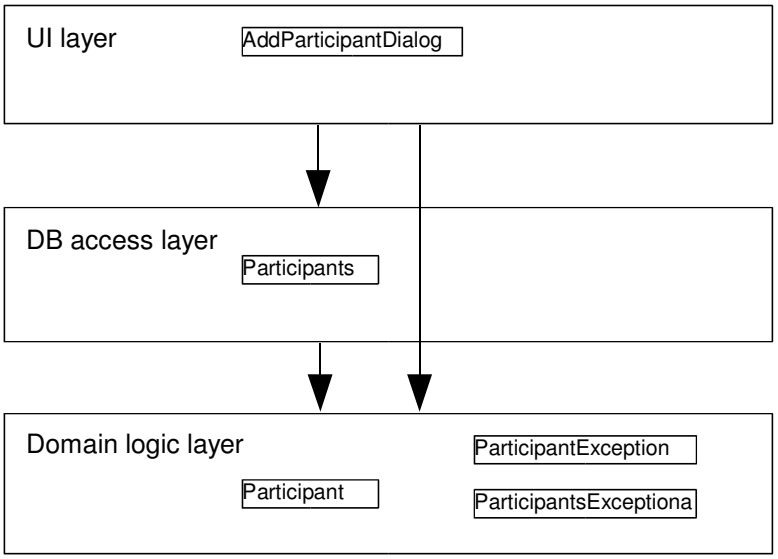


If class A references class B, in the picture we draw an arrow from A to B:



If we only show the references between the layers, the picture above becomes:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



Note some points in the picture:

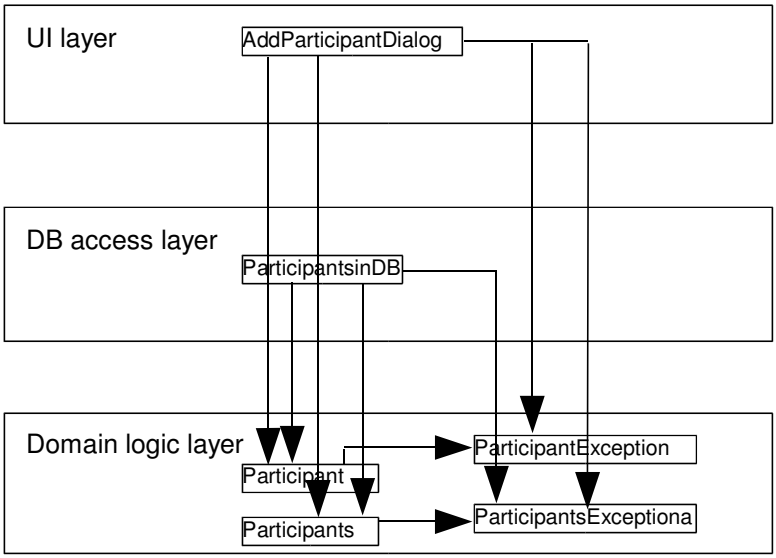
1. The domain logic layer is referenced by the other two layers but not the other way around. Therefore, this layer is easiest to reuse.

2. The database access layer references the domain logic layer but not the user interface layer. Therefore, no matter the system is text-based, GUI-based or web-based, this layer can be reused.

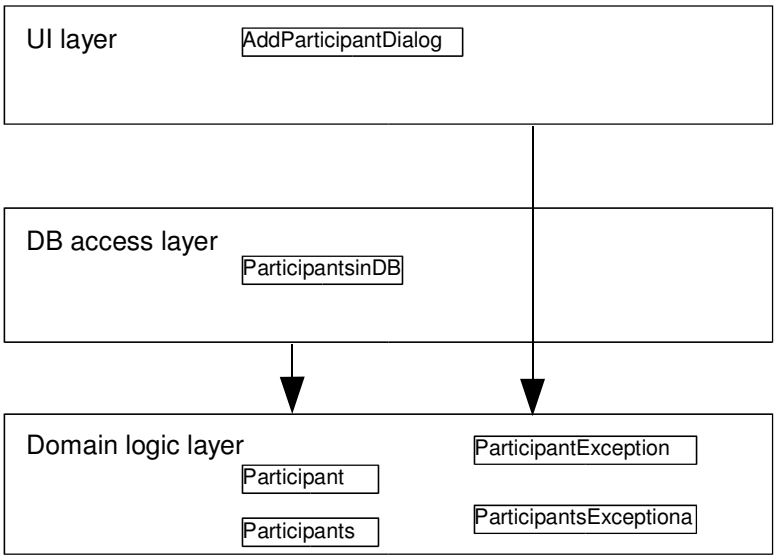
3. The user interface layer references the other two layers. Therefore, this layer is hardest to reuse.

The third point (the user interface layer uses the other two layers) needs some attention: Haven't we made AddParticipantDialog (user interface layer) unaware of the database? Why in the picture it is still referencing the database access layer? The problem comes from the Participants class. The interface of the Participants class has nothing to do with the database. Therefore, its interface belongs to the domain logic layer. However, the implementation of the Participants class is indeed about the database. Therefore, its implementation belongs to the database access layer. Therefore, putting the whole Participants class into the database access layer is not entirely correct. If we need to make it clearer, as mentioned before, we should make Participants an interface and put the current implementation of Participants into a ParticipantsInDB class. Then, the Participants interface will belong to the domain logic layer, while the ParticipantsInDB class will belong to the database access layer:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



If we only show the references between the layers, the picture above becomes:



Now, the user interface layer only references the domain logic layer, but not the database access layer. Therefore, no matter the system uses a database, a text file or an XML file to store its data, this layer can be reused.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Now, there are only two the references between the layers:

1. The user interface layer references the domain logic layer.
2. The database access layer references the domain logic layer.

Not all systems can be divided into three layers like this, with only two references between the layers. For example, the original AddParticipantDialog cannot be classified into any one of these layers. Therefore, the original system cannot be divided like this. This is possible only with systems that are well structured. Therefore, we should set this as our target (in particular, for larger systems).

Many things are user interfaces

Many things are user interfaces. It not only includes windows, buttons and etc., but also includes reports, servlet, jsp, text console and etc. Therefore, do not implement domain logic or database access in servlets (In particular, watch out for those servlets containing lots of code. They are the prime suspects); Nor should you call System.out.print in domain logic.

Other methods

Treating the database as a set of objects is only one way to hide the database. For the other methods, please see the references.

References

- http://c2.com/cgi/wiki?ScatterSqlEverywhere.
- http://c2.com/cgi/wiki?ModelFirst.
- http://c2.com/cgi/wiki?ObjectRelationalMapping.
- Scott Ambler's article at http://www.agiledata.org/essays/mappingObjects.html discusses how to map objects of complicated relationship onto the database.
- Martin Fowler's article at http://www.martinfowler.com/articles/dblogic.html discusses the respective advantages and disadvantages of using a programming language and SQL to express domain logic.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- Martin Fowler has put forth several patterns to access a database (Table Data Gateway, Row Data Gateway, Active Record and Data Mapper): http://www.martinfowler.com/eaaCatalog.

- Robert C. Martin's article at http://www.objectmentor.com/resources/articles/Proxy.pdf describes how to use patterns like Proxy and Stairway to Heaven to encapsulate the access to a database.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. Implement the updateParticipant and getAllParticipantsById methods in the Participants class:

```
class Participants {
   ...
   void updateParticipant(Participant part) {
      //write code here.
   }
   ParticipantIterator getAllParticipantsById() {
      //write code here.
   }
}
```

2. This application is about restaurants. An order has an order ID, a restaurant ID and a customer ID and contains some order items. Each order item contains the food ID, the quantity and unit price. You have created the tables and classes below.

```
create table Orders (
   orderId varchar(20) primary key,
   customerId varchar(20) not null,
   restId varchar(20) not null
);
create table OrderItems (
   orderId varchar(20),
   itemId int,
   foodId varchar(20) not null,
   quantity int not null,
   unitPrice float not null,
   primary key(orderId, itemId)
);
class Order {
   String orderId;
   String customerId;
   String restId;
   OrderItem items[];
}
class OrderItem {
   String foodId;
   int quantity;
   double unitPrice;
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Your tasks are:

1. Create an interface for accessing the orders while hiding the database.

2. Create a class to implement that interface and show how to implement its method for adding an order to the database.

3. Identify which layers they belong to.

3. In the application above, you would like to keep record of the orders that have been delivered and if so, the delivery time. As you have attended the CPTTM Object Oriented Design course, you know it is good to keep the Order class slim, so you decide to create a new class, while keeping the Order class unchanged. So you have written the code below:

```
class OrderDelivery {
   String orderId;
   Date deliveryTime;
}
interface OrderDeliveries {
   boolean isDelivered(String orderId);
   Date getDeliveryTime(String orderId);
   void markAsDelivered(String orderId, Date deliveryTime);
}
```

To store this information in the database, you ask the database administrator. He says that it is most efficient to just add a datetime field to the Orders table. If that field is NULL, it means the order has not been delivered:

```
create table Orders (
   orderId varchar(20) primary key,
   customerId varchar(20) not null,
   restId varchar(20) not null,
   deliveryTime datetime
);
```

Your tasks are:

1. Create a class to implement OrderDeliveries and show how to implement its method for the isDelivered and markAsDelivered methods.

2. Determine if you need to modify your code done in the previous question. If so, where?

3. Determine what happens to the order delivery if an order is deleted?

4. This program below implements a game called "Hangman". The game is played like this: the computer "comes up with" a secret such as "banana". The task of the player is to try to find out this secret. Every turn the player can input one english letter such as "a", then the computer will show the "a" letters in the secret (if any), while the other letters will be shown

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

as a dash. For example, in this case, the computer will show "-a-a-a". In the next turn if the player inputs "b", the computer will show "ba-a-a". In the next turn if the player inputs "c", the computer will still show "ba-a-a" because there is no "c" in the secret. The player has at most 7 turns. If he can find out the secret within 7 turns, he wins. Otherwise he loses. If the player inputs say "b" again, the computer will tell him that it has been guessed before and this guess is not counted (not included in the 7 turns).

Your tasks are:

1. Point out and remove the problems in the code below.

2. Divide your revised code into appropriate layers.

```
class Hangman {
   String secret = "banana";
   String guessedChars = "";
   static public void main(String args[]) {
      new Hangman();
   }
   Hangman() {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      for (int k = 0; k < 7;) { //can guess at most 7 times
         String s = ""; //partially found secret
         for (int i=0; i<secret.length(); i++) {
            char ch = secret.charAt(i);
            if (guessedChars.indexOf(ch)<0) //has it been guessed?
               ch = '-';  //no, hide it. just show a dash.
            s = s+ch;
         }
         System.out.println("Secret: "+s);
         System.out.print("Guess letter: ");
         char ch = br.readLine().charAt(0); //read just one char
         if (guessedChars.indexOf(ch)>=0) { //already guessed?
            System.out.println("You have guessed this char!");
            continue;
         }
         int n = numberOfFoundChars();
         guessedChars = guessedChars+ch;
         int m = numberOfFoundChars();
         if (m>n) {
            System.out.println("Success, you have found letter "+ch);
            System.out.println("Letters found: "+m);
         }
         if (m==secret.length()) {
            System.out.println("You won!");
            return;
         }
         k++;
      }
      System.out.println("You lost!");
   }
   int numberOfFoundChars() {
      int n = 0;
      for (int i=0; i< secret.length(); i++) {
         char ch = secret.charAt(i);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
         if (guessedChars.indexOf(ch)>=0)
            n++;
      }
      return n;
   }
}
```

5. This is a web-based application concerned with training courses. A user can choose a subject area ("IT", "Management", "Language", "Apparel") in a form and click submit. Then the application will display all the courses in that subject area. The servlet doing that is shown below.

```
public class ShowCoursesServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<HTML><TITLE>Course listing</TITLE><BODY>");
      Connection dbConn = ...;
      PreparedStatement st =
         dbConn.prepareStatement("select * from Courses where subjArea=?");
      try {
         st.setString(1, request.getParameter("SubjArea"));
         ResultSet rs = st.executeQuery();
         try {
            out.println("<TABLE>");
            while (rs.next()) {
               out.println("<TR>");
               out.println("<TD>");
               out.println(rs.getString(1)); //course code
               out.println("</TD>");
               out.println("<TD>");
               out.println(rs.getString(2)); //course name
               out.println("</TD>");
               out.println("<TD>");
               out.println(""+rs.getInt(3)); //course fee in MOP
               out.println("</TD>");
               out.println("</TR>");
            }
            out.println("</TABLE>");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
      out.println("</BODY></HTML>");
 }
}
```

Your tasks are:

1. Point out and remove the problems in the code.

2. Divide your revised code into appropriate layers.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Hints

1. To implement getAllParticipantsById, you need to create a class implementing ParticipantIterator that reads the data from a ResultSet.

2. You should have methods to add, delete or update an order and retrieve a selection of orders. For example, in the add method, you need to add one record to the Orders table and add multiple records to the OrderItems table.

3. No hint for this one.

4. UI is mixed with the domain logic. Extract the domain logic into a class like Hangman and rename the original one as HangmanApp. Make sure that HangmanApp uses Hangman but not the reverse.

5. No hint for this one.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Implement the updateParticipant and getAllParticipantsById methods in the Participants class:

```
class Participants {
   ...
   void updateParticipant(Participant part) {
      //write code here.
   }
   ParticipantIterator getAllParticipantsById() {
      //write code here.
   }
}
```

   The code may be like:

```
class Participants {
   ...
   void updateParticipant(Participant part) {
      PreparedStatement st =
         dbConn.prepareStatement(
            "update participants set name=?, telNo=?, region=? where id=?");
      try {
         st.setString(1, part.getName());
         st.setString(2, part.getTelNo());
         st.setString(3, part.getRegion());
         st.setInt(4, part.getId());
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   ParticipantIterator getAllParticipantsById() {
      PreparedStatement st =
         dbConn.prepareStatement(
            "select * from participants order by id");
      return new ParticipantResultSetIterator(st);
   }
}
class ParticipantResultSetIterator implements ParticipantIterator {
   PreparedStatement st;
   ResultSet rs;
   ParticipantResultSetIterator(PreparedStatement st) {
      this.st = st;
      this.rs=st.executeQuery();
   }
   boolean next() {
      return rs.next();
   }
   Participant getParticipant() {
      new Participant(
         rs.getInt(1),
         rs.getString(2),
         rs.getString(3),
         rs.getString(4));
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   void finalize() {
      rs.close();
      st.close();
   }
}
```

2. This application is about restaurants. An order has an order ID, a restaurant ID and a customer ID and contains some order items. Each order item contains the food ID, the quantity and unit price. You have created the tables and classes below.

```
create table Orders (
   orderId varchar(20) primary key,
   customerId varchar(20) not null,
   restId varchar(20) not null
);
create table OrderItems (
   orderId varchar(20),
   itemId int,
   foodId varchar(20) not null,
   quantity int not null,
   unitPrice float not null,
   primary key(orderId, itemId)
);
class Order {
   String orderId;
   String customerId;
   String restId;
   OrderItem items[];
}
class OrderItem {
   String foodId;
   int quantity;
   double unitPrice;
}
```

Your tasks are:

1. Create an interface for accessing the orders while hiding the database.

```
interface Orders {
   void addOrder(Order order);
   void deleteOrder(String orderId);
   void updateOrder(Order order);
   OrderIterator getOrdersById();
}
```

2. Create a class to implement that interface and show how to implement its method for adding an order to the database.

```
class OrdersInDB implements Orders {
   void addOrder(Order order) {
      PreparedStatement st =
         dbConn.prepareStatement("insert into from Orders values(?,?,?)");
      try {
         st.setString(1, order.getId());
         st.setString(2, order.getCustomerId());
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            st.setString(3, order.getRestId());
            st.executeUpdate();
         } finally {
            st.close();
         }
         st = dbConn.prepareStatement(
               "insert into from OrderItems values(?,?,?,?,?)");
         try {
            for (int i = 0; i < order.items.length; i++) {
               OrderItem orderItem = order.items[i];
               st.setString(1, order.getId());
               st.setInt(2, i);
               st.setString(3, orderItem.getFoodId());
               st.setInt(4, orderItem.getQuantity());
               st.setDouble(5, orderItem.getUnitPrice());
               st.executeUpdate();
            }
         } finally {
            st.close();
         }
      }
   }
```

3. Identify which layers they belong to.

   Domain: Orders.

   Database access: OrdersInDB.

3. In the application above, you would like to keep record of the orders that have been delivered and if so, the delivery time. As you have attended the CPTTM Object Oriented Design course, you know it is good to keep the Order class slim, so you decide to create a new class, while keeping the Order class unchanged. So you have written the code below:

```
class OrderDelivery {
   String orderId;
   Date deliveryTime;
}
interface OrderDeliveries {
   boolean isDelivered(String orderId);
   Date getDeliveryTime(String orderId);
   void markAsDelivered(String orderId, Date deliveryTime);
}
```

   To store this information in the database, you ask the database administrator. He says that it is most efficient to just add a datetime field to the Orders table. If that field is NULL, it means the order has not been delivered:

```
create table Orders (
   orderId varchar(20) primary key,
   customerId varchar(20) not null,
   restId varchar(20) not null,
   deliveryTime datetime
);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Your tasks are:

1. Create a class to implement OrderDeliveries and show how to implement its method for the isDelivered and markAsDelivered methods.

```
class OrderDeliveriesInDB implements OrderDeliveries {
   boolean isDelivered(String orderId) {
      PreparedStatement st = dbConn.prepareStatement(
            "select deliveryTime from Orders where orderId=?");
      try {
         st.setString(1, orderId);
         ResultSet rs=st.executeQuery();
         try {
            rs.next();
            return rs.getDate(1)!=null;
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
   void markAsDelivered(String orderId, Date deliveryTime) {
      PreparedStatement st = dbConn.prepareStatement(
            "update Orders set deliveryTime=? where orderId=?");
      try {
         st.setDate(1, new java.sql.Date(deliveryTime.getTime()));
         st.setString(2, orderId);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

2. Determine if you need to modify your code done in the previous question. If so, where?

When adding an order record to the Orders table, there is one more field (deliveryTime). It must be NULL to indicate that the order has not bee delivered. Some DBMS may allow you to use the previous code unchanged, while still having this effect. But to be safe than sorry, do it explicitly:

```
class OrdersInDB implements Orders {
   void addOrder(Order order) {
      PreparedStatement st =
         dbConn.prepareStatement("insert into from Orders values(?,?,?,?)");
      try {
         st.setString(1, order.getId());
         st.setString(2, order.getCustomerId());
         st.setString(3, order.getRestId());
         st.setNull(4, java.sql.Types.TIME);
         st.executeUpdate();
      } finally {
         st.close();
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      ...
   }
}
```

3. Determine what happens to the order delivery if an order is deleted?

   Then the order delivery is also deleted. This sounds fine.

4. This program below implements a game called "Hangman". The game is played like this: the computer "comes up with" a secret such as "banana". The task of the player is to try to find out this secret. Every turn the player can input one English letter such as "a", then the computer will show the "a" letters in the secret (if any), while the other letters will be shown as a dash. For example, in this case, the computer will show "-a-a-a". In the next turn if the player inputs "b", the computer will show "ba-a-a". In the next turn if the player inputs "c", the computer will still show "ba-a-a" because there is no "c" in the secret. The player has at most 7 turns. If he can find out the secret within 7 turns, he wins. Otherwise he loses. If the player inputs say "b" again, the computer will tell him that it has been guessed before and this guess is not counted (not included in the 7 turns).

   Your tasks are:

   1. Point out and remove the problems in the code below.

   2. Divide your revised code into appropriate layers.

```
class Hangman {
   String secret = "banana";
   String guessedChars = "";
   static public void main(String args[]) {
      new Hangman();
   }
   Hangman() {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      for (int k = 0; k < 7;) { //can guess at most 7 times
         String s = ""; //partially found secret
         for (int i=0; i<secret.length(); i++) {
            char ch = secret.charAt(i);
            if (guessedChars.indexOf(ch)<0) //has it been guessed?
               ch = '-';  //no, hide it. just show a dash.
            s = s+ch;
         }
         System.out.println("Secret: "+s);
         System.out.print("Guess letter: ");
         char ch = br.readLine().charAt(0); //read just one char
         if (guessedChars.indexOf(ch)>=0) { //already guessed?
            System.out.println("You have guessed this char!");
            continue;
         }
         int n = numberOfFoundChars();
         guessedChars = guessedChars+ch;
         int m = numberOfFoundChars();
         if (m>n) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            System.out.println("Success, you have found letter "+ch);
            System.out.println("Letters found: "+m);
         }
         if (m==secret.length()) {
            System.out.println("You won!");
            return;
         }
         k++;
      }
      System.out.println("You lost!");
   }
   int numberOfFoundChars() {
      int n = 0;
      for (int i=0; i< secret.length(); i++) {
         char ch = secret.charAt(i);
         if (guessedChars.indexOf(ch)>=0)
            n++;
      }
      return n;
   }
}
```

The code mixes model logic with UI. The comments in the code can be removed by making the code as clear as the comments.

We can extract the domain logic into:

```
class Hangman {
   final static int MAXNOGUESSES = 7;
   String secret = "banana";
   String guessedChars = "";
   boolean reachMaxNoGuesses() {
      return getNoGuessedChars()==MAXNOGUESSES;
   }
   int getNoGuessedChars() {
      return guessedChars.length();
   }
   boolean hasBeenGuessed(char ch) {
      return guessedChars.indexOf(ch)>=0;
   }
   String getPartiallyFoundSecret() {
      String partiallyFoundSecret = "";
      for (int i=0; i< secretLength(); i++) {
         char ch = secret.charAt(i);
         char chToShow = hasBeenGuessed(ch) ? ch : '-';
         partiallyFoundSecret = partiallyFoundSecret+chToShow;
      }
      return partiallyFoundSecret;
   }
   boolean guess(char ch) {
      int n = numberOfFoundChars();
      guessedChars = guessedChars+ch;
      int m = numberOfFoundChars();
      return m>n;
   }
   boolean isSecretFound() {
      return numberOfFoundChars()==secretLength();
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   int numberOfFoundChars() {
      int n = 0;
      for (int i=0; i< secretLength(); i++) {
         char ch = secret.charAt(i);
         if (guessedChars.indexOf(ch)>=0)
            n++;
      }
      return n;
   }
   int secretLength() {
      return secret.length();
   }
}
```

The UI will use the domain logic:

```
class HangmanApp {
   static public void main(String args[]) {
      new HangmanApp();
   }
   HangmanApp() {
      Hangman hangman = new Hangman();
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      while (!hangman.reachMaxNoGuesses()) {
         System.out.println("Secret: "+hangman.getPartiallyFoundSecret());
         System.out.print("Guess letter: ");
         char ch = readOneChar(br);
         if (hangman.hasBeenGuessed(ch)) {
            System.out.println("You have guessed this char!");
            continue;
         }
         if (hangman.guess(ch)) {
            System.out.println("Success, you have found letter "+ch);
            System.out.println("Letters found: "+
               hangman.numberOfFoundChars());
         }
         if (hangman.isSecretFound()) {
            System.out.println("You won!");
            return;
         }
      }
      System.out.println("You lost!");
   }
   char readOneChar(BufferedReader br) {
      return br.readLine().charAt(0);
   }
}
```

5. This is a web-based application concerned with training courses. A user can choose a subject area ("IT", "Management", "Language", "Apparel") in a form and click submit. Then the application will display all the courses in that subject area. The servlet doing that is shown below.

```
public class ShowCoursesServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html");
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      PrintWriter out = response.getWriter();
      out.println("<HTML><TITLE>Course listing</TITLE><BODY>");
      Connection dbConn = ...;
      PreparedStatement st =
         dbConn.prepareStatement("select * from Courses where subjArea=?");
      try {
         st.setString(1, request.getParameter("SubjArea"));
         ResultSet rs = st.executeQuery();
         try {
            out.println("<TABLE>");
            while (rs.next()) {
               out.println("<TR>");
               out.println("<TD>");
               out.println(rs.getString(1)); //course code
               out.println("</TD>");
               out.println("<TD>");
               out.println(rs.getString(2)); //course name
               out.println("</TD>");
               out.println("<TD>");
               out.println(""+rs.getInt(3)); //course fee in MOP
               out.println("</TD>");
               out.println("</TR>");
            }
            out.println("</TABLE>");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
      out.println("</BODY></HTML>");
   }
}
```

Your tasks are:

1. Point out and remove the problems in the code.

2. Divide your revised code into appropriate layers.

The code mixes model logic with database and UI. Remember that servlet is UI. We can extract the database access into:

```
interface Courses {
   Course[] getCoursesInSubject(String subjArea);
}
class CoursesInDB implements Courses {
   Course[] getCoursesInSubject(String subjArea) {
      Connection dbConn = ...;
      PreparedStatement st =
         dbConn.prepareStatement("select * from Courses where subjArea=?");
      try {
         st.setString(1, subjArea);
         ResultSet rs = st.executeQuery();
         try {
            Course courses[];
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            while (rs.next()) {
               Course course = new Course(
                  rs.getString(1),
                  rs.getString(2),
                  rs.getInt(3));
               add course to courses;
            }
            return courses;
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
}
```

The UI will use the domain logic (but not the DB) as:

```
public class ShowCoursesServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<HTML><TITLE>Course listing</TITLE><BODY>");
      Courses courses =
         (Courses)getServletContext().getAttribute("Courses");
      Course coursesInSubject[] =
         courses.getCoursesInSubject(request.getParameter("SubjArea"));
      out.println("<TABLE>");
      for (int i = 0; i < coursesInSubject.length; i++) {
         Course course = coursesInSubject[i];
         out.println("<TR>");
         out.println("<TD>");
         out.println(course.getCourseCode());
         out.println("</TD>");
         out.println("<TD>");
         out.println(course.getCourseName());
         out.println("</TD>");
         out.println("<TD>");
         out.println(course.getCourseFeeInMOP());
         out.println("</TD>");
         out.println("</TR>");
      }
      out.println("</TABLE>");
      out.println("</BODY></HTML>");
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 8

Managing Software Projects with User Stories

What is a user story

Suppose that the customer of this project is a manufacturer of vending machines for soft drinks. They request us to develop software to control their vending machines. We can contact their marketing manager for the requirements on this system software. Therefore, their marketing manager is our customer. He once said: "Whenever someone inserts some money, the vending machine will display how much he has inserted so far. If the money is enough to buy a certain type of soft drink, the light inside the button representing that type of soft drink will be turned on. If he presses one such button, the vending machine will dispense a can of soft drink and return the changes to him."

The description above describes a process by which a user may use the system to complete a certain task that is valuable to him (buy a can of soft drink). Such a process is called a "use case" or "user story". That is, what the customer said above describes a user story.

If we would like to write down a user story, we may use the format below:

Name: Sell soft drink

Events:

1. A user inserts some money.
2. The vending machine displays how much money he has inserted so far.
3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.
4. The user presses a certain lit-up button.
5. The vending machine sells a can of soft drink to him.
6. The vending machine returns the changes to him.

Note that the events in a user story typically look like this:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

1. User does XX.
2. System does YY.
3. User does ZZ.
4. System does TT.
5. ...

User story only describes the external behaviors of a system

A user story should only describe the external behaviors of a system that can be understood by the customer. It completely ignores the internal behaviors of the system. For example, the parts underlined in the user story below are internal behaviors and should not appear in the user story at all:

1. A user inserts some money.
2. The vending machine deposits the money into its money box, sends a command to the screen to display how much money he has inserted so far.
3. The vending machine looks up the price list in its database to check if the money is enough to buy a certain type of soft drink. If yes, the vending machine lights up the button representing that type of soft drink.
4. The user presses a certain lit-up button.
5. The vending machine sells a can of soft drink to him and reduces the inventory in the database.
6. The vending machine returns the changes to him.

This is a common mistake when describing user stories, no matter it is in written or spoken form. In particular, never mention things like databases, records or fields that make no sense to the customer. Stay alerted.

Estimate release duration

What are user stories for? Suppose that the customer hopes to have the system delivered in

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

50 days. Are we going to make it? To answer that question, at the beginning of the project, we will try to find out all the user stories and estimate how much time we will need for each. How? For example, we may have gathered the following user stories from the customer:

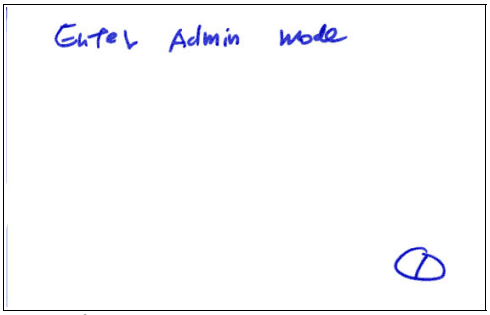
User story	*Brief description*
Sell soft drinks	As mentioned above.
Cancel purchase	After inserting some coins, the user can cancel the purchase.
Enter administration mode	An authorized person can enter administration mode and then add the inventory, set the prices, take the money and etc.
Add soft drinks	An authorized person can add the inventory after entering administration mode.
Take money	An authorized person can take the money after entering administration mode.
Security alarm	If something usually happens, it can turn on its security alarm.
Print monthly sales report	An authorized person can download the data and then print a monthly sales report.

Then find a user story that is among those that are easiest (by "easy", we mean "take less time to implement"). We don't have to be very accurate. Just pick one that feels easy. For example, let's say "enter administration mode" is such a user story. Then we will declare that this user story is worth 1 "story point":

User story	*Story points*
Sell soft drinks	
Cancel purchase	
Enter administration mode	1
Add soft drinks	
Take money	
Security alarm	
Print monthly sales report	

Instead of having a list like above, you will probably use a 4"x6" index card to represent a user story and write the story point on it:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



Such a card is called a "story card".

Then, consider the other user stories. For example, for "take money", we think its should be about as easy as "enter administration mode", so it is also worth 1 story point. We will show it on our list but you will write it on the story card for "take money":

User story	*Story points*
Sell soft drinks	
Cancel purchase	
Enter administration mode	1
Add soft drinks	
Take money	**1**
Security alarm	
Print monthly sales report	

For "cancel purchase", we think it should be about twice as hard as "take money", so it is worth 2 story points:

User story	*Story points*
Sell soft drinks	
Cancel purchase	**2**
Enter administration mode	1
Add soft drinks	
Take money	1
Download sales data to notebook	
Print monthly sales report	

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

For "sell soft drinks", we think it should be twice as hard as "cancel purchase", so it is worth 4 story points:

User story	*Story points*
Sell soft drinks	**4**
Cancel purchase	2
Enter administration mode	1
Add soft drinks	
Take money	1
Security alarm	
Print monthly sales report	

For "add soft drinks", we think it should be harder than "take money" but easier than "sell soft drinks", so it should be somewhere between 1 and 4. Is it harder than "cancel purchase (2)"? We think so, so it is worth 3 story points:

User story	*Story points*
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	**3**
Take money	1
Security alarm	
Print monthly sales report	

Similarly, we think "Security alarm" should be somewhat easier than "add soft drinks", so it is worth 2 story points:

User story	*Story points*
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	**2**

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

User story	*Story points*
Print monthly sales report	

"Print monthly sales report" should be as hard as "sell soft drinks", so it is worth 4 story points and we have totally 17 story points:

User story	*Story points*
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	2
Print monthly sales report	**4**
Total	17

Now pick any user story and estimate how much time it would take you (an individual) to implement. Suppose as we have done something similar to "Take money" before, so we pick it and estimate that it would take 5 days to implement. As this story is worth one story point, it means that we estimate that it takes 5 days to do one story point. As we have totally 17 story points to do, we should need 17x5=85 days to finish the project (if only one developer works on it). Suppose that there are two developers on the team, so we need 85/2=43 days.

So, it seems that we are going to meet the deadline (50 days). But it is too early to tell! This estimate is more a guess than an estimate. Usually developers are extremely poor at workload estimation. In fact it is quite common for people to under-estimate the workload by a factor of four! So, how to get a better estimate?

Actually how fast we can go

To get a better estimate, we should ask the customer to give us two weeks to work on the user stories and measure how many story points we can do. We call a period of two weeks an "iteration".

Which stories to implement in the first iteration? It is totally up to the customer to decide. However, the total number of story points must not exceed our capacity. More specifically, because an iteration contains 10 days (assuming we don't work on weekends) and it is currently estimated that it takes 5 days for one developer to implement one story point, as

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

there are two developers on the team, we should be able to do (10/5)*2=4 story points in one iteration.

Then, customer will chooses any user stories that he would like to get implemented in this iteration. The customer should choose those that he considers the most important, regardless of their apparent ordering, as long as the total doesn't exceed 4 story points. For example, if the sales report and taking the money are the most important stories for him, he can choose them without choosing "sell soft drinks":

User story	***Story points***
Take money	1
Generate monthly sales report in text	2
Total	3

Suppose that when the iteration is ended, we have completed "Take money" but the "Monthly sales report" is not yet completed. Suppose that we estimate that it would take 0.5 story point to complete the missing part. It means we have done only 3-0.5=2.5 story points in this iteration (worse than the original estimate of 4).

This number of 2.5 story points is called our current "velocity". It means 2.5 story points can be done by the team in one iteration. It is very useful for us. It is can be used in two ways.

First, in the next iteration we will assume that we can do 2.5 story points only, and the user stories selected by the customer cannot exceed 2.5 story points.

Second, after getting the velocity of the first iteration, we should re-estimate the release duration. Originally we estimated that it takes 5 days for one developer to do a story point. Now we don't need this estimate anymore, because we have the actual data. We know that the team (2 developers) can finish 2.5 story point in one iteration (10 days). Because we have 17 story points to do for the release, we will need 17/2.5=7 iterations, i.e., 14 weeks, which is 70 days. So, it means we are not going to meet the deadline (50 days)! What should we do?

What if we are not going to make it

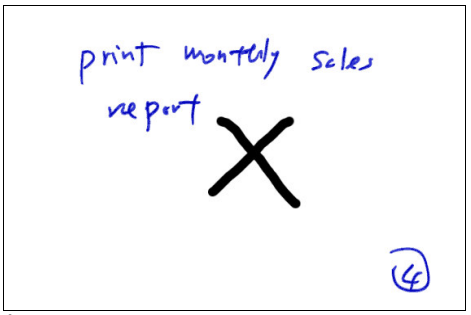
Obviously we can't finish all the user stories. More specifically, in 50 days we can do only 50/10x2.5=12.5 story points. As there are totally 17 story points, we should ask the customer to take out some story cards that are worth 4.5 story points in total and delay them to the next release. The customer should delay those that are not as important as the rest. For example, the customer may delay the report printing:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

User story	*Story points*
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	2
~~Print monthly sales report~~	~~4~~
Total	13

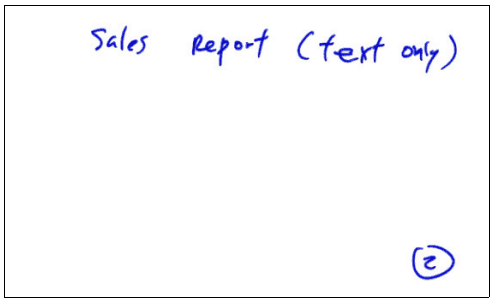
Then he will need to delay another story that is worth at least 0.5 point.

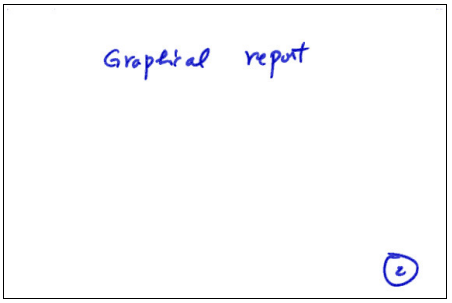
What if the sales report is extremely important for the customer and there is no other stories that can be delayed? Then we may try to simplify some of the stories. For example, originally the sales report is supposed to be done using a third party reporting library and a pie chart is required, if we can generate the report in plain text format (to be imported into Excel for further processing), we think we can reduce the story points from 4 to 2. This will save 2 story points. If the customer agrees, we can split the "print monthly sales report" into two stories "generate monthly sales report in text" and "graphical report" and delay the latter to the next release. You will do that by tearing up the "print monthly sales report" card:



and create two new cards:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org





Then take the "graphical report" card out of this release.

User story	*Story points*
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	2
Generate monthly sales report in text	**2**
Total	**15**

For the other 2.5 story points, we can offer to simplify say "sell soft drinks". Suppose that it is originally supposed to support different types of soft drinks with different prices. If we only support a single type of soft drinks and thus a single price, then we think we can reduce the story point from 4 to 2. If the customer agrees, we can split the "sell soft drinks" into two stories "sell soft drinks (single type)" and "sell multiple soft drink types" and delay the latter to the next release:

User story	*Story points*
Sell soft drinks **(single type)**	**2**
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

User story	*Story points*
Security alarm	2
Generate monthly sales report in text	2
Total	**13**

For the other 0.5 story points, we can offer to simplify say "security alarm". Suppose that it is originally supposed to trigger both the local alarm and inform a specified police station. If we only trigger the local alarm, then we think we can reduce the story point from 2 to 1. If the customer agrees, we can split the "security alarm" story into two stories "security alarm (local)" and "notify police" and delay the latter to the next release:

User story	*Story points*
Sell soft drinks (single type)	2
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm **(local)**	**1**
Generate monthly sales report in text	2
Total	**12**

Now there are only 12 story points to do in the release (<= 12.5). The process above of choosing what stories to include in the current release is called "release planning".

Meeting release deadline by adding developers

In the example above, we try to meet the release deadline by delaying user stories to the next release. This is called "controlling the scope" and is usually the best way. However, if you have tried it but it doesn't work, you may keep the user stories but add more developers. In this example, assume that we would need "n" developers so that in 50 days we can do 17 story points, i.e., (50/10)x2.5x(n/2)=17. It means n=2.7, i.e., we need 3 developers. However:

- A larger team will require more communication between the team members. Therefore doubling the team size will not shorten the release duration by half. It may just shorten it by say 1/3. If the team gets larger than 10 developers, adding more people will probably only slow down the project.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- Adding team members near the end will only slow down the project because the new members don't understand the system enough to do anything useful to it. So it has to be done early.

Steering the project according to the velocity

The velocity of 2.5 story points per iteration is just the velocity of the first iteration, for the second iteration it may turn out to be say 2 or 3 (usually it shouldn't change by too much). If it is 2, then for the third iteration we will assume a velocity of 2 and the user can choose stories totaling up to 2 story points.

For most projects, the velocity will become stable very soon (e.g., after a few iterations). When it does, we should re-estimate the release duration and perform release planning again. If the velocity suggests that we can do say 3 story points instead of 2.5 story points, we will let the customer choose more stories to include in the release. On the contrary, if the velocity suggests that we can do say only 2 story points, we will ask the customer to take out some more stories (split some stories if required), or add more developers if the team is still very small, it is still early in the project and it is absolutely necessary.

How much details do we need to estimate a user story

At the beginning of the project, we need to find out all the major user stories in the release and estimate the story point of each. How to estimate the story points of a user story say like "sell soft drinks"? The name "sell soft drinks" ignores many details such as: What type of money a user can insert? Are notes OK? Is RMB (Chinese currency) OK? What is the intensity of a lit-up button? Can one type of soft drink be presented by two buttons? After pressing a button, will it be turned off? When returning the changes can we simply return lots of 10 cents?

Do we need to find out all these details? We shouldn't need to know the intensity of the button, because it shouldn't affect the workload. However, how the changes should be returned seems to affect the workload and therefore should be determined (returning a heap of 10 cents should be easy; having to use the minimum number of coins may be quite difficult). Whether it needs to handle multiple currencies should also be determined.

In general, we don't need to worry too much about missing the details. For each user story, it is generally enough to ask a few "important" questions.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

What if we can't estimate comfortably

If we can't comfortably estimate a user story, there are some possibilities:

1. The user story is too large. In this case, we can breakdown the user story into several, e.g.:
   - New user story 1: Display the total amount.
   - New user story 2: Light up a button when the total amount is enough.
   - New user story 3: Press a lit-up button to buy soft drink.
2. We have never programmed a vending machine. Therefore, we don't know how hard or easy it is to control it. In this case, we should run some simple experiments such as program to ask a vending machine to return some money. This kind of experiment is called a "spike".

Iteration planning

The whole project will be implemented in iterations. At the start of each iteration, we will let the customer choose those user stories that he would like to get implemented in this iteration. The customer should choose those that he considers most important, regardless of their apparent ordering, as long as the total doesn't exceed the current velocity (2.5 story points). For example, if the sales report is the most important story for him, he can choose it without choosing "sell soft drinks":

User story	***Story points***
Generate monthly sales report in text	2
Total	2

How can we generate the sales report if the system can't sell yet ("sell soft drinks" is not done yet)? Yes, we can. We can hard code some sales data into the system so that the sales report can be generated.

What if he would like to do:

User story	***Story points***
Sell soft drinks (single type)	2
Generate monthly sales report in text	2

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

User story	*Story points*
Total	4

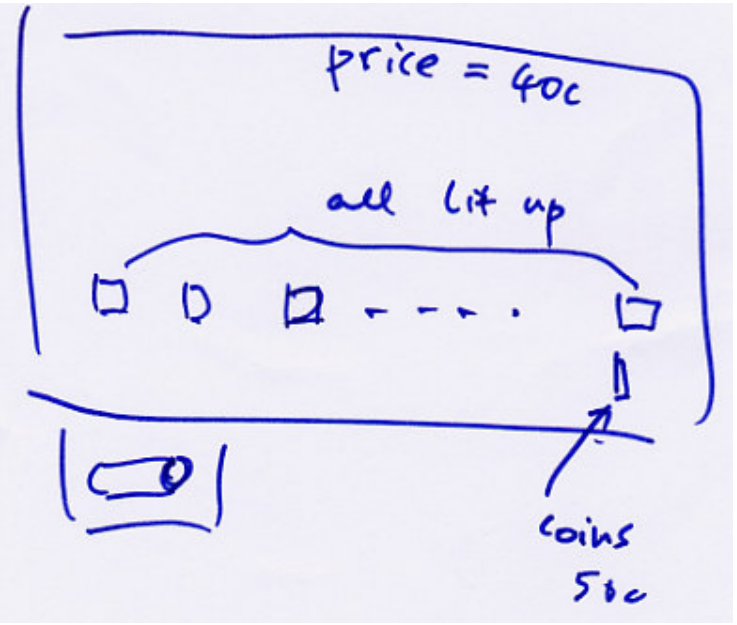
The problem here is that the total is 4 story points and exceeds 2.5. In that case, we can try to simplify by splitting up the stories. For example, we can skip the function of returning changes to the user when selling soft drinks. This may reduce the story points by 1.5. If the customer agrees, we can split "sell soft drinks" into two and delay one of them to the future:

User story	*Story points*
Sell soft drinks (single type; no changes)	0.5
~~Return changes~~	~~1.5~~
Generate monthly sales report in text	2
Total	2.5

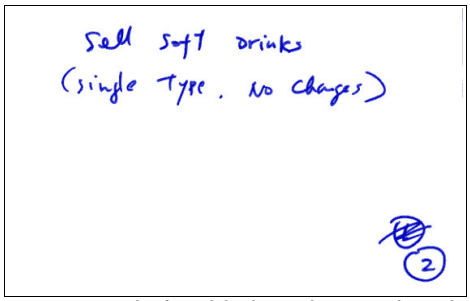
When will we implement "return changes"? It is just a regular story card. If the customer chooses it in the next iteration, we will do it in the next iteration. If he chooses it in the iteration after the next one, we will do it then. It is totally up to him.

For each user story selected for this iteration, unlike release planning where we only wanted just a few important details of a story, now we will ask the customer for all the details required for implementation. For example, for "sell soft drinks", we may draw some sketches on a white board showing the user interaction while providing him with suggestions:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

This is the vending machine...	
The user inserts coins there...	
Suppose that he has inserted 50c and the price is 40c, all the buttons will be lit up. Remember that we only deal with a single type of soft drinks, so there is only one price...	
Then the user presses any button, a soft drink will be put into the dispenser...	
...	

After getting enough details, we may find that "sell soft drinks (single type; no changes)" actually should be worth 2 story points instead of 0.5. This is allowed. Then, we will update the story card:



and let the customer take out some cards for this iteration so that the total is <= 2.5. If it is the opposite, i.e., we find that it is only worth less than 0.5 story points, then we will update the story card and let the customer add more cards for this iteration.

The process above of choosing what stories to do in the upcoming iteration is called "iteration planning".

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

User story is the beginning of communication, not the end of it

Suppose that after getting enough details from the customer, we are ready to start the implementation. Note that we don't have to write down all the details provided by the customer. Why not? If in the future you have a question about this user story, and if the customer is standing in front of you, would you directly ask him or go find the user story description in the requirement specification? Of course you would ask the customer, because in most of the time this live customer can provide requirements that are more accurate, complete and updated than a piece of dead requirement specification. In particular, whenever you have implemented a user story, try to let him see it or actually test drive it, because the more he knows about the system being developed, the more accurate and complete the requirements he can provide in the future. What if you forget about the details in the future? You can ask him again.

Remember, user story is the beginning of communication, not the end of it.

Therefore, usually we don't write down the events in a user story and only write the name of the user story, e.g., writing "Sell soft drinks" is enough.

Going through the iteration

After the customer choosing the stories, in the coming two weeks, we will implement them one by one. For each one we will perform design, coding, testing and etc. After implementing each user story, we will demonstrate the system according to the user story to the customer to see if this is what he wants.

Within the two weeks, if we finish these user stories early, we will ask the customer for more user stories. On the other hand, if we can't finish all these user stories, we will also let the customer know our actual progress.

References

- http://c2.com/cgi/wiki?UserStory.
- http://c2.com/cgi/wiki?UseCase.
- http://www.xprogramming.com/xpmag/whatisxp.htm.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- Kent Beck, Martin Fowler, Planning Extreme Programming, Addison-Wesley, 2000.
- Kent Beck, Extreme Programming Explained, Addison-Wesley, 2000.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.
- Martin Fowler, Kendall Scott, UML Distilled, Addison-Wesley, 1999.
- Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000.
- Jim Highsmith, Agile Project Management: Creating Innovative Products, Pearson Education, 2004.
- Craig Larman, Agile and Iterative Development: A Manager's Guide, Addison-Wesley, 2003.
- Brooks, F.P., The Mythical Man Month: Essays on Software Engineering Anniversary Edition. Addison-Wesley, 1995.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. This application is about courses. A course has a course code, a title, a fee and a number of sessions. A session specifies a date, a starting hour and ending hour. You have also created the following classes:

```
class Course {
   String courseId;
   String title;
   int fee;
   Session sessions[];
}
class Session {
   Date date;
   int startHour;
   int endHour;
}
```

   The DBA (database administator) has created the tables below:

```
create table Courses (
   courseId varchar(20) primary key,
   title varchar(20) not null,
   fee int not null
);
create table Sessions (
   courseId varchar(20),
   sessionId int,
   sessionDate date not null,
   startHour int not null,
   endHour int not null,
   primary key(courseId, sessionId)
);
```

   Your tasks are:

   1. Create an interface for accessing the courses while hiding the database.

   2. Create a class to implement that interface and show how to implement its method for adding a course to the database.

   3. Implement its method for enumerating all courses whose total duration is greater

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

than a specified number of hours.

4. Identify which layers they belong to.

2. In the application above, some courses are now included in the so called "Training Scheme for the employed". For each such course, the students will get an certain reimbursement after completing the course successfully. The discount is a constant for each course but may vary from one course to another. You know it is good to keep the Course class slim, so you decide to create a new class. So you have written the code below:

```
interface TrainingSchemeForEmployedRegistry {
   boolean isInScheme(String courseId);
   int getDiscount(String courseId);
   void addToScheme(String courseId, int discount);
}
```

To store this information in the database, you ask the DBA. He says that it is most efficient to just add a discount field to the Courses table. If that field is NULL, it means the course is not in the scheme:

```
create table Courses (
   courseId varchar(20) primary key,
   title varchar(20) not null,
   fee int not null,
   discount int
);
```

Your tasks are:

1. Create a class to implement TrainingSchemeForEmployedRegistry and show how to implement its method for the isInScheme and addToScheme methods.

2. Determine if you need to modify your code done in the previous question. If so, where?

3. Determine what happens to its scheme status if a course is deleted?

3. This program below implements a game called "MasterMind". The game is played like this: the computer "comes up with" a secret code such as "RGBY" in which "R" means red, "G" means green, "B" means blue, "Y" means yellow, "P" means pink, "C" means cyan. The task of the player is to try to find out this secret code. Every turn the player can input a code also consisting of four colors such as "RBPY". In this case, he has got "R" correct because the secret code also contains "R" in the first position. The same is true for "Y". In contrast, the secret code contains "B", but the position is not correct. In response to the guess, the computer will tell the player that he has got two pegs in correct color and position (but will not tell him that they are "R" and "Y") and that he has got one peg in the correct color but incorrect position (but will not tell him that it's "B"). The player has at most 12 turns. If he can find out the secret code within 12 turns, he wins. Otherwise he loses.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Your tasks are:

1. Point out and remove the problems in the code below.
2. Divide your revised code into appropriate layers.

```
class MasterMind {
   String secret = "RGBY";
   static public void main(String args[]) {
      new MasterMind();
   }
   MasterMind() {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      for (i = 0; i < 12;) { //can guess at most 12 times
         String currentGuess = br.readLine();
         String currentSecret = secret;
         int m = 0; //how many pegs are right in color and position.
         int n = 0; //how many pegs are right in color but wrong position.
         //valid the colors and find those in right color and position.
         for (j = 0; j < currentGuess.length();) {
            //must sure each peg is one of: Red, Yellow, Pink,
            //Green, Blue or Cyan.
            if ("RYPGBC".indexOf(currentGuess.charAt(j))==-1) {
               System.out.println("Invalid color!");
               break;
            }
            if (currentGuess.charAt(j)== currentSecret.charAt(j)) {
               //right color and position.
               m++;
               //delete the peg.
               currentGuess=
                  currentGuess.substring(0, j)+
                  currentGuess.substring(j+1);
               currentSecret=
                  currentSecret.substring(0, j)+
                  currentSecret.substring(j+1);
            } else {
               j++;
            }
         }
         //see how many pegs are in right color but wrong position.
         for (j = 0; j < currentGuess.length();) {
            //is it in right color regardless of the position?
            k = currentSecret.indexOf(guess.charAt(j));
            if (k!=-1)
               n++;
               //delete the peg.
               currentGuess=
                  currentGuess.substring(0, j)+
                  currentGuess.substring(j+1);
               currentSecret=
                  currentSecret.substring(0, k)+
                  currentSecret.substring(k+1);
            } else {
               j++;
            }
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            System.out.println(m+" are right in color and position");
            System.out.println(n+" are right in color but wrong position");
            if (m==4) { //all found?
               System.out.println("You won!");
               return;
            }
            i++;
         }
         System.out.println("You lost!");
      }
   }
```

4. This is a web-based application concerned with food orders. A user can enter an order id in a form and click submit. Then the application will display the customer name of the order and the order items (food id and quantity) in the order. The servlet doing that is shown below.

```
public class ShowOrderServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<HTML><TITLE>Order details</TITLE><BODY>");
      String orderId = request.getParameter("orderId");
      String customerId;
      Connection dbConn = ...;
      PreparedStatement st =
         dbConn.prepareStatement("select * from Orders where orderId=?");
      try {
         st.setString(1, orderId);
         ResultSet rs = st.executeQuery();
         try {
            if (!rs.next()) {
               out.println("Error: Order not found!");
               return;
            }
            customerId = rs.getString("customerId");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
      //find and show the customer's name from his id.
      PreparedStatement st = dbConn.prepareStatement(
            "select * from Customers where customerId=?");
      try {
         st.setString(1, customerId);
         ResultSet rs = st.executeQuery();
         try {
            if (!rs.next()) {
               out.println("Error: Customer not found!");
               return;
            }
            out.println("Customer: "+rs.getString("name"));
         } finally {
            rs.close();
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      } finally {
         st.close();
      }
      //find and show the order items.
      st = dbConn.prepareStatement(
            "select * from OrderItems where orderId=?");
      try {
         st.setString(1, orderId);
         ResultSet rs = st.executeQuery();
         try {
            out.println("<TABLE>");
            while (rs.next()) {
               out.println("<TR>");
               out.println("<TD>");
               out.println(rs.getString(3)); //food id
               out.println("</TD>");
               out.println("<TD>");
               out.println(rs.getInt(4)+""); //quantity
               out.println("</TD>");
               out.println("</TD>");
               out.println("</TR>");
            }
            out.println("</TABLE>");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
      out.println("</BODY></HTML>");
 }
}
```

Your tasks are:

1. Point out and remove the problems in the code below.

2. Divide your revised code into appropriate layers.

5. Come up with 10 user stories and estimate the story points of each. Assume that you have 2 developers and that you estimate one story point takes one developer 4 days to do, estimate how many story points can be done by the team in the first iteration. If you were the customer, which user stories would you like to be implemented in the first iteration? Assume that in the first iteration the team finished just 90% of the story points planned. What is the current velocity? According to this velocity, estimate the project duration. If the customer insists that you deliver the system by 80% of the duration you propose, what should you do? How many story points can be done in the second iteration?

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. This application is about courses. A course has a course code, a title, a fee and a number of sessions. A session specifies a date, a starting hour and ending hour. You have also created the following classes:

```
class Course {
   String courseId;
   String title;
   int fee;
   Session sessions[];
}
class Session {
   Date date;
   int startHour;
   int endHour;
}
```

   The DBA (database administator) has created the tables below:

```
create table Courses (
   courseId varchar(20) primary key,
   title varchar(20) not null,
   fee int not null
);
create table Sessions (
   courseId varchar(20),
   sessionId int,
   sessionDate date not null,
   startHour int not null,
   endHour int not null,
   primary key(courseId, sessionId)
);
```

   Your tasks are:

   1. Create an interface for accessing the courses while hiding the database.

```
interface Courses {
   void addCourse(Course course);
   void deleteCourse(String courseId);
   void updateCourse(Course course);
   CourseIterator getAllCoursesById();
}
```

   2. Create a class to implement that interface and show how to implement its method for adding a course to the database.

```
class CoursesInDB implements Courses {
   void addCourse(Course course) {
      PreparedStatement st =
         dbConn.prepareStatement("insert into from Courses values(?,?,?)");
      try {
         st.setString(1, course.getId());
         st.setString(2, course.getTitle());
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            st.setInt(3, course.getFee());
            st.executeUpdate();
         } finally {
            st.close();
         }
         st = dbConn.prepareStatement(
               "insert into from Sessions values(?,?,?,?,?)");
         try {
            for (int i = 0; i < course.getNoSessions(); i++) {
               Session session = course.getSessionAt(i);
               st.setString(1, course.getId());
               st.setInt(2, i);
               st.setDate(3, session.getDate());
               st.setInt(4, session.getStartHour());
               st.setInt(5, session.getEndHour());
               st.executeUpdate();
            }
         } finally {
            st.close();
         }
      }
   }
```

3. Implement its method for enumerating all courses whose total duration is greater than a specified number of hours.

Update the interface first:

```
interface Courses {
   ...
   CourseIterator getCoursesLongerThan(int duration);
}
```

You may implement the selection using SQL. It is more efficient because only those courses meeting the condition are transferred. But it is useless for another storage mechanism other than a DB.

```
class CoursesInDB {
   ...
   CourseIterator getCoursesLongerThan(int duration) {
      PreparedStatement st =
         dbConn.prepareStatement(
            "select * from Courses where "+
            "(select sum(endHour-startHour) from Sessions "+
            "where Courses.courseId=Sessions.courseId)>? "+
            "order by courseId");
      st.setInt(1, duration);
      return new CourseResultSetIterator(st);
   }
}
class CourseResultSetIterator implements CourseIterator {
   PreparedStatement stToLoadCourses;
   ResultSet rsOfCourses;
   PreparedStatement stToLoadSessions;
   CourseResultSetIterator(
      Connection dbConn, PreparedStatement stToLoadCourses) {
      this.stToLoadCourses = stToLoadCourses;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      this.rsOfCourses = stToLoadCourses .executeQuery();
      this.stToLoadSessions = dbConn.prepareStatement(
            "select * from Sessions where courseId=? order by sessionId");
   }
   boolean next() {
      return rsOfCourses.next();
   }
   Course getCourse() {
      Course course = new Course(
         rsOfCourses.getString(1),
         rsOfCourses.getString(2),
         rsOfCourses.getInt(3));
      loadSessionsFromDB(course);
      return course;
   }
   void loadSessionsFromDB(Course course) {
      stToLoadSessions.setString(1, course.getId());
      ResultSet rsOfSessions=stToLoadSessions.executeQuery();
      try {
         while (rsOfSessions.next()) {
            Session session = new Session(
               rsOfSessions.getDate(3),
               rsOfSessions.getInt(4),
               rsOfSessions.getInt(5));
            course.addSession(session);
         }
      } finally {
         rsOfSessions.close();
      }
   }
   void finalize() {
      stToLoadSessions.close();
      rsOfCourses.close();
      stToLoadCourses.close();
   }
}
```

You may also implement the selection using domain logic in Courses. Courses will become an abstract class instead of an interface. This method is less efficient because all the courses are transferred. But it can be used with any other storage mechanisms.

```
class Course {
   ...
   int getDuration() {
      int duration = 0;
      for (int i = 0; i < getNoSessions(); i++) {
         duration += getSessionAt(i).getDuration();
      }
      return duration;
   }
}
class Session {
   ...
   int getDuration() {
      return endHour-startHour;
   }
}
abstract class Courses {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   ...
   CourseIterator getCoursesLongerThan(final int duration) {
      final CourseIterator allCourses = getAllCoursesById();
      return new CourseIterator() {
         Course currentCourse;
         boolean next() {
            while (allCourses.next()) {
               currentCourse = allCourses.getCourse();
               if (currentCourse.getDuration()>duration) {
                  return true;
               }
            }
            return false;
         }
         Course getCourse() {
            return currentCourse;
         }
      };
   }
}
```

4. Identify which layers they belong to.

Domain: Courses, Course, Session, CourseIterator.

DB access: CoursesInDB, CourseResultSetIterator.

2. In the application above, some courses are now included in the so called "Training Scheme for the employed". For each such course, the students will get an certain reimbursement after completing the course successfully. The discount is a constant for each course but may vary from one course to another. You know it is good to keep the Course class slim, so you decide to create a new class. So you have written the code below:

```
interface TrainingSchemeForEmployedRegistry {
   boolean isInScheme(String courseId);
   int getDiscount(String courseId);
   void addToScheme(String courseId, int discount);
}
```

To store this information in the database, you ask the DBA. He says that it is most efficient to just add a discount field to the Courses table. If that field is NULL, it means the course is not in the scheme:

```
create table Courses (
   courseId varchar(20) primary key,
   title varchar(20) not null,
   fee int not null,
   discount int
);
```

Your tasks are:

1. Create a class to implement TrainingSchemeForEmployedRegistry and show how to

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

implement its method for the isInScheme and addToScheme methods.

```
class TrainingSchemeForEmployedRegistryInDB
   implements TrainingSchemeForEmployedRegistry {
   boolean isInScheme(String courseId) {
      PreparedStatement st =
         dbConn.prepareStatement(
            "select discount from Courses where courseId=?");
      try {
         st.setString(1, courseId);
         ResultSet rs=st.executeQuery();
         rs.next();
         return rs.getObject(1)!=null;
      } finally {
         st.close();
      }
   }
   void addToScheme(String courseId, int discount) {
      PreparedStatement st =
         dbConn.prepareStatement(
            "update Courses set discount=? where courseId=?");
      try {
         st.setInt(1, discount);
         st.setString(2, courseId);
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   ...
}
```

2. Determine if you need to modify your code done in the previous question. If so, where?

When adding an course record to the Courses table, there is one more field (discount). It must be NULL to indicate that the course is not in the scheme. Some DBMS may allow you to use the previous code unchanged, while still having this effect. But to be safe than sorry, do it explicitly:

```
class CoursesInDB implements Courses {
   void addCourse(Course course) {
      PreparedStatement st =
         dbConn.prepareStatement(
            "insert into from Courses values(?,?,?,?)");
      try {
         st.setString(1, course.getId());
         st.setString(2, course.getTitle());
         st.setInt(3, course.getFee());
         st.setNull(4, java.sql.Types.INTEGER);
         st.executeUpdate();
      } finally {
         st.close();
      }
      ...
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

3. Determine what happens to its scheme status if a course is deleted?

   The scheme status is also deleted. This is what we want.

3. This program below implements a game called "MasterMind". The game is played like this: the computer "comes up with" a secret code such as "RGBY" in which "R" means red, "G" means green, "B" means blue, "Y" means yellow, "P" means pink, "C" means cyan. The task of the player is to try to find out this secret code. Every turn the player can input a code also consisting of four colors such as "RBPY". In this case, he has got "R" correct because the secret code also contains "R" in the first position. The same is true for "Y". In contrast, the secret code contains "B", but the position is not correct. In response to the guess, the computer will tell the player that he has got two pegs in correct color and position (but will not tell him that they are "R" and "Y") and that he has got one peg in the correct color but incorrect position (but will not tell him that it's "B"). The player has at most 12 turns. If he can find out the secret code within 12 turns, he wins. Otherwise he loses.

   Your tasks are:

   1. Point out and remove the problems in the code below.

   2. Divide your revised code into appropriate layers.

```
class MasterMind {
   String secret = "RGBY";
   static public void main(String args[]) {
      new MasterMind();
   }
   MasterMind() {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      for (int i = 0; i < 12;) { //can guess at most 12 times
         String currentGuess = br.readLine();
         String currentSecret = secret;
         int m = 0; //how many pegs are right in color and position.
         int n = 0; //how many pegs are right in color but wrong position.
         //valid the colors and find those in right color and position.
         for (int j = 0; j < currentGuess.length();) {
            //must sure each peg is one of: Red, Yellow, Pink,
            //Green, Blue or Cyan.
            if ("RYPGBC".indexOf(currentGuess.charAt(j))==-1) {
               System.out.println("Invalid color!");
               break;
            }
            if (currentGuess.charAt(j)== currentSecret.charAt(j)) {
               //right color and position.
               m++;
               //delete the peg.
               currentGuess=
                  currentGuess.substring(0, j)+
                  currentGuess.substring(j+1);
               currentSecret=
                  currentSecret.substring(0, j)+
                  currentSecret.substring(j+1);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            } else {
               j++;
            }
         }
         //see how many pegs are in right color but wrong position.
         for (int j = 0; j < currentGuess.length();) {
            //is it in right color regardless of the position?
            int k = currentSecret.indexOf(currentGuess.charAt(j));
            if (k!=-1) {
               n++;
               //delete the peg.
               currentGuess=
                  currentGuess.substring(0, j)+
                  currentGuess.substring(j+1);
               currentSecret=
                  currentSecret.substring(0, k)+
                  currentSecret.substring(k+1);
            } else {
               j++;
            }
         }
         System.out.println(m+" are right in color and position");
         System.out.println(n+" are right in color but wrong position");
         if (m==4) { //all found?
            System.out.println("You won!");
            return;
         }
         i++;
      }
      System.out.println("You lost!");
   }
}
```

The code mixes model logic with UI. Some code is duplicated. The comments in the code can be removed by making the code as clear as the comments.

We can extract the domain logic into:

```
class MasterMind {
   static final int MAX_NO_GUESSES=12;
   static final int NO_PEGS=4;
   String secret = "RGBY";
   GuessResult makeGuess(String guess) throws InvalidColorException {
      GuessPartialResult partialResult=
         new GuessPartialResult(secret, guess);
      partialResult.validateColors();
      partialResult.findPegsInRightColorAndPos();
      partialResult.findPegsInRightColorIgnorePos();
      return partialResult.makeFinalResult();
   }
}
class InvalidColorException extends Exception {
}
```

GuessPartialResult represents the intermediate state when the pegs are being compared:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class GuessPartialResult {
   String secretLeft;
   String guessLeft;
   int noPegsInRightColorAndPos;
   int noPegsInRightColorButWrongPos;
   GuessPartialResult(String initSecret, String initGuess) {
      secretLeft=initSecret;
      guessLeft=initGuess;
   }
   void validateColors() {
      for (int j = 0; j < guessLeft.length();) {
         assertValidColor(guessLeft.charAt(j));
      }
   }
   void assertValidColor(char color) throws InvalidColorException {
      final String RED="R";
      final String YELLOW="Y";
      final String PINK="P";
      final String GREEN="G";
      final String BLUE="B";
      final String CYAN="C";
      final String VALID_COLORS=RED+YELLOW+PINK+GREEN+BLUE+CYAN;
      if (VALID_COLORS.indexOf(color)==-1) {
         throw new InvalidColorException();
      }
   }
   void findPegsInRightColorAndPos() {
      for (int j = 0; j < guessLeft.length();) {
         if (isPegInRightColorAndPos(j)) {
            noPegsInRightColorAndPos++;
            guessLeft=deletePegAt(guessLeft, j);
            secretLeft=deletePegAt(secretLeft, j);
         } else {
            j++;
         }
      }
   }
   void findPegsInRightColorIgnorePos() {
      for (int j = 0; j < guessLeft.length();) {
         int k = findFirstPegInSecretOfColor(guessLeft.charAt(j));
         if (k!=-1) {
            noPegsInRightColorButWrongPos++;
            guessLeft=deletePegAt(guessLeft, j);
            secretLeft=deletePegAt(secretLeft, k);
         } else {
            j++;
         }
      }
   }
   boolean isPegInRightColorAndPos(int idx) {
      return guessLeft.charAt(idx)== secretLeft.charAt(idx);
   }
   int findFirstPegInSecretOfColor(char color) {
      return secretLeft.indexOf(color);
   }
   String deletePegAt(String pegs, int idx) {
      return pegs.substring(0, idx)+pegs.substring(idx+1);
   }
   GuessResult makeFinalResult() {
      return new GuessResult(
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            noPegsInRightColorAndPos,
            noPegsInRightColorButWrongPos);
      }
}
```

GuessResult represents the final result of a guess:

```
class GuessResult {
   int noPegsInRightColorAndPos;
   int noPegsInRightColorButWrongPos;
   boolean allFound() {
      return noPegsInRightColorAndPos==MasterMind.NO_PEGS;
   }
}
```

The UI will use the domain logic:

```
class MasterMindApp {
   static public void main(String args[]) {
      new MasterMindApp();
   }
   MasterMindApp() {
      MasterMind masterMind = new MasterMind();
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      String guess = br.readLine();
      for (int i = 0; i < MasterMind.MAX_NO_GUESSES;) {
         try {
            GuessResult guessResult = masterMind.makeGuess(guess);
            System.out.println(
               guessResult.getNoPegsInRightColorAndPos()+
               " are right in color and position");
            System.out.println(
               guessResult.getNoPegsInRightColorButWrongPos()+
               " are right in color but wrong position");
            if (guessResult.allFound()) {
               System.out.println("You won!");
               return;
            }
         } catch (InvalidColorException e) {
            System.out.println("Invalid color!");
            break;
         }
         i++;
      }
      System.out.println("You lost!");
   }
}
```

The layers are:

- Domain: MasterMind, GuessPartialResult, GuessResult, InvalidColorException.
- UI: MasterMindApp.

4. This is a web-based application concerned with food orders. A user can enter an order id in

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

a form and click submit. Then the application will display the customer name of the order and the order items (food id and quantity) in the order. The servlet doing that is shown below.

```
public class ShowOrderServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<HTML><TITLE>Order details</TITLE><BODY>");
      String orderId = request.getParameter("orderId");
      String customerId;
      Connection dbConn = ...;
      PreparedStatement st =
         dbConn.prepareStatement("select * from Orders where orderId=?");
      try {
         st.setString(1, orderId);
         ResultSet rs = st.executeQuery();
         try {
            if (!rs.next()) {
               out.println("Error: Order not found!");
               return;
            }
            customerId = rs.getString("customerId");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
      //find and show the customer's name from his id.
      st = dbConn.prepareStatement(
         "select * from Customers where customerId=?");
      try {
         st.setString(1, customerId);
         ResultSet rs = st.executeQuery();
         try {
            if (!rs.next()) {
               out.println("Error: Customer not found!");
               return;
            }
            out.println("Customer: "+rs.getString("name"));
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
      //find and show the order items.
      st = dbConn.prepareStatement(
            "select * from OrderItems where orderId=?");
      try {
         st.setString(1, orderId);
         ResultSet rs = st.executeQuery();
         try {
            out.println("<TABLE>");
            while (rs.next()) {
               out.println("<TR>");
               out.println("<TD>");
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
               out.println(rs.getString(3)); //food id
               out.println("</TD>");
               out.println("<TD>");
               out.println(rs.getInt(4)+""); //quantity
               out.println("</TD>");
               out.println("</TD>");
               out.println("</TR>");
            }
            out.println("</TABLE>");
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
      out.println("</BODY></HTML>");
   }
}
```

Your tasks are:

1. Point out and remove the problems in the code below.

2. Divide your revised code into appropriate layers.

The code mixes model logic with database and UI. Remember that servlet is UI. We can extract the domain code into:

```
class Order {
   ...
   Customer customer;
   OrderItem orderItems[];
}
class OrderItem {
   ...
   String foodId;
   int quantity;
}
class Customer {
   ...
}
interface Orders {
   ...
   Order getOrder(String orderId);
}
interface Customers {
   ...
   Customer getCustomer(String customerId);
}
```

We can extract the database access code into:

```
class OrdersInDB implements Orders {
   Order getOrder(String orderId) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      PreparedStatement st =
         dbConn.prepareStatement("select * from Orders where orderId=?");
      try {
         st.setString(1, orderId);
         ResultSet rs = st.executeQuery();
         try {
            if (!rs.next()) {
               throw new OrdersException();
            }
            String customerId = rs.getString("customerId");
            CustomersInDB customers = new CustomersInDB();
            Customer customer = customers.getCustomer(customerId);
            Order order = new Order(...,customer, ...);
            getOrderItemsFromDB(order);
            return order;
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
   void getOrderItemsFromDB(Order order) {
      PreparedStatement st = dbConn.prepareStatement(
         "select * from OrderItems where orderId=?");
      try {
         st.setString(1, order.getId());
         ResultSet rs = st.executeQuery();
         try {
            while (rs.next()) {
               order.addOrderItem(new OrderItem(
                  ...,
                  rs.getString("foodId"),
                  rs.getInt("quantity"),
                  ...));
            }
         } finally {
            rs.close();
         }
      } finally {
         st.close();
      }
   }
}
class CustomersInDB implements Customers {
   Customer getCustomer(String customerId) {
      PreparedStatement st = dbConn.prepareStatement(
            "select * from Customers where customerId=?");
      try {
         st.setString(1, customerId);
         ResultSet rs = st.executeQuery();
         try {
            if (!rs.next()) {
               throw new CustomersException();
            }
            return new Customer(..., rs.getString("name"), ...);
         } finally {
            rs.close();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            }
         } finally {
            st.close();
         }
      }
}
```

The UI will use the domain logic (but not the DB) as:

```
public class ShowOrderServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      Orders orders =
         (Orders)getServletContext().getAttribute("Orders");
      out.println("<HTML><TITLE>Order details</TITLE><BODY>");
      String orderId = request.getParameter("orderId");
      try {
         Order order = orders.getOrder(orderId);
      } catch (OrdersException e) {
         out.println("Error: Order not found!");
         return;
      } catch (CustomersException e) {
         out.println("Error: Customer not found!");
         return;
      }
      out.println("Customer: "+order.getCustomer().getName());
      showOrderItems(out, order);
      out.println("</BODY></HTML>");
   }
   void showOrderItems(PrintWriter out, Order order) {
      out.println("<TABLE>");
      for (int i = 0; i < order.getNoOrderItems(); i++) {
         out.println("<TR>");
         out.println("<TD>");
         out.println(order.getOrderItemAt(i).getFoodId());
         out.println("</TD>");
         out.println("<TD>");
         out.println(order.getOrderItemAt(i).getQuantity()+"");
         out.println("</TD>");
         out.println("</TD>");
         out.println("</TR>");
      }
      out.println("</TABLE>");
 }
}
```

The layers are:

- Domain: Order, OrderItem, Orders, Customer, Customers, OrdersException, CustomersException.
- DB access: OrdersInDB, CustomersInDB.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- UI: ShowOrderServlet.

5. Come up with 10 user stories and estimate the story points of each. Assume that you have 2 developers and that you estimate one story point takes one developer 4 days to do, estimate how many story points can be done by the team in the first iteration. If you were the customer, which user stories would you like to be implemented in the first iteration? Assume that in the first iteration the team finished just 90% of the story points planned. What is the current velocity? According to this velocity, estimate the project duration. If the customer insists that you deliver the system by 80% of the duration you propose, what should you do? How many story points can be done in the second iteration?

   Assume the 10 user stories are:

Name	***Story points***
C1	2
C2	3
C3	2
C4	1
C5	4
C6	5
C7	3
C8	2
C9	4
C10	1

   The total is 25 story points. As one story point needs 4 days and we have two developers, the team should be able to do 10/4x2=5 story points in the first iteration.

   If we were the customer, we would pick C1, C3 and C4. The total is 5 story points (<= 5).

   If in the first iteration we only did 90% of the story points, i.e., 4.5 points. So the current velocity is 4.5 points. According to this velocity, as there are 25 story points, we will need 25/4.5=5.6 iterations. We cannot have a partial iteration, so we will count it as 6 iterations. So the project duration is 6x10=60 days.

   If the customer needs to get the system in just 80% of the proposed duration, i.e., 48 days, we may let him choose which user stories to include as long as the total does not exceed (48/10)x4.5=21.6 story points. It means he needs to take out at least 3.4 story points. For example, he may take out C5 (4 points); he may take out C9 (4 points); he

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

may take out C7 and C10 (3+1 points). It is totally up to him.

Should he insist to include all the user stories without reducing any functionality (this is extremely rare!), then we need "n" developers such that (48/10)x4.5x(n/2)=25, i.e., "n" should be 2.3. As we cannot have 0.3 of a developer, we need to hire one additional developer and the customer will have to pay for the salary.

For the second iteration, the customer can only choose user stories not exceeding 4.5 story points.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 9

OO Design with CRC Cards

A sample user story

Suppose that the customer of this project is a manufacturer of vending machines for soft drinks. They request us to develop software to control their vending machines. Suppose that the customer describes the following user story:

Name: Sell soft drink

Events:

1. A user inserts some money.
2. The vending machine displays how much money he has inserted so far.
3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.
4. The user presses a certain lit-up button.
5. The vending machine sells a can of soft drink to him.
6. The vending machine returns the changes to him.

In order to implement this user story, what classes do we need? Let's implement each step in the user story, as shown below.

Design for a user story

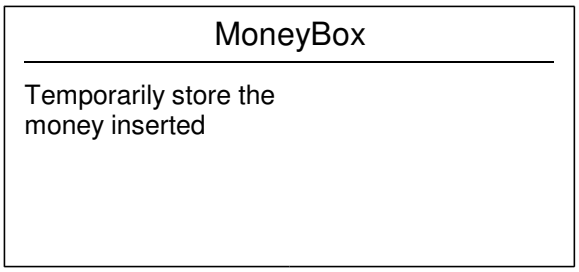
In order to implement steps 1 and 2 in the user story, we need to find out the behaviors that the system needs to perform:

1. A user inserts some money.
    - Behavior 1: Note that someone has inserted some money.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

2. The vending machine displays how much money he has inserted so far.
    - Behavior 2: Find out how much money the user has inserted so far.
    - Behavior 3: Display it.
3. ...
4. ...
5. ...
6. ...

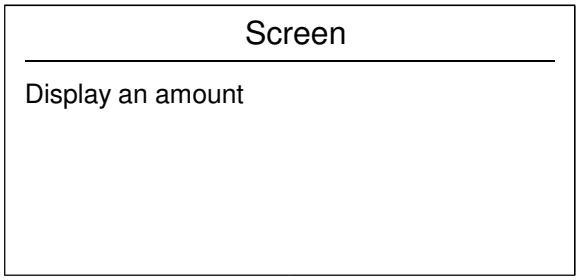
Who will perform these behaviors? Suppose that the money box is represented by a MoneyBox class, that it is alerted whenever someone inserts some money (behavior 1) and that it keeps track of how much money the user has inserted (behavior 2) (we write the following on a 4 inch by 6 inch index card by hand):



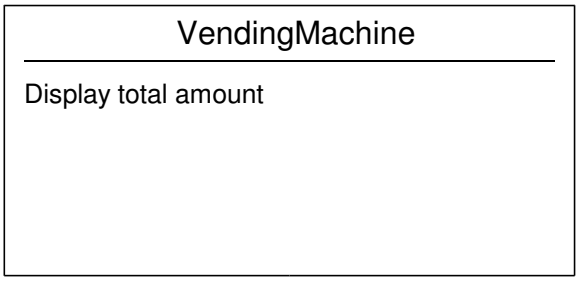
The phrase "temporarily store the money inserted" written on the card is a "responsibility". In order to implement this responsibility, MoneyBox may need quite a few attributes and methods such as a balance attribute plus methods like putMoney, takeMoney, returnMoney, getAmount and etc. At the moment, we hope to think on a high level, therefore we think in terms of responsibilities instead of attributes and methods.

OK, let run it again. The user inserts some money. MoneyBox is alerted. It knows how much money has been inserted. However, who is going to display the total amount (behavior 3)? Suppose that there is a Screen class (make a new card):

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



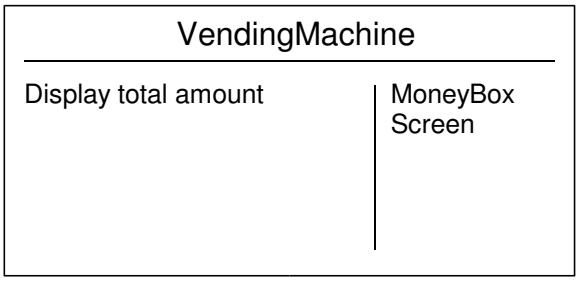
Who is responsible for sending the amount from MoneyBox to Screen? (put one hand on the MoneyBox card and the other on the Screen card) How about creating a VendingMachine class (make a new card):



A user inserts some money. MoneyBox notifies VendingMachine (move MoneyBox to VendingMachine). VendingMachine gets the total amount from MoneyBox (move VendingMachine to MoneyBox), then asks Screen display the total amount (move VendingMachine to Screen).

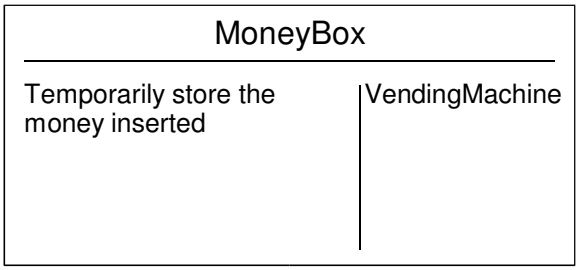
Suppose that we hope to emphasize that VendingMachine needs to work with MoneyBox and Screen, we can update the VendingMachine card as:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



The words "MoneyBox" and "Screen" on the right side of the card state: VendingMachine needs to collaborate with MoneyBox and Screen, i.e., MoneyBox and Screen are both "collaborators" of VendingMachine.

Because MoneyBox needs to notify VendingMachine whenever someone inserts some money, VendingMachine is also a collaborator of MoneyBox. Therefore, if we think there is a need to emphasize this, we can change MoneyBox to:



Here we assume that there is no such a need.

Because on these cards we write the name of a class, its responsibilities and the collaboration relationships, we call these cards "CRC cards". The letters "CRC" stand for Class, Responsibility and Collaboration respectively.

Now, let's find out the behaviors needed to be performed by the system in order to implement step 3 in the user story:

1. A user inserts some money.

2. The vending machine displays how much money he has inserted so far.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.
    - Behavior 1: Note that someone has inserted some money.
    - Behavior 2: Find out how much money the user has inserted so far.
    - Behavior 3: Determine the price of each type of soft drink.
    - Behavior 4: Determine which button corresponds to which type of soft drink.
    - Behavior 5: If the total amount exceeds the price of a type of soft drink, light up the corresponding button.
4. ...
5. ...
6. ...

Who is going to perform these behaviors? Suppose that MoneyBox notifies VendingMachine (move MoneyBox to VendingMachine) that someone has inserted some money (behavior 1). VendingMachine gets the total amount from MoneyBox (behavior 2), then find out the price of each type of soft drink (behavior 3). From where does it get the prices? It may be a PriceList object (make a new card):

After VendingMachine gets the prices (move VendingMachine to PriceList), it needs to determine to which button each type of soft drink corresponds. How about creating a SoftDrink (make a new card) object to represent a type of soft drink and let it record to which button it corresponds (behavior 4):

As we have SoftDrink now, why not let it record its own price? Therefore we no longer need

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

SoftDrink
Know which button it corresponds to

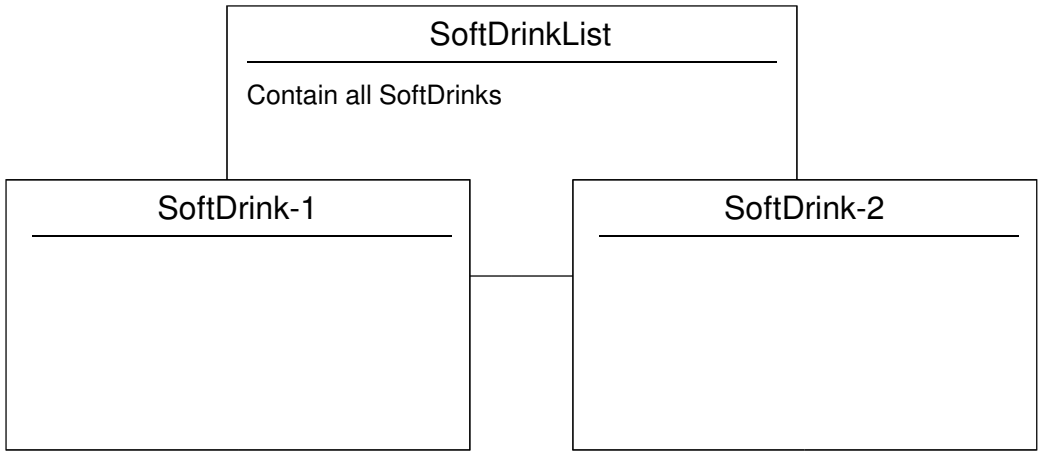
PriceList (tear it up and throw it into the trash bin). However, we still need an object to contain all the SoftDrinks. Let's call it SoftDrinkList (make a new card):

SoftDrinkList
Contain all SoftDrinks

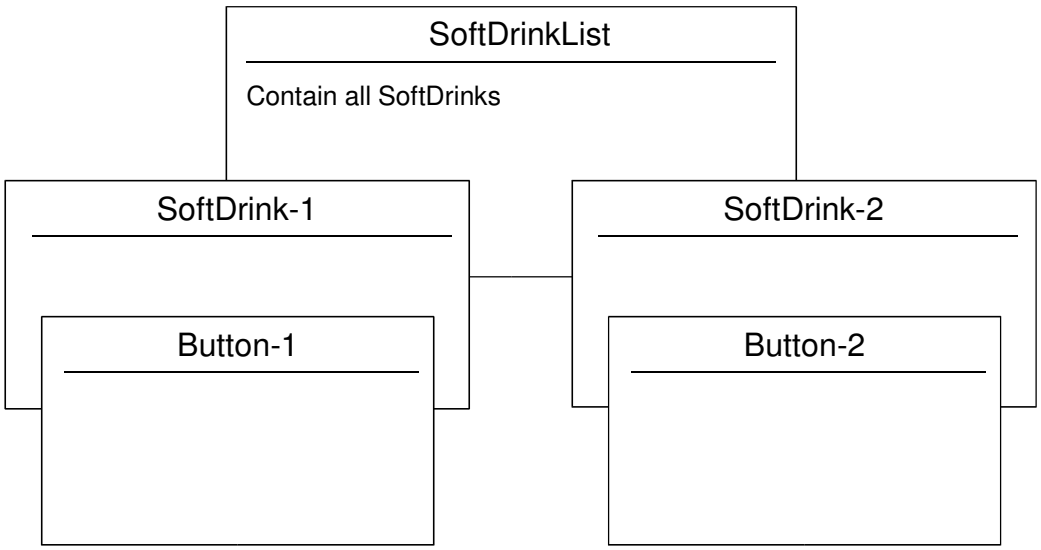
SoftDrink
Know which button it corresponds to Know its own price

OK, let run it again. MoneyBox notifies VendingMachine (move MoneyBox to VendingMachine) that someone has inserted some money (behavior 1). VendingMachine gets the total amount from MoneyBox (behavior 2), then find out all types of soft drinks (move VendingMachine to SoftDrinkList). Suppose that there are two types of soft drinks (rename SoftDrink as SoftDrink-1, make another card named SoftDrink-2, then put them under SoftDrinkList):

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

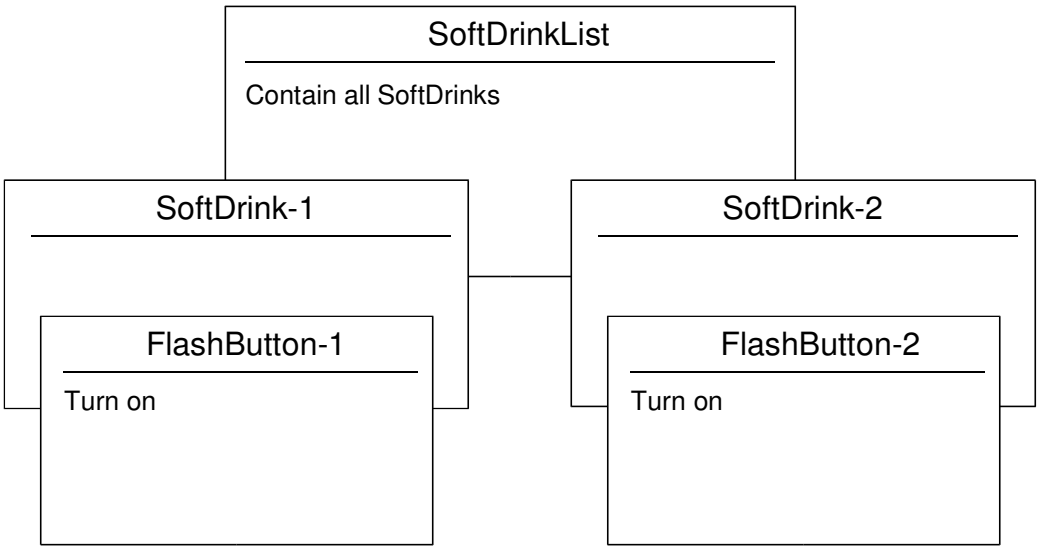


VendingMachine asks SoftDrink-1 for its price (move VendingMachine to SoftDrink-1). Suppose that the total amount is not enough. VendingMachine asks SoftDrink-2 for its price (move VendingMachine to SoftDrink-2). Suppose that the total amount is enough. VendingMachine asks SoftDrink-2 to which button it corresponds (move VendingMachine to SoftDrink-2). Suppose that it corresponds to a certain Button (make a card for the button and put it along with SoftDrink-2. Also make one for SoftDrink-1):

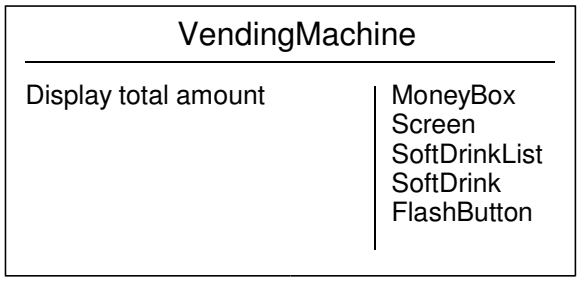


Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Finally, VendingMachine asks Button-2 to light itself up. As a Button is responsible for lighting itself up, how about renaming it to FlashButton:



OK, we have finished the behaviors needed by step 3. If required, we may update VendingMachine like:



Now, let's find out the behaviors needed to be performed by the system in order to implement steps 4 to 6 in the user story:

1. A user inserts some money.

2. The vending machine displays how much money he has inserted so far.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.

4. The user presses a certain lit-up button.

   - Behavior 1: Note that someone has pressed a button.

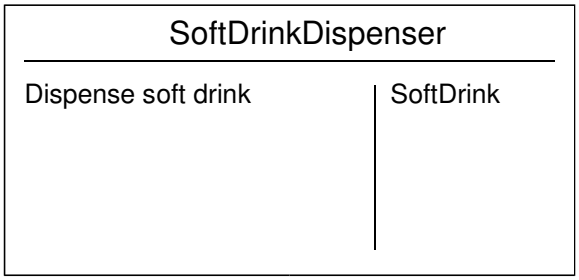
5. The vending machine sells a can of soft drink to him.

   - Behavior 2: Check to make sure the button is lit-up.
   - Behavior 3: Find out the type of soft drink corresponding to that button.
   - Behavior 4: Dispense a can of soft drink of this type.
   - Behavior 5: Turn off the button.

6. The vending machine returns the changes to him.

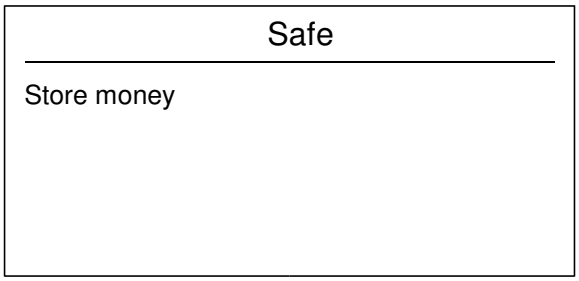
   - Behavior 6: Return the changes.

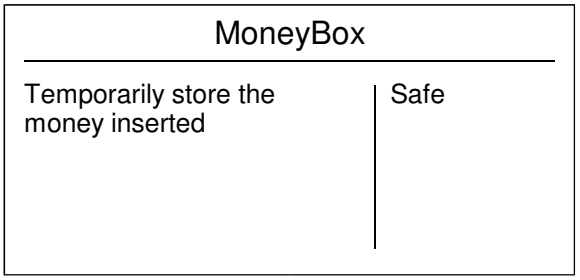
The user presses a button e.g. FlashButton-2 (put left hand on FlashButton-2). FlashButton-2 checks if it is lit up. If yes, it notifies VendingMachine (put right hand on VendingMachine. Move FlashButton-2 to VendingMachine). VendingMachine asks SoftDrink-1 if this is its button (move VendingMachine to SoftDrink-1). It says no. VendingMachine asks SoftDrink-2 if this is its button (move VendingMachine to SoftDrink-2). It says yes. Therefore, VendingMachine dispenses a can of soft drink belong to type SoftDrink-2. Who is responsible for this action? It may be a SoftDrinkDispenser (make a new card):



Then VendingMachine turns off FlashButton-2 (move VendingMachine to FlashButton-2), asks SoftDrink-2 for its price (move it to SoftDrink-2), asks MoneyBox for the total amount the user has inserted (move it to MoneyBox), tells MoneyBox to put the money into the safe (make a new card to represent the safe and move MoneyBox to it):

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org





Then VendingMachine calculates the changes and tells the Safe to return the changes (move VendingMachine to Safe).

Typical usage of CRC cards

Why use CRC cards instead of Word or the advanced CASE tools?

1. The space on a card for writing is very limited, prohibiting us from writing too many responsibilities. If there are many responsibilities (e.g., more than 4), check if we can write in a more abstract way, merging several responsibilities into one. If no, consider creating a new class to share its responsibilities.

2. CRC cards are mainly used for exploring or discussing various design alternatives. A design doesn't work? Simply throw the cards away. In addition, once the design is done, we can throw all the cards away. Their purpose is not for documentation.

3. We can move the CRC cards back and forth or put the related cards together.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CRC cards are mainly used for discussing and establishing a quick design. We should not spend a lot of time in this activity, nor should we expect the design created to be working perfectly (in most of the time it does not entirely work). When we code, we still need to continuously adjust the design. Only when we design and code together can we efficiently create good designs.

There is no formal definition for the usage of CRC cards. We don't need to care too much what to write on each card, or whether something is missing on the cards. For some people, maybe writing the class names is already enough. Some people may consider that it helps them think better if the responsibilities are also written. Some other people may hope to also write the collaborators. We don't need to care too much what it means to put one card on another, or what it means to move one card to another. As long as we keeping talking while we manipulate the cards, the meaning of the actions will become very clear.

References

- http://c2.com/doc/oopsla89/paper.html.
- http://c2.com/cgi/wiki?CrcCards.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

1. Use CRC cards to design for the user story: A user inserts his ATM card into an ATM. The system asks him for the password. He inputs the password. The system asks him to choose an operation. He chooses withdrawal. The system asks how much he would like to withdraw. He inputs the amount. The system gives him the money. The system asks him to choose an operation. He chooses quit. The system ejects the ATM card.

   You must use real cards to design. You must not use Word or some other software instead. Try moving the cards in the process to help you think.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 10

Acceptance Test

Have we implemented a user story correctly

Suppose that the customer of this project is a conference and exhibition organizer. They request us to develop a software system to manage the participants of a particular conference. Suppose that they request us to implement four user stories in this iteration. One of the user stories is shown below:

Name: Import participants

Events:

1. A user asks the system to read the information about a batch of participants from a text file. The information about a participant include ID, password, name, address and email.

2. The system saves the information about the participants. In the future a user can input the ID of a participant, the system will retrieve the information about that participant.

3. The system sends an email to each participant, telling him his ID and password.

Strictly speaking, the underlined portion in the above user story should not appear here because it describes the internal behavior of the system. What is important for the user is that he can retrieve the information of the participants in the future. However, we allow it here because it should help the users understand.

We start by asking the customer about the details of the user story. For example:

- What is the format of the text file? Suppose that the customer says that every line contains a participant. Its various data items are separated by tabs.

- Must the ID, password, name, address and email be present? Suppose that the customer allows the address to be absent, while the other data items must be present. Otherwise, it will skip the line.

- What should the system do if a participant ID already exists? Suppose that the customer hopes to skip the line.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- So on.

We ask the customer, use CRC cards or some other methods to establish or discuss a quick design, write code and improve the design at the same time. Suppose that in two days we have finished all the code and there is nothing to improve in the design. However, we still need to do an important task: Test our code to see if it is implementing that user story correctly.

How to test

How to test? For example, we can run the following "test case":

Test case 1: Import participants

1. Create the following file:

```
p001   123456     Mary Lam        abc     mary@hotmail.com
p004   888999     John Chan       def     john@yahoo.com
p002   mypasswd   Paul Lei        ghi     paul@excite.com
```

2. Delete all the participants in the system to prevent the case that p001, p002 or p004 already exists.

3. Run the system and let it import the file above into the database.

4. How to check if it has correctly imported the file? We should have a user story to let a user input a participant ID and then have the system display the information of that participant. We can implement that user story first, and then input p001 as the participant ID, check if the system will display the information of p001 (123456, Mary Lam and etc.), then input p002 and then p004 and etc.

5. How to check if it has sent the emails? We can contact Mary, John and Paul and ask them if they have received the emails and whether the contents of the emails are correct.

This kind of test is called "acceptance test" or "functional test". This kind of test only tests the external behaviors of a system, ignoring what modules (classes) are inside the system.

In order to test whether it is really OK to not to specify the address of a participant, we create a new test case:

Test case 2: Import a participant without address

1. Create the following file:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
p001  123456   Mary Lam    abc     mary@hotmail.com
p004  888999   John Chan           john@yahoo.com
p002  mypasswd Paul Lei    ghi     paul@excite.com
```

2. Delete all the participants in the system.
3. Run the system and let it import the file above into the database.
4. Input p004 as the participant ID and check if the system displays the information of p004.
5. Contact John and ask him if he has received the email and whether the contents of the email are correct.

In order to test whether it will insist that all the data items other than address be specified, we create a new test case:

Test case 3: Import participants without other information

1. Create the following file:

```
      123456   Mary Lam    abc     mary@hotmail.com
p004  888999              def     john@yahoo.com
p002           Paul Lei    ghi     paul@excite.com
p005  secret   Mike Chan   jkl
```

2. Delete all the participants in the system.
3. Run the system and let it import the file above into the database.
4. Input an empty string, p002, p004 and p005 as the participant ID respectively and check if the system displays the information of any of them (it should not).
5. Contact Mary, Paul and Mike and ask them if they have received any emails (they should not have).

In order to test whether it will not add a participant that already exists, we create a new test case:

Test case 4: Import a participant with a duplicate ID

1. Create the following file:

```
p001   123456 Mary Lam    abc     mary@hotmail.com
p001   888999 John Chan   def     john@yahoo.com
```

2. Delete all the participants in the system.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

3. Run the system and let it import the file above into the database.
4. Input p001 as the participant ID and check if the system displays the information of Mary instead of John.
5. Contact Mary and ask her if she has received the email.
6. Contact John and ask him if he has received any email (he should not have).

Problems with manual tests

We have created four test cases. They have one thing in common: Each step in the test cases (create text file, run the system, input a participant ID, check if the correct information is displayed, contact the people to check if they have received the emails and etc.) needs to be performed manually. Therefore, it takes a lot of effort and time.

This is a serious problem. When we implement the other user stories, we will have to modify the code or add new code. This may introduce bugs. Therefore, whenever the code is changed we hope to run the above test cases again to see if they still pass. If not, we can start finding the bug immediately (where is the bug? Very likely it is in the code that we just modified or added). However, because it takes too much effort and time to run these test cases, it is impossible for us to do it often.

In order to solve this problem, we hope that the test cases can be run automatically without human intervention. This kind of test is called "automated acceptance test".

Automated acceptance tests

We can write the code below to automatically run the four test cases above, where each method corresponds to one test case:

```
class AcceptTestForImportParticipants {
   static void testImport() {
      ...
   }
   static void testImportWithoutAddress() {
      ...
   }
   static void testImportWithoutOtherInfo() {
      ...
   }
   static void testImportDupId() {
      ...
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
```

Where, testImport will implement test case 1. Before writing testImport, let's see this test case again:

Test case 1: Import participants

1. Create the following file:

```
p001  123456   Mary Lam    abc     mary@hotmail.com
p004  888999   John Chan   def     john@yahoo.com
p002  mypasswd Paul Lei    ghi     paul@excite.com
```

2. Delete all the participants in the system to prevent the case that p001, p002 and p004 already exist.

3. Run the system and let it import the file above into the database.

4. How to check if it has correctly imported the file? We should have a user story to let a user input a participant ID and then have the system display the information of that participant. We can implement that user story first, and then input p001 as the participant ID, check if the system will display the information of p001 (123456, Mary Lam and etc.), then input p002 and then p004 and etc.

5. How to check if it has sent the emails? We can contact Mary, John and Paul and ask them if they have received the emails and whether the contents of the emails are correct.

Now we need to implement each step above. We will use a "Command" to represent each step. To run a step (a command), just call the run method of the command. If it succeeds, run should return true:

```
interface Command {
   boolean run();
}
```

Now, implement the first step (create a text file):

```
class AcceptTestForImportParticipants {
   static void testImport() {
      Command commands[] = {
         new CreateImportFileCommand(
            "p001\t123456\tMary Lam\tabc\tmary@hotmail.com\n"+
            "p004\t888999\tJohn Chan\tdef\tjohn@yahoo.com\n"+
            "p002\tmypasswd\tPaul Lei\tghi\tpaul@excite.com"),
         ...
      };
      runCommands(commands);
   }
   static void runCommands(Command commands[]) {
      for (int i = 0; i < commands.length; i++) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            if (!commands[i].run()) {
               System.out.println("Test failed!");
            }
         }
      }
   }
   class CreateImportFileCommand implements Command {
      String fileContent;
      CreateImportFileCommand(String fileContent) {
         this.fileContent = fileContent;
      }
      boolean run() {
         FileWriter fileWriter = new FileWriter("sourceFile.txt");
         try {
            fileWriter.write(fileContent);
            return true;
         } finally {
            fileWriter.close();
         }
      }
   }
```

Implement the second step (Delete all the participants in the system):

```
   class AcceptTestForImportParticipants {
      static void testImport() {
         Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            ...
         };
         runCommands(commands);
      }
   }
```

In order to implement DeleteAllParticipantsCommand, we assume that the system has provided such a function in the Participants class. DeleteAllParticipantsCommand can simply call it:

```
   class DeleteAllParticipantsCommand implements Command {
      boolean run() {
         ConferenceSystem.getInstance().parts.deleteAll();
         return true;
      }
   }
   class ConferenceSystem {
      Participants parts;
      static ConferenceSystem getInstance() {
         ...
      }
   }
   class Participants {
      void deleteAll() {
         ...
      }
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Implement the third step (ask the system to import from the file):

```
class AcceptTestForImportParticipants {
   static void testImport() {
      Command commands[] = {
         new CreateImportFileCommand(...),
         new DeleteAllParticipantsCommand(),
         new ImportParticipantsFromFileCommand(),
         ...
      };
      runCommands(commands);
   }
}
class ImportParticipantsFromFileCommand implements Command {
   boolean run() {
      ConferenceSystem.getInstance().parts.importFromFile("sourceFile.txt");
      return true;
   }
}
class Participants {
   void deleteAll() {
      ...
   }
   void importFromFile(String path) {
      ...
   }
}
```

Implement the fourth step (Retrieve p001 and etc. from the system):

```
class AcceptTestForImportParticipants {
   static void testImport() {
      Command commands[] = {
         new CreateImportFileCommand(...),
         new DeleteAllParticipantsCommand(),
         new ImportParticipantsFromFileCommand(),
         new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
         new CheckParticipantStoredCommand("p002", ...),
         new CheckParticipantStoredCommand("p004", ...),
         ...
      };
      runCommands(commands);
   }
}
class CheckParticipantStoredCommand implements Command {
   Participant part;
   CheckParticipantStoredCommand(String partId, String password, ...) {
      part = new Participant(partId, password, ...);
   }
   boolean run() {
      return ConferenceSystem.getInstance().parts.checkParticipantStored(part);
   }
}
class Participants {
   boolean checkParticipantStored(Participant part) {
      return part.equals(getParticipantById(part.getId()));
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   Participant getParticipantById(String partId) {
      ...
   }
}
class Participant {
   boolean equals(Object obj) {
      ...
   }
}
```

Now we need to implement the fifth step (check if Mary and etc. have received the emails), but it is rather difficult. For example, it is not easy to write code to check the mail box of Mary. At the same time, we don't have her password. Besides, she may have downloaded the mail before we check. Also, the email may be just on its way to her mail box. We can only lower our expectation. Instead of checking whether she has received the email, we check whether the system has sent the email. To check whether the system has sent the email, strictly speaking we need to check whether it has issued the SMTP commands, whether the recipient and email contents embedded in the commands are correct. This is also not easy to do. Therefore, we lower our expectation again. We assume that the system will record the recipient and contents of every email it sends. In the test we only check this record:

```
class AcceptTestForImportParticipants {
   static void testImport() {
      Command commands[] = {
         new CreateImportFileCommand(...),
         new DeleteAllParticipantsCommand(),
         new ImportParticipantsFromFileCommand(),
         new CheckParticipantStoredCommand("p001",  "123456",      "Mary Lam", ...),
         new CheckParticipantStoredCommand("p002", ...),
         new CheckParticipantStoredCommand("p004", ...),
         new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
         new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
         new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
         ...
      };
      runCommands(commands);
   }
}
class CheckRemoveOldestMailCommand implements Command {
   MailRecord mailRecord;
   CheckRemoveOldestMailCommand(String recipientEmail, String mailContent) {
      mailRecord = new MailRecord(recipientEmail, mailContent);
   }
   boolean run() {
      MailLog mailLog = ConferenceSystem.getInstance().mailLog;
      return mailRecord.equals(mailLog.takeOldestMailRecord());
   }
}
class ConferenceSystem {
   Participants parts;
   MailLog mailLog;
   static ConferenceSystem getInstance() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      ...
   }
}
class MailLog {
   MailRecord mailRecords[];
   MailRecord takeOldestMailRecord() {
      ...
   }
}
class MailRecord {
   MailRecord(String recipientEmail, String mailContent) {
      ...
   }
   boolean equals(Object obj) {
      ...
   }
}
```

In fact, there is a problem: When the test case is run, MailLog may contain some residual records. When we take the oldest MailRecord, we will get the wrong one. To solve this problem, testImport should empty the MailLog at the beginning:

```
class AcceptTestForImportParticipants {
   static void testImport() {
      Command commands[] = {
         new EmptyMailLogCommand(),
         new CreateImportFileCommand(...),
         new DeleteAllParticipantsCommand(),
         new ImportParticipantsFromFileCommand(),
         new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
         new CheckParticipantStoredCommand("p002", ...),
         new CheckParticipantStoredCommand("p004", ...),
         new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
         new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
         new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
         ...
      };
      runCommands(commands);
   }
}
class EmptyMailLogCommand implements Command {
   boolean run() {
      MailLog mailLog = ConferenceSystem.getInstance().mailLog;
      mailLog.empty();
      return true;
   }
}
class MailLog {
   MailRecord mailRecords[];
   MailRecord takeOldestMailRecord() {
      ...
   }
   void empty() {
      ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   }
}
```

We find that at the beginning of a test case, we commonly need to empty many things such as MailLog, Participants and etc. Therefore, we seem to need a command to empty or initialize all the things in the system. Let's call it SystemInit. Therefore, we no longer needs the DeleteAllParticipants command and the EmptyMailLog command:

```
class AcceptTestForImportParticipants {
   static void testImport() {
      Command commands[] = {
         new SystemInitCommand(),
         new CreateImportFileCommand(...),
         new ImportParticipantsFromFileCommand(),
         new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
         new CheckParticipantStoredCommand("p002", ...),
         new CheckParticipantStoredCommand("p004", ...),
         new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
         new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
         new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
         ...
      };
      runCommands(commands);
   }
}
class SystemInitCommand implements Command {
   boolean run() {
      ConferenceSystem.getInstance().init();
      return true;
   }
}
class ConferenceSystem {
   Participants parts;
   MailLog mailLog;
   void init() {
      parts.deleteAll();
      mailLog.empty();
   }
}
```

We have completely automated test case 1. Below is the complete code:

```
class ConferenceSystem {
   Participants parts;
   MailLog mailLog;
   void init() {
      parts.deleteAll();
      mailLog.empty();
   }
   static ConferenceSystem getInstance() {
      ...
   }
}
class Participants {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      void deleteAll() {
         ...
      }
      void importFromFile(String path) {
         ...
      }
      boolean checkParticipantStored(Participant part) {
         return part.equals(getParticipantById(part.getId()));
      }
      Participant getParticipantById(String partId) {
         ...
      }
   }
   class Participant {
      boolean equals(Object obj) {
         ...
      }
   }
   class MailRecord {
      MailRecord(String recipientEmail, String mailContent) {
         ...
      }
      boolean equals(Object obj) {
         ...
      }
   }
   class MailLog {
      MailRecord mailRecords[];
      MailRecord takeOldestMailRecord() {
         ...
      }
   }
   class AcceptTestForImportParticipants {
      static void testImport() {
         Command commands[] = {
            new SystemInitCommand(),
            new CreateImportFileCommand(...),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001",  "123456",      "Mary
   Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
   & ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
   "),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
   "),
            ...
         };
         runCommands(commands);
      }
      static void testImportWithoutAddress() {
         ...
      }
      static void testImportWithoutOtherInfo() {
         ...
      }
      static void testImportDupId() {
         ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
    static void runCommands(Command commands[]) {
       for (int i = 0; i < commands.length; i++) {
          if (!commands[i].run()) {
             System.out.println("Test failed!");
          }
       }
    }
}
interface Command {
    boolean run();
}
class SystemInitCommand implements Command {
    boolean run() {
       ConferenceSystem.getInstance().init();
       return true;
    }
}
class CreateImportFileCommand implements Command {
    String fileContent;
    CreateImportFileCommand(String fileContent) {
       this.fileContent = fileContent;
    }
    boolean run() {
       FileWriter fileWriter = new FileWriter("sourceFile.txt");
       try {
          fileWriter.write(fileContent);
          return true;
       } finally {
          fileWriter.close();
       }
    }
}
class ImportParticipantsFromFileCommand implements Command {
    boolean run() {
       ConferenceSystem.getInstance().parts.importFromFile("sourceFile.txt");
       return true;
    }
}
class CheckParticipantStoredCommand implements Command {
    Participant part;
    CheckParticipantStoredCommand(String partId, String password, ...) {
       part = new Participant(partId, password, ...);
    }
    boolean run() {
       return ConferenceSystem.getInstance().parts.checkParticipantStored(part);
    }
}
class CheckRemoveOldestMailCommand implements Command {
    MailRecord mailRecord;
    CheckRemoveOldestMailCommand(String recipientEmail, String mailContent) {
       mailRecord = new MailRecord(recipientEmail, mailContent);
    }
    boolean run() {
       MailLog mailLog = ConferenceSystem.getInstance().mailLog;
       return mailRecord.equals(mailLog.takeOldestMailRecord());
    }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Commands should NOT include domain logic

All the XXXCommand classes above have one thing in common: They all contain just a little code. This is not by chance. This is because they are simulating the requests or actions of the users (e.g., ask the system to import a file). As the system already contains code to satisfy these requests, the commands only need to call the existing code in the system.

When we are writing these command classes, we may find that there is no existing code in the system for use. For example, when writing the CheckParticipantStoredCommand, we may find that there is no method like getParticipantById in the Participants class. If we are not careful, we may choose to implement this function directly in CheckParticipantStoredCommand:

```
class CheckParticipantStoredCommand implements Command {
   Participant part;
   ..
   boolean run() {
      return part.equals(getParticipantById(part.getId()));
   }
   Participant getParticipantById(String partId) {
      //lots of code to read the data from the database.
      ...
      ...
      ...
   }
}
```

However, there is a problem here. Because the users need to retrieve the information of the participants, getParticipantById has implemented a function really needed by the users (user story). As CheckParticipantStoredCommand is only used for testing, when the system is delivered, all these XXXCommand classes including CheckParticipantStoredCommand should be excluded. But here we would like to keep getParticipantById because it includes useful functionality (domain logic).

Therefore, all these XXXCommand classes should only implement test cases, not domain logic. Domain logic should be implemented by the "regular" classes in the system such as Participants, Participant and etc. The XXXCommand classes should only need to call these "regular" classes. If you find that a participant Command class contains "too much" code, or some other classes in the system need to call this Command class, you need to check if this Command class has implemented the business logic needed by the system. If yes, you should move the logic into the "regular" classes in the system.

Writing test cases first as the requirements

Suppose that after implementing this "import participants" user story, we work with the customer to write all the four automated test cases. We run them to see if they pass. If yes, it

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

means the user story has been implemented (for the best results, try to run them at least once in front of the customer). However, unfortunately, the system fails in a test case that tries to import the following file given by the customer:

```
p001  123456      Mary Lam    abc     mary@extremely.long.domain.com
p004  888999      John Chan   def     john@yahoo.com
p002  mypasswd Paul Lei       ghi     paul@excite.com
```

After two hours of debugging, we finally find that the maximum size of the email address in the database is set to 26 characters, but now the email of Mary contains 32 characters. Therefore, the database only has mary@extremely.long.doma, not mary@extremely.long.domain.com. Therefore, when retrieving p001, checkParticipantStored fails.

If at the beginning when we started to implement this user story, we had worked with the customer to write the test cases, we would have learned from the customer or from the test cases that we needed at least 32 characters to store an email address. We would have avoided the mistake when we created the database. We would also have saved that two hours of debugging.

Therefore, test cases should be written before we start to implement a user story. Test cases are an important component of our software requirements (the live customer is the other component). As part of the requirements, the test cases can guide our implementation work. The advantages of the live customer are that the information is most updated and that the communication is efficient (conversation is easier than reading); The advantages of (automated) test cases are that they are accurate and can be run often.

Make the test cases understandable to the customer

Only the customer knows the requirements. Therefore, only he has the right to specify the contents of the test cases. What we can do at most is just to help him write them down. We must never determine the contents of the test cases on his behalf. At the moment, we use Java code to specify the contents of the test cases, which can't be understood by a regular customer who is not a developer:

```
class AcceptTestForImportParticipants {
   static void testImport() {
      Command commands[] = {
         new SystemInitCommand(),
         new CreateImportFileCommand(...),
         new ImportParticipantsFromFileCommand(),
         new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
         new CheckParticipantStoredCommand("p002", ...),
         new CheckParticipantStoredCommand("p004", ...),
         new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
         new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
        };
        runCommands(commands);
    }
}
```

This is a serious problem. If the customer can't understand the test cases, how can he judge whether what we write is exactly what he has in mind? In order to make this test case easier to understand, we can rewrite it as a text file named testImport.test with the following contents:

```
SystemInit
CreateImportFile
   p001,123456,Mary Lam,abc,mary@hotmail.com
   p004,888999,John Chan,def,john@yahoo.com
   p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckParticipantStored,p002,mypasswd,Paul Lei,ghi,paul@excite.com
CheckParticipantStored,p004,888999,John Chan,def,john@yahoo.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
CheckRemoveOldestMail,john@yahoo.com,This is your Id &...
CheckRemoveOldestMail,paul@excite.com,This is your Id &...
```

Let's call this kind of file a "test file". In a test file basically every command takes one line, with the exception of CreateImportFile which takes five lines:

```
CreateImportFile
   p001,123456,Mary Lam,abc,mary@hotmail.com
   p004,888999,John Chan,def,john@yahoo.com
   p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
```

From the view of the customer, the system can execute this testImport.test file. That is, it can sequentially execute each command in this file.

In order to support this kind of test file, we need to know how to read in a test file and then generate a list of corresponding commands:

```
class TestFileParser {
   CommandParser commandParsers[] = {
      new SystemInitCommandParser(),
      new CheckParticipantStoredCommandParser(),
      new CreateImportFileCommandParser(),
      ...
   };
   Command[] parseCommandsFromFile(String pathToTestFile) {
      TokenIterator tokenIterator = parseFileToTokens(pathToTestFile);
      return parseCommands(tokenIterator);
   }
   TokenIterator parseFileToTokens(String path) {
      ...
   }
   Command[] parseCommands(TokenIterator tokenIterator) {
      Command commands[];
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        while (tokenIterator.hasToken()) {
           Command nextCommand = parseCommand(tokenIterator);
           append nextCommand to commands;
        }
        return commands;
     }
     Command parseCommand(TokenIterator tokenIterator) {
        for (int i = 0; i < commandParsers.length; i++) {
           Command command = commandParsers[i].parse(tokenIterator);
           if (command != null) {
              return command;
           }
        }
        throw new CommandSyntaxException();
     }
  }
  interface TokenIterator {
     boolean hasToken();
     String peekToken();
     String takeToken();
     void next();
  }
  interface CommandParser {
     Command parse(TokenIterator tokenIterator);
  }
  class SystemInitCommandParser implements CommandParser {
     Command parse(TokenIterator tokenIterator) {
        if (tokenIterator.peekToken().equals("SystemInit")) {
           tokenIterator.next();
           return new SystemInitCommand();
        } else {
           return null;
        }
     }
  }
  class CheckParticipantStoredCommandParser implements CommandParser {
     Command parse(TokenIterator tokenIterator) {
        if (tokenIterator.peekToken().equals("CheckParticipantStored")) {
           tokenIterator.next();
           String partId = tokenIterator.takeToken();
           String password = tokenIterator.takeToken();
           String name = tokenIterator.takeToken();
           String address = tokenIterator.takeToken();
           String email = tokenIterator.takeToken();
           return new CheckParticipantStoredCommand(
              partId,
              password,
              name,
              address,
              email);
        } else {
           return null;
        }
     }
  }
  class CreateImportFileCommandParser implements CommandParser {
     ...
  }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

A test file doesn't have to be a text file

A test file doesn't have to be a text file. For example, it may be an Excel file:

SystemInit					
CreateImportFile					
	p001	123456	Mary Lam	abc	mary@hotmail.com
	p004	888999	John Chan	def	john@yahoo.com
	p002	mypasswd	Paul Lei	ghi	paul@excite.com
ImportFromFile					
CheckParticipantStored	p001	123456	Mary Lam	abc	mary@hotmail.com
CheckParticipantStored	p002	mypasswd	Paul Lei	ghi	paul@excite.com
CheckParticipantStored	p004	888999	John Chan	def	john@yahoo.com
...					

Of course, if we really used an Excel file, our TestFileParser would have to be changed accordingly. It is also possible to use an HTML file or some other formats.

Use test cases to prevent the system functionality from downgrading

Whenever we work with the customer to create a test file, we place it into a folder named "Queued". At the beginning all the test files are placed here. Whenever we make a test file pass, we move it from Queued into another folder named "Passed". Whenever we modify the code or add some code, we need to run all the test files in Passed again to make sure that they still pass. If a certain test file fails, we must modify the code to make it pass again. We must not move it from Passed back to Queued. That is, once a test file is moved into Passed, it will never get out of there (unless the customer finds an error in the test file itself), i.e., the movement from Queued to Passed is strictly one way only and can't be reversed.

Because this movement is one way only, there will be more and more test files in Passed and less and less test files in Queued, meaning that the functionality of our system can only go up and up. It will never go down. Finally when Queued becomes empty and if the customer has no more new test cases, we are done.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

References

- http://c2.com/cgi/wiki?AcceptanceTest.
- Ward Cunningham has written an acceptance test framework called "Fit". It is available at http://fit.c2.com.
- Based on Fit, FitNesse is a web-based environment allowing developers to create and run acceptance tests. It is available at http://fitnesse.org.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

1. Write the test cases 2, 3 and 4 for the user story "import participants" as test files and implement the commands needed (you can assume the system already contains the required functions).

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Write the test cases 2, 3 and 4 for the user story "import participants" as test files and implement the commands needed (you can assume the system already contains the required functions).

   The test file for test case 2:

```
SystemInit
CreateImportFile
   p001,123456,Mary Lam,abc,mary@hotmail.com
   p004,888999,John Chan,,john@yahoo.com
   p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckParticipantStored,p002,mypasswd,Paul Lei,ghi,paul@excite.com
CheckParticipantStored,p004,888999,John Chan,,john@yahoo.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
CheckRemoveOldestMail,john@yahoo.com,This is your Id &...
CheckRemoveOldestMail,paul@excite.com,This is your Id &...
```

   The test file for test case 3:

```
SystemInit
CreateImportFile
   ,123456,Mary Lam,abc,mary@hotmail.com
   p004,888999,,def,john@yahoo.com
   p002,,Paul Lei,ghi,paul@excite.com
   p005,secret,Mike Chan,jkl,
CreateImportFileEnd
ImportFromFile
CheckNoParticipantsStored
CheckMailLogEmpty
```

   We need to implement two new commands: CheckNoParticipantsStored and CheckMailLogEmpty.

```
class CheckNoParticipantsStoredCommand implements Command {
   boolean run() {
      Participants parts = ConferenceSystem.getInstance().parts;
      return parts.isEmpty();
   }
}
class CheckMailLogEmptyCommand implements Command {
   boolean run() {
      MailLog mailLog = ConferenceSystem.getInstance().mailLog;
      return mailLog.isEmpty();
   }
}
```

   The test file for test case 4:

```
SystemInit
CreateImportFile
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   p001,123456,Mary Lam,abc,mary@hotmail.com
   p001,888999,John Chan,def,john@yahoo.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 11

How to Acceptance Test a User Interface

How to control a user interface

Suppose that the customer requests us to implement the following user story:

Name: Add or edit a participant

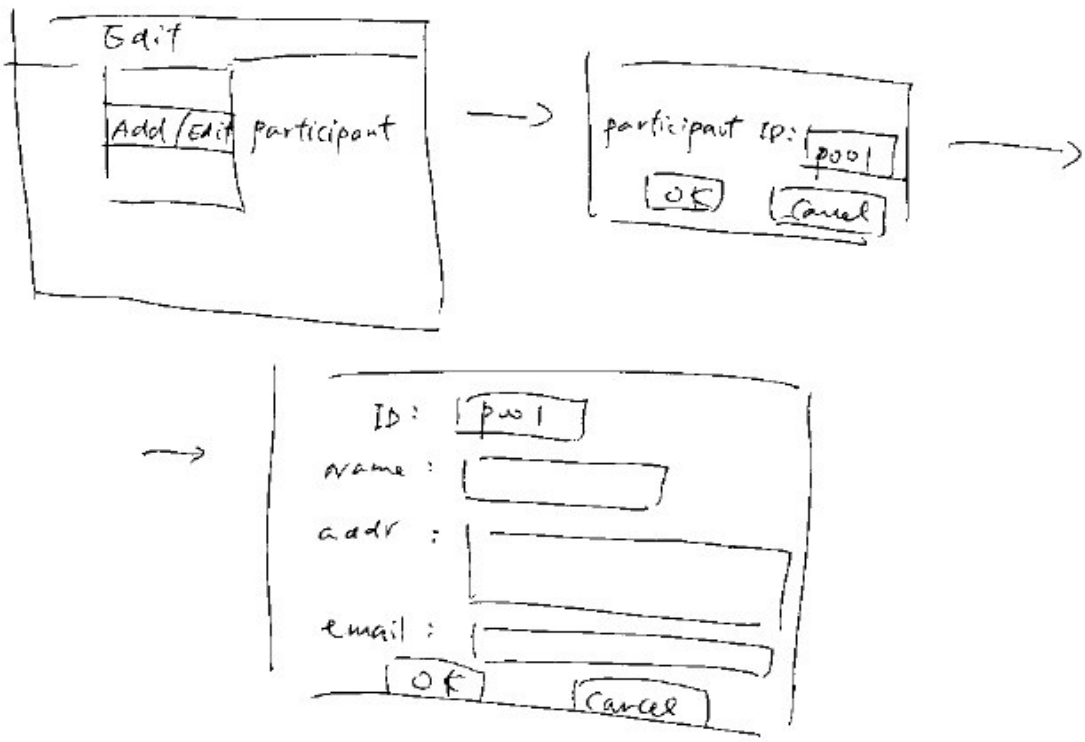
Events:

1. The user inputs a participant ID.
2. If it is a new ID, the user inputs the name, address and email of the participant.
3. If it is an existing ID, the system displays the name, address and email of the participant for the user to edit.
4. The system saves the information about the participant. In the future a user can input the ID of this participant, the system will retrieve the information about him.

Strictly speaking, the underlined portion in the above user story should not appear here because it describes the internal behavior of the system. What is important for the user is that he can retrieve the information of that participant in the future. However, we allow it here because it should help the user understand.

After discussing with us, the customer accepts our suggestion to adopt such a user interface: A user chooses a menu item "Add/Edit Participant" in the main window, then input a participant ID in a dialog. After clicking OK, the system will display another dialog to let the user input or edit the other information of the participant. After clicking OK, the system will save the information. There is a cancel button on both dialogs. Below is the sketches you drew on the white board during the discussion:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



The question is, how to test this function that involves a user interface?

Test each user interface component separately

The above user story involves 3 user interface components:

1. The menu item "Add/Edit Participant".
2. The dialog allowing the user to input a participant ID. Let's call it ParticipantIdDialog.
3. The dialog allowing the user to input or edit the information of a participant. Let's call it ParticipantDetailsDialog.

Usually it is quite hard to test several user interface components together (Why? It takes quite some effort to explain. Therefore, please trust me. If not, you can try testing them all in one go and see). Therefore, usually we will test each user interface component separately.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Because the most complicated component in this example is ParticipantDetailsDialog, we will see how to test it first.

How to test ParticipantDetailsDialog

We can use the following test case to test ParticipantDetailsDialog: Initialize the system, provide a new participant ID to ParticipantDetailsDialog, input the name, address and email of the participant, click OK and see if the system has saved the information.

The corresponding test file may be:

```
SystemInit
ShowParticipantDetailsDialog,p001
SetParticpantName,Mike Chan
SetParticpantPassword,888888
SetParticpantAddress,aaabbb
SetParticpantEmail,mike@excite.com
ClickOK
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

In which the ShowParticipantDetailsDialog command creates a ParticipantDetailsDialog and provides it with p001 as the participant ID. The SetParticipantName command simulates the user's action of inputting the name of the participant (Mike Chan). Similarly, SetParticipantPassword, SetParticipantAddress and SetParticipantEmail simulate the user's action of inputting the password, address and email respectively. ClickOK simulates the action of clicking OK.

However, there is a problem here. For example, when SetParticipantName is run, how does it know which ParticipantDetailsDialog it should send the name "Mike Chan" to? In our system there is not a global ParticipantDetailsDialog object. Besides, as mentioned above, that ParticipantDetailsDialog is dynamically created by the ShowParticipantDetailsDialog command. Similarly, how does ClickOK know it should click the OK button on which dialog?

Therefore, we hope to use a variable in the test file to store that dialog, e.g.:

```
SystemInit
dialog=ShowParticipantDetailsDialog,p001
SetParticpantName,dialog,Mike Chan
SetParticpantPassword,dialog,888888
SetParticpantAddress,dialog,aaabbb
SetParticpantEmail,dialog,mike@excite.com
ClickOK,dialog
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

However, this will make the test files as complicated as a programming language. A possible solution is to make commands like SetParticipantName "sub-commands" of the ShowParticipantDetailsDialog command, e.g.:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
SystemInit
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

In which the 7 lines from ShowParticipantDetailsDialogStart to ShowParticipantDetailsDialogEnd define only one ShowParticipantDetailsDialog command. Each of 5 lines in this command is a sub-command such as SetParticipantName. When this ShowParticipantDetailsDialog command is run, it will create a ParticipantDetailsDialog object and then tell all its 5 sub-commands to use this ParticipantDetailsDialog.

These commands can be implemented like this:

```
class ShowParticipantDetailsDialogCommand implements Command {
   String partId;
   SubCommand subCommands[];
   ShowParticipantDetailsDialogCommand(String partId) {
      this.partId=partId;
   }
   void addSubCommand(SubCommand subCommand) {
      //add subCommand to subCommands
      ...
   }
   boolean run() {
      ParticipantDetailsDialog partDetailsDlg=
         new ParticipantDetailsDialog(partId);
      //do not call showModal on the dialog, otherwise it will show on the screen,
      //waiting for a real user to interact with it.
      For (int i=0; i<subCommands.length; i++) {
         subCommands[i].setDialogToUse(partDetailsDlg);
         if (!subCommands[i].run()) {
            return false;
         }
      }
      return true;
   }
   static abstract class SubCommand implements Command {
      ParticipantDetailsDialog partDetailsDlg;
      void setDialogToUse(ParticipantDetailsDialog partDetailsDlg) {
         this.partDetailsDlg=partDetailsDlg;
      }
   }
   static class SetParticipantNameCommand extends SubCommand {
      String partName;
      SetParticipantNameCommand(String partName) {
         this.partName=partName;
      }
      boolean run() {
         partDetailsDlg.setParticipantName(partName);
         return true;
      }
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      static class ClickOKCommand extends SubCommand {
         boolean run() {
            partDetailsDlg.clickOK();
            return true;
         }
      }
   }
   class ParticipantDetailsDialog extends Jdialog {
      String partId;
      JtextField partName;
      JtextField partEmail;
      ...
      JButton OK;
      Jbutton cancel;
      ParticipantDetailsDialog(String partId) {
         this.partId=partId;
      }
      void setParticipantName(String name) {
         partName.setText(name);
      }
      void setParticipantPassword(String password) { ... }
      void setParticipantEmail(String email) { ... }
      void setParticipantAddress(String address) { ... }
      void clickOK() { ... }
      void clickCancel() { ... }
   }
```

Note that the run method in ShowParticipantDetailsDialog creates a ParticipantDetailsDialog, but it doesn't call showModal to show it. This is right. If it did, the dialog would be shown on the screen, waiting for a user to input. Then the ShowParticipantDetailsDialog command could not continue to execute its sub-commands.

Other things to test

We have tested whether ParticipantDetailsDialog can handle the case when a new participant is added. We also need to test the case when the participant ID already exists or when the user clicks Cancel. Then, we should test ParticipantIdDialog. Because the behavior of a menu item is too simple. We don't need to test it.

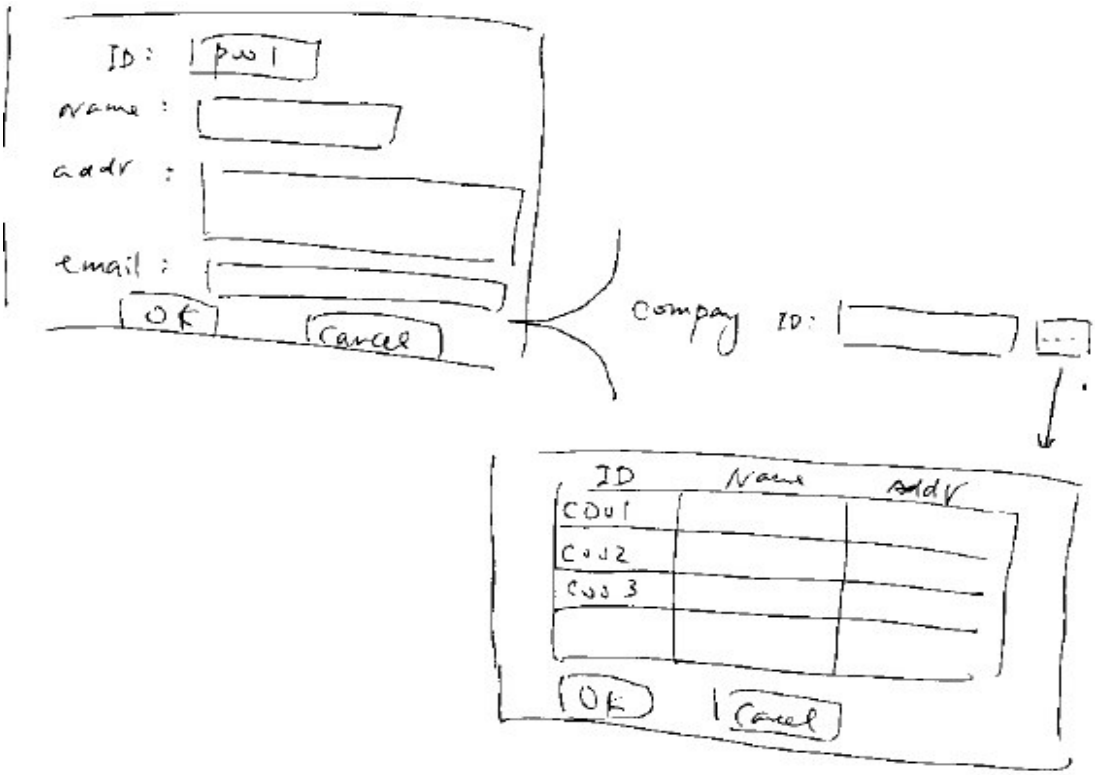
Because the way to test these is the same, we will not talk about it here.

What if one dialog needs to pop up another?

Commonly we need to pop up one dialog from another. For example, ParticipantDetailsDialog may need the users to input the company ID of the participant. To make it convenient for the user, by the side of the company ID edit box there is a "..." button. The user can click this button to display another dialog (let's call it CompaniesDialog) to display all the companies in

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

the system and then chooses one of them, saving the need to input that company ID manually. Below is the sketches you drew on the white board during the discussion:



The question is, how to test this function of selecting a company? Would the test file below work?

```
SystemInit
AddCompany,c001,...
AddCompany,c002,...
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ShowCompaniesDialog
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,...
```

In which, in order to have some companies for a user to select, we use two AddCompany commands to add two companies (with ID of c001 and c002 respectively) to the system. The ShowCompaniesDialog command pops up a CompaniesDialog.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

However, this ShowCompaniesDialog command doesn't work. In addition to creating a CompaniesDialog, we also need to interact with the CompaniesDialog, e.g., select a particular company and then click OK. Therefore, we change it to:

```
SystemInit
AddCompany,c001,...
AddCompany,c002,...
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ShowCompaniesDialogStart
      SelectRowInGrid,1
      ClickOK
   ShowCompaniesDialogEnd
   GetCompanyId,c002
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,...
```

In which we use a SelectRowInGrid command to select row 1 in the grid (assuming the first row is row 0), that is, the row representing c002. Then we use a ClickOK command to click OK. In the ParticipantDetailsDialog command we use a GetCompanyId sub-command to check if c002 has been input automatically as the company ID.

This method works. However, we are testing ParticipantDetailsDialog and CompaniesDialog together. As mentioned before, usually it is easier to test each user interface component separately. Therefore, we wish that ParticipantDetailsDialog didn't use another dialog. Does it really have to use CompaniesDialog? In fact, what it really needs is just to get a company ID from someone. Therefore, we can abstract the CompaniesDialog as a string provider or a string source and let ParticipantDetailsDialog use a string source instead of a dialog:

```
interface StringSource {
   String getString();
}
class CompaniesDialog extends Jdialog implements StringSource {
   String getString() {
      showModal();
      return OKClicked() ? getSelectedCompanyId() : null;
   }
   boolean OKClicked() { ... }
   String getSelectedCompanyId() { ... }
}
class ParticipantDetailsDialog extends Jdialog {
   String partId;
   StringSource companyIdSource;
   JtextField partName;
   JtextField partEmail;
   JtextField compId;
   ...
   JButton OK;
   Jbutton cancel;
   Jbutton chooseCompany;
   ParticipantDetailsDialog(String partId) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        this.partId=partId;
        this.companyIdSource=new CompaniesDialog();
    }
    void clickChooseCompany() {
        String companyId=companyIdSource.getString();
        if (companyId!=null) {
            compId.setText(companyId);
        }
    }
}
```

The original test file can be split into two. One file tests ParticipantDetailsDialog:

```
SystemInit
AddCompany,c001,...
AddCompany,c002,...
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ClickChooseCompany,c002
   GetCompanyId,c002
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,...
```

The other file tests CompaniesDialog:

```
SystemInit
AddCompany,c001,...
AddCompany,c002,...
ShowCompaniesDialogStart
   SelectRowInGrid,1
   ClickOK
   GetSelectedCompanyId,c002
ShowCompaniesDialogEnd
```

The ClickChooseCompany sub-command can be implemented like this:

```
class ShowParticipantDetailsDialogCommand implements Command {
   ...
   static class ClickChooseCompanyCommand extends SubCommand {
      String companyId;
      ClickChooseCompanyCommand(String companyId) {
         this.companyId=companyId;
      }
      boolean run() {
         partDetailsDlg.companyIdSource=new StringSource() {
            String getString() {
               return companyId;
            }
         });
         partDetailsDlg.clickChooseCompany();
         return true;
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        }
    }
```

References

- http://c2.com/cgi/wiki?GuiTesting.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Problems

1. Write a test file to test whether ParticipantDetailsDialog can handle the case when a user clicks Cancel. Implement the required commands.

2. Write a test file to test whether ParticipantDetailsDialog can handle the case when the participant ID already exists. Implement the required commands.

3. Implement the setParticipantAddress method in ParticipantDetailsDialog.

4. Write some test files to test ParticipantIdDialog. Implement the required commands.

Hints

1. No hint for this one.

2. No hint for this one.

3. There should be a GetParticipantId sub-command that checks the participant ID input by user.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Write a test file to test whether ParticipantDetailsDialog can handle the case when a user clicks Cancel. Implement the required commands.

```
SystemInit
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ClickCancel
ShowParticipantDetailsDialogEnd
CheckNoParticipantsStored
```

No command needs to be implemented.

2. Write a test file to test whether ParticipantDetailsDialog can handle the case when the participant ID already exists. Implement the required commands.

```
SystemInit
AddParticipant,p001,Mary Lam,123456,abc,mary@hotmail.com
AddParticipant,p004,John Chan,888999,def,john@yahoo.com
ShowParticipantDetailsDialogStart,p001
   GetParticpantName,Mary Lam
   GetParticpantPassword,123456
   GetParticpantAddress,abc
   GetParticpantEmail,mary@hotmail.com
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

Implementation of the commands:

```
class AddParticipantCommand implements Command {
   Participant part;
   AddParticipantCommand(Participant part) {
      this.part=part;
   }
   boolean run() {
      ConferenceSystem.getInstance().parts.addParticipant(part);
      return true;
   }
}
class ShowParticipantDetailsDialogCommand implements Command {
   ...
   static class GetParticipantNameCommand extends SubCommand {
      String partName;
      GetParticipantNameCommand(String partName) {
         this.partName=partName;
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      boolean run() {
         return partDetailsDlg.getParticipantName.equals(partName);
      }
   }
   static class GetParticipantPasswordCommand extends SubCommand {
      ...
   }
   static class GetParticipantAddressCommand extends SubCommand {
      ...
   }
   static class GetParticipantEmailCommand extends SubCommand {
      ...
   }
}
class ParticipantDetailsDialog extends JDialog {
   ...
   JTextField partName;
   ...
   String getParticipantName() {
      return partName.getText();
   }
}
```

3. Implement the setParticipantAddress method in ParticipantDetailsDialog.

```
class ShowParticipantDetailsDialogCommand implements Command {
   ...
   static class SetParticipantAddressCommand extends SubCommand {
      String partAddress;
      SetParticipantAddressCommand(String partAddress) {
         this.partAddress=partAddress;
      }
      boolean run() {
         partDetailsDlg.setParticipantAddress(partAddress);
         return true;
      }
   }
   static class ClickOKCommand extends SubCommand {
      boolean run() {
         partDetailsDlg.clickOK();
         return true;
      }
   }
}
class ParticipantDetailsDialog extends JDialog {
   ...
   JTextField partAddress;
   ...
   void setParticipantAddress(String address) {
      partAddress.setText(address);
   }
}
```

4. Write some test files to test ParticipantIdDialog. Implement the required commands.

   Test if it can return the participant ID input by a user:

```
SystemInit
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
ShowParticipantIDDialogStart
   SetParticpantID,p001
   ClickOK
   GetParticpantID,p001
ShowParticipantIDDialogEnd
```

Test if it will return empty if a user clicks Cancel:

```
SystemInit
ShowParticipantIDDialogStart
   SetParticpantID,p001
   ClickCancel
   GetParticpantID,
ShowParticipantIDDialogEnd
```

Implementation of the commands:

```
class ShowParticipantIDDialogCommand implements Command {
   ...
   static abstract class SubCommand implements Command {
      ...
   }
   boolean run() {
      ParticipantIDDialog partIDDlg=new ParticipantIDDialog();
      for (int i=0; i<subCommands.length; i++) {
         subCommands[i].setDialogToUse(partIDDlg);
         if (!subCommands[i].run()) {
            return false;
         }
      }
      return true;
   }
   static class SetParticipantIDCommand extends SubCommand {
      String partID;
      SetParticipantIDCommand(String partID) {
         this.partID=partID;
      }
      boolean run() {
         partIDDlg.setParticipantID(partID);
         return true;
      }
   }
   static class GetParticipantIDCommand extends SubCommand {
      String partID;
      GetParticipantIDCommand(String partID) {
         this.partID=partID;
      }
      boolean run() {
         return partIDDlg.getParticipantID().equals(partID);
      }
   }
   static class ClickOKCommand extends SubCommand {
      boolean run() {
         partIDDlg.clickOK();
         return true;
      }
   }
   static class ClickCancelCommand extends SubCommand {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      boolean run() {
         partIDDlg.clickCancel();
         return true;
      }
   }
}
class ParticipantIDDialog extends JDialog {
   ...
   JTextField partID;
   ...
   void setParticipantID(String id) {
      partID.setText(id);
   }
   void clickOK() { ... }
   void clickCancel() { ... }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 12

Unit Test

Unit test

Suppose that you are writing a CourseCatalog class to record the information of some courses:

```
class CourseCatalog {
   CourseCatalog() {
      ...
   }
   void add(Course course) {
      ...
   }
   void remove(Course course) {
      ...
   }
   Course findCourseWithId(String id) {
      ...
   }
   Course[] findCoursesWithTitle(String title) {
      ...
   }
}
class Course {
   Course(String id, String title, ...) {
   }
   String getId() {
      ...
   }
   String getTitle() {
      ...
   }
}
```

In order to ensure that CourseCatalog is free of bug, we should test it. For example, in order to see if its add method is really adding a Course into it, we can do it this way:

```
class TestCourseCatalog {
   static void testAdd() {
      CourseCatalog cat = new CourseCatalog();
      Course course = new Course("c001", "Java prgoramming", ...);
      cat.add(course);
      if (!cat.findCourseWithId(course.getId()).equals(course)) {
         throw new TestFailException();
      }
   }
   public static void main(String args[]) {
      testAdd();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
}
```

Because testAdd needs to check if two Course objects are equal, the Course class needs to provide an equals method:

```
class Course {
   ...
   boolean equals(Object obj) {
      ...
   }
}
```

Maybe we are not very confident about the add method yet. Maybe we are worried that it may not be able to store two or more courses. Therefore we write another test:

```
class TestCourseCatalog {
   static void testAdd() {
      ...
   }
   static void testAddTwo() {
      CourseCatalog cat = new CourseCatalog();
      Course course1 = new Course("c001", "Java prgoramming", ...);
      Course course2 = new Course("c002", "OO design", ...);
      cat.add(course1);
      cat.add(course2);
      if (!cat.findCourseWithId(course1.getId()).equals(course1)) {
         throw new TestFailException();
      }
      if (!cat.findCourseWithId(course2.getId()).equals(course2)) {
         throw new TestFailException();
      }
   }
   public static void main(String args[]) {
      testAdd();
      testAddTwo();
   }
}
```

Similarly, we can test the remove method in CourseCatalog:

```
class TestCourseCatalog {
   static void testAdd() {
      ...
   }
   static void testAddTwo() {
      ...
   }
   static void testRemove() {
      CourseCatalog cat = new CourseCatalog();
      Course course = new Course("c001", "Java prgoramming", ...);
      cat.add(course);
      cat.remove(course);
      if (cat.findCourseWithId(course.getId()) != null) {
         throw new TestFailException();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        }
    }
    public static void main(String args[]) {
        testAdd();
        testAddTwo();
        testRemove();
    }
}
```

The purpose of testAdd, testAddTwo and testRemove above is to test the CourseCatalog class. Each of them is called a test case. However, they are not acceptance tests, but are the so-called "unit tests", because they only test a unit (CourseCatalog class).

How is a unit test differ from an acceptance test? An acceptance test tests the external behavior of a system, while a unit test tests a unit (class). Acceptance tests belong to the client. We have no right to determine their contents. We can at most help the customer write down the acceptance tests according to the user stories. Units tests belong to us, because what classes are there in the system and what each class should do are decided by us. The client has no right to get involved, nor do we need his participation. We simply write the unit tests according to our expectation on a unit (class). Because of this, this kind of test is also called "programmer test".

Use JUnit

When writing unit tests, we can make use of a software package called "JUnit". With JUnit, we should change the above unit testing code to:

```
import junit.framework.*;
import junit.swingui.*;
public class TestCourseCatalog extends TestCase {
    CourseCatalog cat;
    protected void setUp() {
        cat = new CourseCatalog();
    }
    public void testAdd() {
        Course course = new Course("c001", "Java prgoramming", ...);
        cat.add(course);
        assertEquals(cat.findCourseWithId(course.getId()), course);
    }
    public void testAddTwo() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        cat.add(course2);
        assertEquals(cat.findCourseWithId(course1.getId()), course1);
        assertEquals(cat.findCourseWithId(course2.getId()), course2);
    }
    public void testRemove() {
        Course course = new Course("c001", "Java prgoramming", ...);
        cat.add(course);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        cat.remove(course);
        assertTrue(cat.findCourseWithId(course.getId()) == null);
    }
    public static void main(String args[]) {
        TestRunner.run(TestCourseCatalog.class);
    }
}
```

Below we will explain the changes.

The assertEquals(X, Y) method checks if X and Y are equal (it calls the equals method of X). If not, it throws an exception. assertEquals is provided by the TestCase class. Because now TestCourseCatalog extends TestCase, we can directly call assertEquals.

The TestCase class is provided by JUnit. It is in the junit.framework package. Therefore, we need to import junit.framework.* in order to use it.

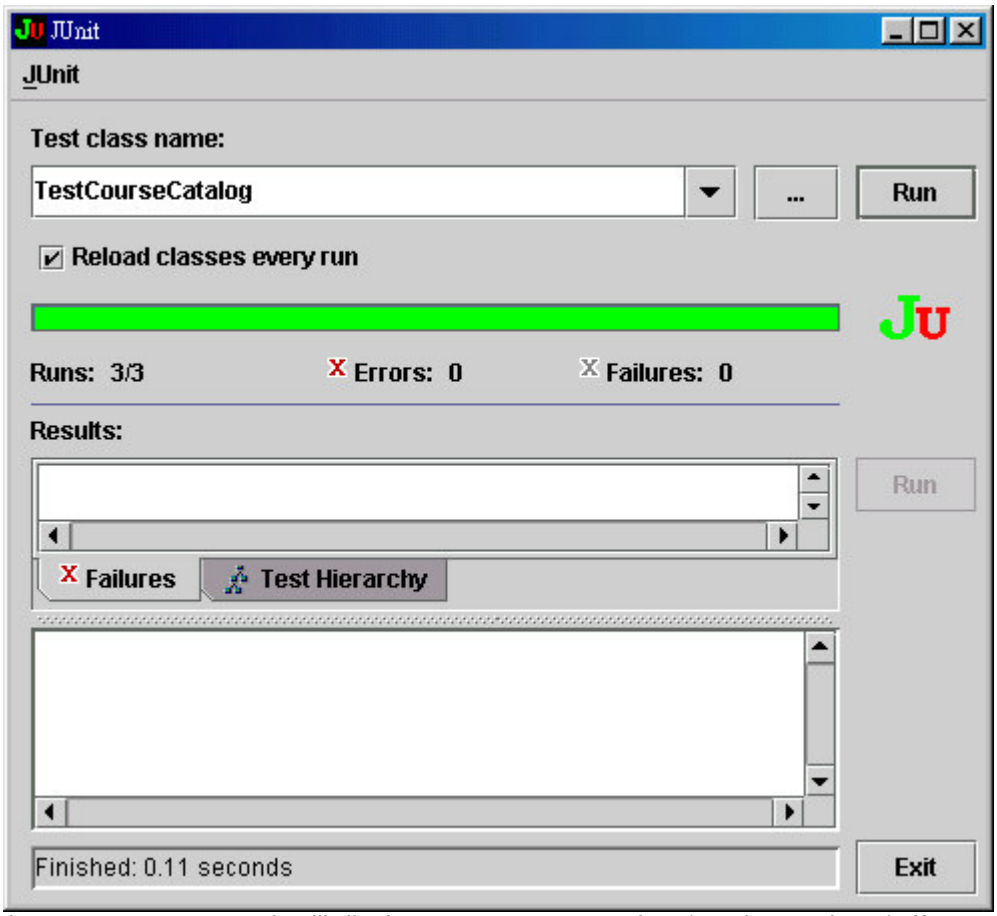
testAdd, testAddTwo and testRemove all need to create a CourseCatalog object. We have put this statement into the setUp method. As we will see, before each test case is run, this setUp method will be called.

Because the cat variable is now initialized in setUp, but it is used in testAdd and etc., it has to be an instance variable (originally it was a local variable in each method). In addition, testAdd and etc. must now be instance methods (originally they were static methods).

testRemove uses assertTrue(X), which checks if X is true. If not, it throws an exception. Just like assertEquals, it is provided by the TestCase class.

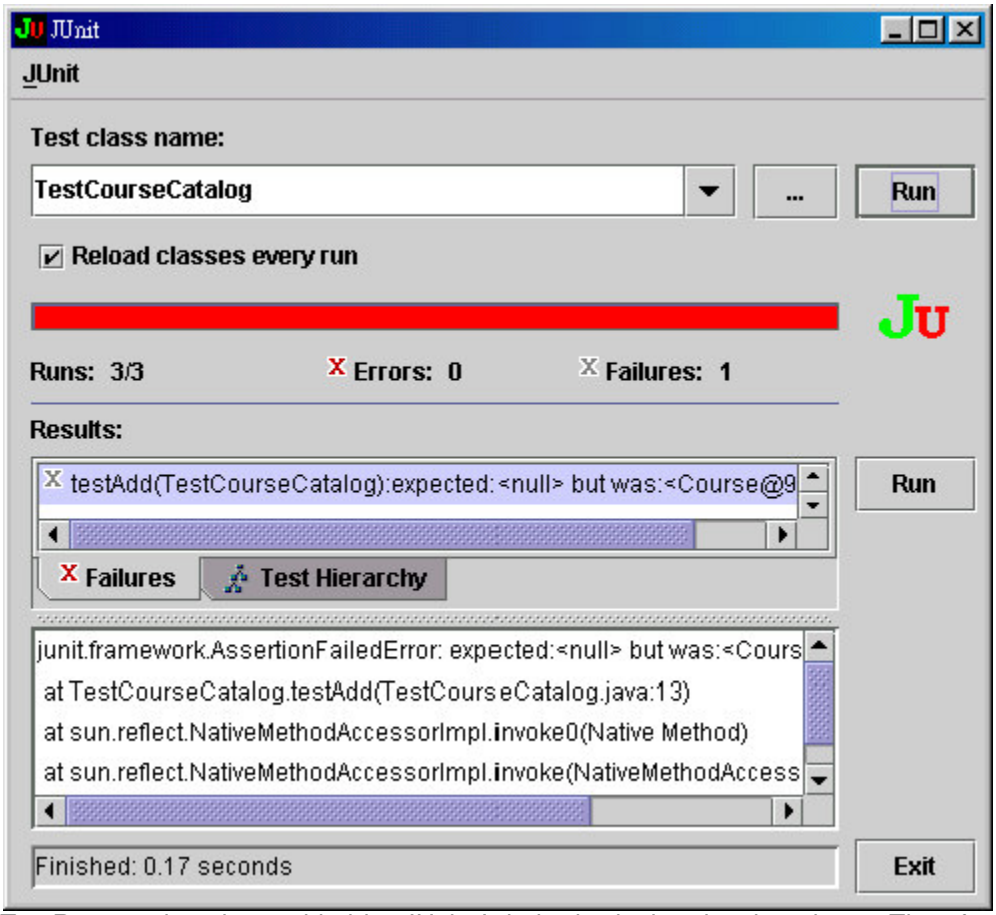
The main method calls a static method named run in the TestRunner class. It will display a window like the one shown below and run the three test cases: testAdd, testAddTwo and testRemove.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



If all three test cases pass, it will display a green progress bar (as shown above). If some test cases fail, it will display a red progress bar and indicate which line in which source file caused the error:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



The TestRunner class is provided by JUnit. It is in the junit.swingui package. Therefore, we need to import junit.swingui.* in order to use it.

How does TestRunner know that we have defined these three test cases (testAdd and etc.)? We call it like this: TestRunner.run(TestCourseCatalog.class). This run method will use the reflection ability of Java to find those methods in TestCourseCatalog that are public, whose names start with "test" and that take no parameters. It takes each such method as a test case. Therefore, now we must declare testAdd and etc. as public.

In order to run testAdd, TestRunner will create a new TestCourseCatalog object, call its setUp method (if any), then call testAdd and finally call its tearDown method (if any). In order to run testAddTwo, TestRunner will create another new TestCourseCatalog object, call its setUp method (if any), then call testAdd and finally call its tearDown method (if any). Therefore, testAdd and testAddTwo are not running on the same TestCourseCatalog object.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

TestRunner are not in the same package as TestCourseCatalog. In order to let TestRunner create a TestCourseCatalog object, the TestCourseCatalog class must be made public.

Usually what are setUp and tearDown used for? If we establish a database connection in setUp, usually we will close the connection in tearDown. Similarly, if we create a temp file in setUp, usually we will delete it in tearDown.

Do we need to unit test all classes

Basically, we should unit test every method in every class in the system, unless that method is too simple to break. For example, for the Course class:

```
class Course {
   String id;
   String title;
   Course(String id, String title, ...) {
      this.id = id;
      this.title = title;
   }
   String getId() {
      return id;
   }
   String getTitle() {
      return title;
   }
   boolean equals(Object obj) {
      if (obj instanceof Course) {
         Course c = (Course)obj;
         return this.id.equals(c.id) && this.title.equals(c.title);
      }
      return false;
   }
}
```

Because its constructor, getId and getTitle methods are very simple, we don't need to test them. What is left is just the equals method:

```
import junit.framework.*;
import junit.swingui.*;
public class TestCourse extends TestCase {
   public void testEquals() {
      Course course1 = new Course("c001", "Java prgoramming");
      Course course2 = new Course("c001", "Java prgoramming");
      Course course3 = new Course("c001", "OO design");
      Course course4 = new Course("c002", "Java prgoramming");
      assertEquals(course1, course2);
      assertTrue(!course1.equals(course3));
      assertTrue(!course1.equals(course4));
   }
   public static void main(String args[]) {
      TestRunner.run(TestCourse.class);
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
```

How to run all the unit tests

Now there are four unit test cases in the system. Three of them test CourseCatalog; One tests Course. We hope to be able to frequently run all the unit test cases in the system in one go. To do that, we may do it this way:

```
public class TestAll {
   public static void main(String args[]) {
      TestRunner.run(TestAll.class);
   }
   public static Test suite() {
      TestSuite suite = new TestSuite();
      suite.addTest(new TestSuite(TestCourseCatalog.class));
      suite.addTest(new TestSuite(TestCourse.class));
      return suite;
   }
}
```

The suite method in the above code creates and returns a TestSuite object, which contains all the test cases in the system. We can add several test cases to a TestSuite. We can also add several TestSuites into another TestSuite. When TestRunner runs a TestSuite, it will actually run all the test cases in the TestSuite.

The main method in the code above calls TestRunner.run(TestAll.class). As mentioned above, the run method will find those methods in the TestAll class that are public, whose names start with "test" and that take no parameters and then take them as test cases. However, before performing this searching, it will first check if TestAll has a static method named "suite". If yes, it will call this suite method and use the TestSuite it returns instead of performing the searching.

If we create a new class such as TimeTable, we need to create the corresponding TestTimeTable to unit test TimeTable. At that time, we need to add a line to TestAll (very easy to forget. Be careful):

```
public class TestAll {
   public static void main(String args[]) {
      TestRunner.run(TestAll.class);
   }
   public static Test suite() {
      TestSuite suite = new TestSuite();
      suite.addTest(new TestSuite(TestCourseCatalog.class));
      suite.addTest(new TestSuite(TestCourse.class));
      suite.addTest(new TestSuite(TestTimeTable.class));
      return suite;
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

When to run the unit tests

When to run the unit tests? For example:

- After we have just finished writing or changing CourseCatalog, we should run the main in TestCourseCatalog.
- After we have just finished writing or changing Course, we should run the main method in TestCourse.
- When we are about to run all the acceptance tests that have passed, we should run the main method in TestAll first.
- When we have free time, we should run the main method in TestAll.

Keep the unit tests updated

Suppose that we have added a new method like getCount in CourseCatalog. We should add one or more test cases in TestCourseCatalog to test it, e.g.,:

```
public class TestCourseCatalog extends TestCase {
   CourseCatalog cat;
   ...
   public void testGetCountOnEmptyCatalog() {
      assertEquals(cat.getCount(), 0);
   }
   public void testGetCount() {
      Course course1 = new Course("c001", "Java prgoramming", ...);
      Course course2 = new Course("c002", "Java prgoramming", ...);
      cat.add(course1);
      cat.add(course2);
      assertEquals(cat.getCount(), 2);
   }
}
```

Use unit tests to prevent the same bug from striking again

When we find a bug in our code, e.g., the remove(Course course) method in CourseCatalog should only delete this particular course object, but it incorrectly deletes all the other courses with the same titles as this one. After finding this bug, we shouldn't rush to fix it. Instead, we should add the following test case in TestCourseCatalog:

```
public class TestCourseCatalog extends TestCase {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    CourseCatalog cat;
    ...
    public void testRemoveKeepOthersWithSameTitle() {
       Course course1 = new Course("c001", "Java prgoramming", ...);
       Course course2 = new Course("c002", "Java prgoramming", ...);
       cat.add(course1);
       cat.add(course2);
       cat.remove(course1);
       assertEquals(cat.findCourseWithId(course2.getId()), course2);
    }
}
```

Of course, currently this test case will fail. It serves as a bug report, telling us that there is a bug in our code. An ordinary bug report can be put aside and ignored. In contrast, we have to run all the unit tests. When faced with the red progress bar in TestRunner, we will never forget this bug.

Another more important effect of this unit test is, if in the future we introduce the same bug into the remove method, this unit test will immediately tell us something is wrong.

How to test if an exception is thrown

We can call remove to delete a Course from CourseCatalog. If there is no such a Course in CourseCatalog, what should remove do? It can do nothing, return an error code or throw an exception. Now, let's assume that it should throw a CourseNotFoundException:

```
class CourseNotFoundException extends Exception {
}
class CourseCatalog {
   void remove(Course course) throws CourseNotFoundException {
      ...
   }
}
```

Accordingly, we need to test if it is really doing that. Therefore, in TestCourseCatalog we add a test case:

```
public class TestCourseCatalog extends TestCase {
   CourseCatalog cat;
   ...
   public void testRemove() {
      ...
   }
   public void testRemoveNotFound() {
      Course course1 = new Course("c001", "Java prgoramming", ...);
      Course course2 = new Course("c002", "OO design", ...);
      cat.add(course1);
      cat.remove(course2); //how to test if it throws a CourseNotFoundException?
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

The question is, how to test if cat.remove(course2) really throws a CourseNotFoundException? We may do it this way:

```
public class TestCourseCatalog extends TestCase {
   ...
   public void testRemoveNotFound() {
      Course course1 = new Course("c001", "Java prgoramming", ...);
      Course course2 = new Course("c002", "OO design", ...);
      cat.add(course1);
      try {
         cat.remove(course2);
         assertTrue(false);
      } catch (CourseNotFoundException e) {
      }
   }
}
```

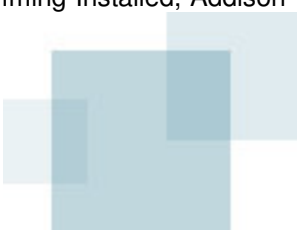
If it doesn't throw any exception, assertTrue(false) will be executed and will make the test fail. If the code does throw a CourseNotFoundException, it will be caught and then nothing more is done. So the test will be considered to have succeeded. If the code throws an exception other than a CourseNotFoundException, the exception will propagate to JUnit. When JUnit receives an exception, it considers that the test has failed.

Instead of calling assertTrue(false), we can call the fail function provided by JUnit:

```
public void testRemoveNotFound() {
   ...
   try {
      cat.remove(course2);
      fail(); //It is the same as assertTrue(false).
   } catch (CourseNotFoundException e) {
   }
}
```

References

- http://www.junit.org.
- http://c2.com/cgi/wiki?UnitTest.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.



Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Problems

1. Unit test the findCoursesWithTitle method in CourseCatalog.

2. If we use the add method in CourseCatalog to add a Course object but there is another Course object in CourseCatalog with the same ID, the add method will throw a CourseDuplicateException. Unit test this behavior.

3. Add a clear method to CourseCatalog. It will delete all the Course objects in CourseCatalog. Unit test this clear method.

Hints

1. No hint for this one.

2. No hint for this one.

3. In order to test this clear method, you may need to add a new method in CourseCatalog.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Unit test the findCoursesWithTitle method in CourseCatalog.

```
public class TestCourseCatalog extends TestCase {
      ...
   public void testFindCoursesWithTitle() {
      Course course1 = new Course("c001", "Java prgoramming", ...);
      Course course2 = new Course("c002", "OO design", ...);
      Course course3 = new Course("c003", "Java Programming", ...);
      cat.add(course1);
      cat.add(course2);
      cat.add(course3);
      Course courses[] = cat.findCoursesWithTitle(course1.getTitle());
      assertEquals(courses.length, 2);
      assertEquals(courses[0], course1);
      assertEquals(courses[1], course3);
   }
}
```

2. If we use the add method in CourseCatalog to add a Course object but there is another Course object in CourseCatalog with the same ID, the add method will throw a CourseDuplicateException. Unit test this behavior.

```
public class TestCourseCatalog extends TestCase {
      ...
   public void testAddDup() {
      Course course1 = new Course("c001", "Java prgoramming", ...);
      Course course2 = new Course("c002", "OO design", ...);
      cat.add(course1);
      try {
         cat.add(course2);
         fail();
      } catch (CourseDuplicateException e) {
      }
   }
}
```

3. Add a clear method to CourseCatalog. It will delete all the Course objects in CourseCatalog. Unit test this clear method.

```
public class TestCourseCatalog extends TestCase {
   public void testClear() {
      Course course1 = new Course("c001", "Java prgoramming", ...);
      Course course2 = new Course("c002", "OO design", ...);
      cat.add(course1);
      cat.add(course2);
      cat.clear();
      assertEquals(cat.countCourses(), 0);
   }
}
class CourseCatalog {
   ...
   int countCourses() {
      ...
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 13

Test Driven Development

Implementing a user story on student enrollment

Suppose that we are working on a student management system and that currently we are working on a user story on student enrollments. A student enrolls in a certain course by paying the course fee specified in the course. The details of the enrollment must be recorded (e.g., student id, course code, date, handled by whom, etc.). The details of the payment must be recorded (e.g., amount, cash or credit card, etc.). The student and the course must have been registered in the system. He can enroll only if there is at least one free seat in the course. The number of seats in a course is specified in the course. A seat is taken when someone enrolls in it. A seat is reserved if someone makes a reservation. The reservation is usually maintained for 24 hours but can be changed by the users. A course may consist of a number of "modules". For example, a Java programming course may consist of a "Java Programming Language" module and a "JDBC" module. Each module is just like a course. However, some modules can not be enrolled in isolation. That is, you must enroll in the whole course in order to enroll in that module. A student can enroll in the whole course only if all the modules have at least one free seat.

Suppose that in the system there are some useful classes:

```
class Student {
   ...
}
class Course {
   ...
   Course[] getModules() { ... }
}
class Reservation {
   Date reserveDate;
   int daysReserved;
   ...
}
class StudentSet {
   ...
}
class CourseSet {
   ...
}
class ReservationSet {
   Reservation[] getReservationsFor(String courseCode) { ... }
   ...
}
```

We figure that to implement this story, we should create an Enrollment class and an

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

EnrollmentSet class. The various checking can be done before an Enrollment is added to the database:

```
class Enrollment {
   ...
}
class EnrollmentSet {
   void add(Enrollment enrollment) {
      assertValid(enrollment);
      addToDB(enrollment);
   }
   void assertValid(Enrollment enrollment) {
      ...
   }
   void addToDB(Enrollment enrollment) {
      ...
   }
}
```

So, we will continue to work on them:

```
class Enrollment {
   String studentId;
   String courseCode;
   Date enrolDate;
   Payment payment;
}
class Payment {
   int amount;
   Payment(int amount) {
      ...
   }
}
class CashPayment extends Payment {
   CashPayment(int amount) {
      ...
   }
}
class CreditCardPayment extends Payment {
   String cardNo;
   String nameOnCard;
   Date expiryDate;
   CreditCardPayment(String cardNo,  String nameOnCard, Date expiryDate,  int
amount) {
      ...
   }

}
class EnrollmentSet {
   StudentSet studentSet;
   CourseSet courseSet;
   ReservationSet reservationSet;

   EnrollmentSet(StudentSet studentSet, CourseSet courseSet, ReservationSet
reservationSet) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        this.studentSet = studentSet;
        this.courseSet = courseSet;
        this.reservationSet = reservationSet;
    }
    void add(Enrollment enrollment) {
        assertValid(enrollment);
        addToDB(enrollment);
    }
    void assertValid(Enrollment enrollment) {
        assertStudentExists(enrollment.getStudentId());
        assertCourseExists(enrollment.getCourseCode());
        assertPaymentCorrect(enrollment);
        assertHasFreeSeat(enrollment);
        assertCanEnrolInIsolation(enrollment);
    }
}
```

It is not finished yet. We still have to write the assertStudentExists, assertCourseExists, assertPaymentCorrect, assertHasFreeSeat, assertCanEnrollInIsolation and addToDB methods. Let's implement them one by one. Implement assertStudentExists first:

```
class EnrollmentSet {
    ...
    void assertStudentExists(String studentId) {
        studentSet.assertStudentExists(studentId);
    }
}
```

This is easy. Let's assume that the StudentSet class already has such an assertStudentExists method. So this method is done. Similarly, implement assertCourseExists:

```
class EnrollmentSet {
    ...
    void assertCourseExists(String courseCode) {
        courseSet.assertCourseExists(courseCode);
    }
}
```

Next, implement assertPaymentCorrect:

```
class EnrollmentSet {
    ...
    void assertPaymentCorrect(Enrollment enrollment) {
        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (enrollment.getPayment().getAmount()!=course.getFee()) {
            throw new IncorrectPaymentException();
        }
    }
}
```

This is not hard. Next, implement assertHasFreeSeat:

```
class EnrollmentSet {
    ...
    void assertHasFreeSeat(Enrollment enrollment) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (course.getNoSeats() <= getNoEnrollments(course)+getNoActiveReservations
(course)) {
            throw new CourseFullException();
        }
    }
}
```

This is getting difficulty: This method requires us to write two other methods: getNoEnrollments and getNoActiveReservations. So we have to write them one by one:

```
class EnrollmentSet {
    Connection conn;
    ...
    EnrollmentSet(
        Connection conn,
        StudentSet studentSet,
        CourseSet courseSet,
        ReservationSet reservationSet) {
        this.conn = conn;
        this.studentSet = studentSet;
        this.courseSet = courseSet;
        this.reservationSet = reservationSet;
    }
    int getNoEnrollments(Course course) {
        PreparedStatement st =
            conn.prepareStatement("select count(*) from enrollments where
courseCode=?");
        try {
            st.setString(1, course.getCode());
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        } finally {
            st.close();
        }
    }
}
```

Then:

```
class EnrollmentSet {
    ...
    int getNoActiveReservations(Course course) {
        return reservationSet.getNoActiveReservations(course.getCode());
    }
}
```

Suppose that there is no getNoActiveReservations method in the ReservationSet class. It means we need to add it:

```
class ReservationSet {
    Reservation[] getReservationsFor(String courseCode) { ... }
    int getNoActiveReservations(String courseCode) {
        int activeCount = 0;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      Reservation reservations[] = getReservationsFor(courseCode);
      for (int i = 0; i < reservations.length; i++) {
         if (reservations[i].isActive()) {
            activeCount++;
         }
      }
      return activeCount;
   }
}
```

In turn we have to add an isActive method to the Reservation class:

```
class Reservation {
   Date reserveDate;
   int daysReserved;
   boolean isActive() {
      Date today = new Date();
      return getLastReservedDate().before(today);
   }
   Date getLastReservedDate() {
      GregorianCalendar lastReservedDate = new GregorianCalendar();
      lastReservedDate.setTime(reserveDate);
      lastReservedDate.add(GregorianCalendar.DATE, daysReserved);
      return lastReservedDate.getTime();
   }
}
```

Next, implement assertCanEnrolInIsolation:

```
class EnrollmentSet {
   ...
   void assertCanEnrolInIsolation(Enrollment enrollment) {
      Course course = courseSet.getByCode(enrollment.getCourseCode());
      if (course.isModule() && !course.canEnrolInIsolation()) {
         throw new CannotEnrolInIsolation();
      }
   }
}
```

Let's assume the Course class already has the isModule and canEnrolInIsolation methods:

```
class Course {
   boolean isModule() { ... }
   boolean canEnrolInIsolation() { ... }
}
```

Finally, implement the addToDB method. Let's assume that the enrollments table is like this:

```
CREATE TABLE enrollments (
   courseCode VARCHAR(50),
   studentId VARCHAR(50),
   enrolDate DATE,
   amount INT,
   paymentType VARCHAR(20),
   cardNo VARCHAR(20),
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    expiryDate DATE,
    nameOnCard VARCHAR(50)
);
```

The addToDB method is:

```
class EnrollmentSet {
   private void addToDB(Enrollment enrollment) {
      PreparedStatement st =
         conn.prepareStatement(
            "insert into enrollments values (?,?,?,?,?,?,?,?)");
      try {
         st.setString(1, enrollment.getCourseCode());
         st.setString(2, enrollment.getStudentId());
         st.setDate(3, new Date(enrollment.getEnrolDate().getTime()));
         Payment payment = enrollment.getPayment();
         st.setInt(4, payment.getAmount());
         if (payment instanceof CashPayment) {
            st.setString(5, "Cash");
            st.setNull(6, Types.VARCHAR);
            st.setNull(7, Types.DATE);
            st.setNull(8, Types.VARCHAR);
         } else if (payment instanceof CreditCardPayment) {
            CreditCardPayment creditCardPayment =
               (CreditCardPayment) payment;
            st.setString(5, "CreditCard");
            st.setString(6, creditCardPayment.getCardNo());
            st.setDate(
               7,
               new Date(creditCardPayment.getExpiryDate().getTime()));
            st.setString(8, creditCardPayment.getNameOnCard());
         }
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
}
```

We have finished writing all the code. We have written about 150 lines of code without running any of it. At this moment, the biggest question on our mind is: Is this code correct? In fact, when we were writing this code, this worry had been accumulating. The more complicated the code was (e.g., addToDB, getNoEnrollments, isActive), the more worried we were. The more code we wrote, the more worried we were. For example, when we found that we needed to write two other methods in order to implement one method, we got more worried. Our accumulated worry can only be addressed by test running the code. How to test the code?

How to test the code just written

If we had a GUI making use of this code, we would be much tempted to just test run the

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

system, input some data manually then inspect the database to see if it is working. However, this is no good because the test is not automated and will not be run frequently. For example, if someone changes the code and introduces a bug by accident, if he doesn't test run the application again or just test it in a very simple way, he may not notice anything wrong. Therefore, a much better way is to have automated unit tests. How to unit test say the add method of EnrollmentSet? We may test adding a valid enrollment first:

```
class EnrollmentSetTest extends TestCase {
   testAddValidEnrollment() {
      Connection conn = ...;
      try {
         EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
         enrollmentSet.deleteAll();
         Enrollment enrollment =
            new Enrollment(
               "s001",
               "c001",
               new Date(),
               new CashPayment(100));
         enrollmentSet.add(enrollment);
         assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
      } finally {
         conn.close();
      }
   }
}
```

This test is not complete yet. Several things are missing: the database connection must be setup, we must add the student "s001" to the database, we must add the course "c001" to the database. Let's create the database connection first. Let's assume we are using a postgreSQL test database named "testdb" on our own computer:

```
class EnrollmentSetTest extends TestCase {
   testAddValidEnrollment() {
      Class.forName("org.postgresql.Driver");
      Connection conn =
         DriverManager.getConnection(
            "jdbc:postgresql://localhost/testdb",
            "username",
            "password");
      try {
         EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
         enrollmentSet.deleteAll();
         Enrollment enrollment =
            new Enrollment(
               "s001",
               "c001",
               new Date(),
               new CashPayment(100));
         enrollmentSet.add(enrollment);
         assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
      } finally {
         conn.close();
      }
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
```

Next, add the student "s001" to the database. To do that, we need to create such a Student object first. This is not as easy as it seems because the Student class has a constructor that takes a lot of parameters:

```
class Student {
   String studentId;
   String idCardNo;
   String name;
   boolean isMale;
   Date dateOfBirth;
   ContactInfo address;
   Employment employment;

   Student(
      String studentId,
      String idCardNo,
      String name,
      boolean isMale,
      Date dateOfBirth,
      ContactInfo address,
      Employment employment) {
      ...
   }
}
class ContactInfo {
   String email;
   String telNo;
   String faxNo;
   String postalAddress;

   public ContactInfo(String email, String telNo, String faxNo, String
postalAddress) {
      ...
   }
}
public class Employment {
   String companyName;
   String jobTitle;
   int yearsOfService;
   ContactInfo contactInfo;
   public Employment(
      String companyName,
      String jobTitle,
      int yearsOfService,
      ContactInfo contactInfo) {
      ...
   }
}
```

Therefore, the create such a student, the code will be like:

```
class EnrollmentSetTest extends TestCase {
   testAddValidEnrollment() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        Class.forName("org.postgresql.Driver");
        Connection conn =
           DriverManager.getConnection(
              "jdbc:postgresql://localhost/testdb",
              "username",
              "password");
        try {
           StudentSet studentSet = new StudentSet(conn);
           studentSet.deleteAll();
           Student student =
              new Student(
                 "s001",
                 "9/29741/8",
                 "Paul Chan",
                 true,
                 new Date(),
                 new ContactInfo(
                    "paul@yahoo.com",
                    "123456",
                    "123457",
                    "postal address"),
                 new Employment(
                    "ABC Ltd.",
                    "Manager",
                    2,
                    new ContactInfo(
                       "info@abc.com",
                       "111000",
                       "111001",
                       "ABC postal address")));
           studentSet.add(student);
           EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
           enrollmentSet.deleteAll();
           Enrollment enrollment =
              new Enrollment(
                 "s001",
                 "c001",
                 new Date(),
                 new CashPayment(100));
           enrollmentSet.add(enrollment);
           assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
        } finally {
           conn.close();
        }
     }
  }
```

This is terrible. All we need is just his student id and his presence. We don't need all the other information such his name, id card number, date of birth, contact information, employment information and etc. Can we set them to null?

```
Student student = new Student("s001", null, null,true, null, null, null);
studentSet.add(student);
```

No. Because when the student is added to the database, various validity checking will be performed and the data in say the contact information will also be saved to the database. If

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

you set the reference to null, the program will crash. So, we have to bite the bullet to input all the data.

However, the horror doesn't stop here. The test is still incomplete. We still need to add the course "c001" to the database. We will face a similar difficulty: All we need is the course code, its fee, number of seats. We don't need other information such as course name, course content outline, instructor, schedule, and lots of other information.

This is getting out of hand. We must find a way to solve these problems.

Solve these problems by writing tests first

Let's throw away all this code and implement the story in a totally different way. First, let's write the unit test (yes, we don't write any other code yet!). The test is simple: we need to have an enrollment and an enrollment set, then add the enrollment to the enrollment set:

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.add(enrollment);
   }
}
```

Note that at the moment we have no Enrollment class nor EnrollmentSet class yet. In addition, the constructor for the enrollment object takes two parameters only: the student id and course code. Why these parameters? Simply because they are the parameters that come to me at the moment and I don't bother to think of other parameters. Similarly, I can't think of any parameters for the constructor for the enrollment set, so we will just let it take no parameter.

But how to test if the enrollment is really added to the database? Should we inspect the database:

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.add(enrollment);
      Class.forName("org.postgresql.Driver");
      Connection conn =
         DriverManager.getConnection(
            "jdbc:postgresql://localhost/testdb",
            "username",
            "password");
      ...
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

This is too much code and it is too troublesome to check the database. We must come up with some way to replace the database. For example, all the EnrollmentSet needs is to write the Enrollment to somewhere. That somewhere is not necessarily a database. Let's call this EnrollmentStorage. We will create an EnrollmentStorage in the test, let the EnrollmentSet use it and finally check if it has really added the Enrollment to that EnrollmentStorage:

```
interface EnrollmentStorage {
   void add(Enrollment enrollment);
}
class EnrollmentStorageForTest implements EnrollmentStorage {
   Enrollment enrollmentStored;
   void add(Enrollment enrollment) {
      enrollmentStored = enrollment;

   }
};

class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      EnrollmentStorageForTest storage = new EnrollmentStorageForTest();
      enrollmentSet.setStorage(storage);
      enrollmentSet.add(enrollment);
      assertTrue(storage.enrollmentStored==enrollment);
   }
}
```

Of course, at the moment the unit test will not even compile. Let's create the Enrollment class and EnrollmentSet class. However, we will just write enough code that is necessary to make the unit test compile, so that we can run the unit test ASAP:

```
class Enrollment {
   Enrollment(String studentId, String courseCode) {
   }
}
class EnrollmentSet {
   void setStorage(EnrollmentStorage storage) {
   }
   void add(Enrollment enrollment) {
   }
}
```

Yes, there is no code in the methods! As we said, we would like to run the unit test ASAP. Now, run the unit test. It fails and we see a red bar in JUnit (as expected). Why run it if we know it is going to fail? We will explain that later. For the moment, simply remember to run a test and see it fail before writing the implementation code. Now, we can go ahead to write the code required to make the test pass:

```
class EnrollmentSet {
   EnrollmentStorage storage;

   void setStorage(EnrollmentStorage storage) {
      this.storage = storage;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
    void add(Enrollment enrollment) {
       storage.add(enrollment);
    }
}
```

This is just three lines of code. We can do it in a minute and now we can run the test again. Now it passes (green bar). We have made a progress. Next, we should write another unit test. For example, test if it can check the student is registered. However, before that, we can simplify the first unit test using an anonymous class to get rid of the EnrollmentStorageForTest whose name is not telling anything and therefore is bothering me:

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      final Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setStorage(new EnrollmentStorage() {
         void add(Enrollment enrollmentToStore) {
            assertTrue(enrollmentToStore == enrollment);
         }
      });
      enrollmentSet.add(enrollment);
   }
}
```

As we have changed the code, run the unit test again to see if the bar is still green. Yes, it is. So, we are safe to go ahead.

Testing if the student is registered

Now, write the test to see if it really checks the student is registered. The test will try to enroll a student who is really registered. But how to tell the EnrollmentSet that this student is registered? Adding to StudentSet is a lot of work as we have seen. Just like replacing EnrollmentStorage for the database, we can let it use a StudentRegistryChecker instead of a StudentSet:

```
interface StudentRegistryChecker {
   boolean isRegistered(String studentId);
}
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      ...
   }
   void testStudentRegistered() {
      final Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
         boolean isRegistered(String studentId) {
            return studentId.equals("s001");
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        });
        enrollmentSet.setStorage(new EnrollmentStorage() {
           void add(Enrollment enrollmentToStore) {
              assertTrue(enrollmentToStore == enrollment);
           }
        });
        enrollmentSet.add(enrollment);
     }
  }
```

However, it is very similar to testAddEnrollment. In particular, the code to test if the enrollment is added to the storage. Usually it makes little sense to test the same behavior twice. testStudentRegistered should just test the checking behavior before the enrollment is added to the storage. This suggests that EnrollmentSet should have an assertValid method so that we can call it instead of the add method:

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      ...
   }
   void testStudentRegistered() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
         boolean isRegistered(String studentId) {
            return studentId.equals("s001");
         }
      });
      enrollmentSet.assertValid(enrollment);
   }
}
```

Now the test is much cleaner. Of course, we need to test if add really calls assertValid before sending the enrollment to the storage. But for the moment, let's put it onto our todo list:

TODO
Test if add really calls assertValid.

We will make the current test (testStudentRegistered) pass first. So, create the necessary methods so that the test will compile:

```
class EnrollmentSet {
   EnrollmentStorage storage;

   void setStorage(EnrollmentStorage storage) {
      this.storage = storage;
   }
   void add(Enrollment enrollment) {
      storage.add(enrollment);
   }
   void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
   }
   void assertValid(Enrollment enrollment) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
}
```

Now run the two tests. Oops! We get a green bar! This is because at the moment it will treat any enrollment as valid. So our really valid enrollment is also considered valid without any additional effort. Of course, we know that this assertValid should really do something like using the StudentRegistryChecker to lookup the student. However, before we do it, we must have a failing test first. So, let's try to check an unregistered student. We expect that it should throw a StudentNotFoundException:

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      ...
   }
   void testStudentRegistered() {
      ...
   }
   void testStudentUnregistered() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
         boolean isRegistered(String studentId) {
            return false;
         }
      });
      try {
         enrollmentSet.assertValid(enrollment);
         fail();
      } catch (StudentNotFoundException e) {
      }
   }
}
```

It won't compile because StudentNotFoundException is undefined yet. Create an empty class so that the test will compile:

```
class StudentNotFoundException extends RuntimeException {
}
```

Now run all the tests. It fails. This is expected. Let's write the implementation code to make it pass:

```
class EnrollmentSet {
   EnrollmentStorage storage;
   StudentRegistryChecker studentRegistryChecker;

   void setStorage(EnrollmentStorage storage) {
      this.storage = storage;
   }
   void add(Enrollment enrollment) {
      storage.add(enrollment);
   }
   void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
      this.studentRegistryChecker = registryChecker;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
    void assertValid(Enrollment enrollment) {
       if (studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
          throw new StudentNotFoundException();
       }
    }
}
```

However, we need to have a getStudentId method in Enrollment. Usually we need to write a failing test for this method before we write the method, but it is such a simple getter method (see the code below) that we believe that there is no need to test it. So we go ahead to write the method:

```
class Enrollment {
   String studentId;

   Enrollment(String studentId, String courseCode) {
      this.studentId = studentId;
   }
   String getStudentId() {
      return studentId;
   }
}
```

As a side note, now it is the first time that we make use of the student id passed to the constructor of the enrollment. It means it has been useless until now because the tests didn't require the student id.

Now run the tests. Surprisingly, both testStudentRegistered and testStudentUnregistered fail! The bug is very likely to be in the code (about 10 lines) we just added. The most likely place is the assertValid method:

```
void assertValid(Enrollment enrollment) {
   if (studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
      throw new StudentNotFoundException();
   }
}
```

Obviously this is throwing an exception when the student is registered. This is wrong. So, let's fix it:

```
void assertValid(Enrollment enrollment) {
   if (!studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
      throw new StudentNotFoundException();
   }
}
```

Now run the tests again. We see a green bar. From this incident we can see the benefit of "test a little, code a little": if we introduce a bug, it is detected right away and it can be easily located. Compare this with our original approach where we wrote about 150 lines of code that remained untested for hours.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Testing if there is a free seat

Now, we will pick another thing to test. For example, we may test if it checks if the course exists. However, it should be very similar to the checking of the student, so there is little that we can learn by implementing it. In general, we should pick something that is not too hard to do but we can still learn from it. In this case, we may pick the free seat checking. However, it is still too large. We can ignore the part about reservations for now and concentrate on the seats that are really taken. First, add the part about reservations to our todo list:

TODO
Test if add really calls assertValid.
Test if reservations are considered.

Now, let's test if it is considering the seats taken by enrollments. In the test we will set the total number of seats to say two, make the enrollment set believe that two students have enrolled, then try to validate a new enrollment. If it works, it should throw a ClassFullException:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsTaken() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      //How to make enrollmentSet believe two students have enrolled in c001?
      try {
         enrollmentSet.assertValid(enrollment);
         fail();
      } catch (ClassFullException e) {
      }
   }
}
```

The question is: How to make enrollmentSet believe two students have enrolled in course c001? We may add two enrollments there, but it is quite troublesome because we will have to create two enrollments, make sure they are valid and then implement reading enrollments from the storage. This is too much work. So, let's use a EnrollmentCounter instead:

```
interface EnrollmentCounter {
   int getNoEnrollments(String courseCode);
}
class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsTaken() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
         int getNoEnrollments(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      try {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            enrollmentSet.assertValid(enrollment);
            fail();
         } catch (ClassFullException e) {
         }
      }
   }
```

But we will need to make enrollmentSet believe that the total number of seats for course c001 is two. To do that, we may let it use a CourseLookup to find the Course object and then get the class size in it. But having to create a Course object is still too much work. A simpler way is to let it use a ClassSizeGetter:

```
public interface ClassSizeGetter {
   public int getClassSize(String courseCode);
}
class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsTaken() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
         int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
         int getNoEnrollments(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      try {
         enrollmentSet.assertValid(enrollment);
         fail();
      } catch (ClassFullException e) {
      }
   }
}
```

The test won't compile yet. Create ClassFullException, the setClassSizeGetter and the setEnrollmentCounter methods so that the test compiles:

```
class ClassFullException extends RuntimeException {
}
class EnrollmentSet {
   EnrollmentStorage storage;
   StudentRegistryChecker studentRegistryChecker;

   void assertValid(Enrollment enrollment) {
      ...
   }
   void setClassSizeGetter(ClassSizeGetter sizeGetter) {
   }
   void setEnrollmentCounter(EnrollmentCounter counter) {
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Now run the tests. It fails. But the error is unusual: It is a NullPointerException. By further inspection, it is because the studentRegistryChecker in the EnrollmentSet is null. Right, it is checking if the student s001 is registered, but we haven't setup a StudentRegistryChecker. To solve this problem, we could copy the code to setup a StudentRegistryChecker:

```
void testAllSeatsTaken() {
   Enrollment enrollment = new Enrollment("s001", "c001");
   EnrollmentSet enrollmentSet = new EnrollmentSet();
   enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
      boolean isRegistered(String studentId) {
         return false;
      }
   });
   enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
      int getClassSize(String courseCode) {
         return courseCode.equals("c001") ? 2 : 0;
      }
   });
   enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
      int getNoEnrollments(String courseCode) {
         return courseCode.equals("c001") ? 2 : 0;
      }
   });
   try {
      enrollmentSet.assertValid(enrollment);
      fail();
   } catch (ClassFullException e) {
   }
}
```

But this is too much work. In fact, we are testing the student registration checking again. As we have said before, it is generally not good to test the same thing again. This is suggesting that EnrollmentSet should have an assertHasSeat method so that we can test this method alone. Of course, we will also need to test that the add method calls assertHasSeat before adding the enrollment to the storage. Accordingly, the assertValid method should be called assertStudentRegistered instead:

```
class EnrollmentSet {
   ...
   void assertStudentRegistered(Enrollment enrollment) {
      ...
   }
   void assertHasSeat(Enrollment enrollment) {
   }
}
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      ...
   }
   void testStudentRegistered() {
      ...
      enrollmentSet.assertStudentRegistered(enrollment);
      ...
   }
   void testStudentUnregistered() {
      ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        enrollmentSet.assertStudentRegistered(enrollment);
        ...
    }
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoEnrollments(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat(enrollment);
            fail();
        } catch (ClassFullException e) {
        }
    }
}
```

Now run the tests again. The current test fails. This is expected. Let's write the implementation code to make the test pass:

```
class EnrollmentSet {
    ...
    void assertStudentRegistered(Enrollment enrollment) {
        ...
    }
    void assertHasSeat(Enrollment enrollment) {
        String courseCode = enrollment.getCourseCode();
        if (enrollmentCounter.getNoEnrollments(courseCode)
            >= classSizeGetter.getClassSize(courseCode)) {
            throw new ClassFullException();
        }
    }
}
```

It uses a getCourseCode method of the Enrollment class. As it is just a simple getter, we will implement it without writing a failing test for it:

```
class Enrollment {
    String studentId;
    String courseCode;

    Enrollment(String studentId, String courseCode) {
        this.studentId = studentId;
        this.courseCode = courseCode;
    }
    String getStudentId() { ... }
    String getCourseCode() {
        return courseCode;
    }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

This is the first time that we use the courseCode passed to the Enrollment's constructor.

Now, run the tests. They pass. To be sure that it is really working, we should test the case when there are indeed free seats. We can basically copy the code of testAllSeatsTaken. We simply change the number of enrollments from two to one and change to expect that it will not throw any exception:

```
public void testHasAFreeSeat() {
   Enrollment enrollment = new Enrollment("s001", "c001");
   EnrollmentSet enrollmentSet = new EnrollmentSet();
   enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
      int getClassSize(String courseCode) {
         return courseCode.equals("c001") ? 2 : 0;
      }
   });
   enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
      int getNoEnrollments(String courseCode) {
         return courseCode.equals("c001") ? 1 : 0;
      }
   });
   enrollmentSet.assertHasSeat(enrollment);
}
```

Now run the tests. They pass.

Testing if enrollments in the parent course are considered

But we haven't finished testing it yet. We still need to test a very interesting behavior: When looking for a free seat, it should not only consider the course itself, but also its parent course if it is a module. For example, if c002 is a module of c001 and there are three enrollments for c001 and two enrollments for c002, if the class size of c002 is five, it is already full. At the moment, we are using the getNoEnrollments method of EnrollmentCounter to find the number of enrollments. For c002 in the above example, will it return two or five? If it returns two, EnrollmentSet must check the parent of c002. If it returns five, there is nothing EnrollmentSet needs to do. To take the easier route, we will assume that it will return five. To make it more explicit, let rename getNoEnrollments to getNoSeatsTaken (Enrollment is just one of the possible reasons for a seat in a module being taken):

```
interface EnrollmentCounter {
   int getNoSeatsTaken(String courseCode);
}
```

Just to be safe, run the tests again and they should pass.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Testing if reservations are considered

Next, pick a task from our todo list:

TODO
Test if add really calls assertStudentRegistered and assertHasSeat.
Test if reservations are considered.

Let's do the second task. We will setup a ReservationCounter to return the reservations for a given course (including its parent if it is a module). We will setup the context so that some seats are taken and the rest are reserved. This is to test that assertHasSeat will consider both enrollments and reservations. It is very similar to testAllSeatsTaken:

```
public interface ReservationCounter {
   public int getNoSeatsReserved(String courseCode);
}
class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsReservedOrTaken() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
         int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 3 : 0;
         }
      });
      enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
         int getNoSeatsTaken(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      enrollmentSet.setReservationCounter(new ReservationCounter() {
         int getNoSeatsReserved(String courseCode) {
            return courseCode.equals("c001") ? 1 : 0;
         }
      });
      try {
         enrollmentSet.assertHasSeat(enrollment);
         fail();
      } catch (ClassFullException e) {
      }
   }
}
```

Create an empty setReservationCounter method to make test compile:

```
class EnrollmentSet {
   ...
   void setReservationCounter(ReservationCounter counter) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
}
```

Run the tests and the current one fails. This is expected. Write the implementation code so that the test passes:

```
class EnrollmentSet {
   ...
   ReservationCounter reservationCounter;
   ...
   public void assertHasSeat(Enrollment enrollment) {
      String courseCode = enrollment.getCourseCode();
      if (enrollmentCounter.getNoSeatsTaken(courseCode)
         + reservationCounter.getNoSeatsReserved(courseCode)
         >= classSizeGetter.getClassSize(courseCode)) {
         throw new ClassFullException();
      }
   }
   void setReservationCounter(ReservationCounter counter) {
      this.reservationCounter = counter;
   }
}
```

Now run the tests. The current one passes but two previous tests fail: testAllSeatsTaken and testHasAFreeSeat. They both trigger a NullPointerException. This is because in these tests we didn't setup a ReservationCounter for the EnrollmentSet to use. A quick way to fix it is to setup a ReservationCounter in these tests. For example:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsTaken() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
         int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
         int getNoEnrollments(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      enrollmentSet.setReservationCounter(new ReservationCounter() {
         int getNoSeatsReserved(String courseCode) {
            return 0;
         }
      });
      try {
         enrollmentSet.assertHasSeat(enrollment);
         fail();
      } catch (ClassFullException e) {
      }
   }
   public void testHasAFreeSeat() {
      ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        enrollmentSet.setReservationCounter(new ReservationCounter() {
           int getNoSeatsReserved(String courseCode) {
              return 0;
           }
        });
        ...
     }
  }
```

Now run the tests again and they should pass.

Testing if add performs validation

Next we will pick the last task on our todo list:

TODO
Test if add really calls assertStudentRegistered and assertHasSeat.

We can do it like this:

```
class EnrollmentSetTest extends TestCase {
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
         void assertStudentRegistered(Enrollment enrollment) {
            callLog.append("x");
         }
         void assertHasSeat(Enrollment enrollment) {
            callLog.append("y");
         }
      };
      enrollmentSet.setStorage(new EnrollmentStorage() {
         void add(Enrollment enrollmentToStore) {
            callLog.append("z");
         }
      });
      Enrollment enrollment = new Enrollment("s001", "c001");
      enrollmentSet.add(enrollment);
      assertEquals(callLog.toString(), "xyz");
   }
}
```

This test assumes that add will first call assertStudentRegistered, then assertHasSeat and finally add the enrollment to the storage. The StringBuffer callLog is used to log the calls. For it to be accessible to the inner classes, it has to be final. We cannot use a String instead because a String object cannot be changed.

Now run the tests and the current test fails. This is expected because at the moment the add

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

method is not validating the enrollment. Add the code so that it does:

```
class EnrollmentSet {
   void add(Enrollment enrollment) {
      assertStudentRegistered(enrollment);
      assertHasSeat(enrollment);
      storage.add(enrollment);
   }
   ...
}
```

The current test passes, but the first test fails with a NullPointerException:

```
void testAddEnrollment() {
   final Enrollment enrollment = new Enrollment("s001", "c001");
   EnrollmentSet enrollmentSet = new EnrollmentSet();
   enrollmentSet.setStorage(new EnrollmentStorage() {
      void add(Enrollment enrollmentToStore) {
         assertTrue(enrollmentToStore == enrollment);
      }
   });
   enrollmentSet.add(enrollment);
}
```

This is because now add will validate the enrollment but we have not setup the needed objects yet (e.g., StudentRegistryChecker, ClassSizeGetter, etc.). We could setup all these objects but it is too much work. As always we should separate the code that is being tested into a new method so that our tests don't test the same thing. Let's call the new method addToStorage:

```
void testAddEnrollment() {
   final Enrollment enrollment = new Enrollment("s001", "c001");
   EnrollmentSet enrollmentSet = new EnrollmentSet();
   enrollmentSet.setStorage(new EnrollmentStorage() {
      void add(Enrollment enrollmentToStore) {
         assertTrue(enrollmentToStore == enrollment);
      }
   });
   enrollmentSet.addToStorage(enrollment);
}
```

Create an empty addToStorage method:

```
class EnrollmentSet {
   ...
   void addToStorage(Enrollment enrollment) {
   }
}
```

As it is now testing addToStorage instead of testing add, let's rename it from testAddEnrollment to testAddToStorage:

```
void testAddToStorage() {
   final Enrollment enrollment = new Enrollment("s001", "c001");
   EnrollmentSet enrollmentSet = new EnrollmentSet();
   enrollmentSet.setStorage(new EnrollmentStorage() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        void add(Enrollment enrollmentToStore) {
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.addToStorage(enrollment);
}
```

Run the test and expect it to fail. But it doesn't! At the moment addToStorage is not really adding the enrollment to the storage. But the add method of our storage object is never called at all. So the assertTrue inside is simply not executed. This means that this test is not working. That is, it is useless. This is why we need to see a failing test before writing implementation code. If the test indeed fails, it means the test should be working. If it doesn't fail, it means the test is broken. That is, we are testing the test itself with this procedure. Now back to the case at hand. How to correct the test? We can use a call log again:

```
void testAddToStorage() {
    final StringBuffer callLog = new StringBuffer();
    final Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStorage(new EnrollmentStorage() {
        void add(Enrollment enrollmentToStore) {
            callLog.append("x");
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.addToStorage(enrollment);
    assertEquals(callLog.toString(), "x");
}
```

Now run it again and it should fail. This is good. Write the implementation code so that the test passes:

```
class EnrollmentSet {
    ...
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment);
        assertHasSeat(enrollment);
        addToStorage(enrollment);
    }
    void addToStorage(Enrollment enrollment) {
        storage.add(enrollment);
    }
}
```

Now run the tests and they should pass.

Changing Enrollment's constructor breaks all existing tests

Now, our todo list is empty, but it doesn't mean that we have nothing to do. From the user story it is clear that we still need to do many things such as checking the payment, saving the

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

enrollment data to the database and etc. Now, let's test to see if it will check the payment. We will store a payment in an enrollment whose amount is equal to the course fee, call assertPaymentCorrect and expect it to pass:

```
interface CourseFeeLookup {
   int getFee(String courseCode);
}
class EnrollmentSetTest extends TestCase {
   ...
   void testPaymentCorrect() {
      Enrollment enrollment = new Enrollment("s001", "c001", new CashPayment(100));
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
         int getFee(String courseCode) {
            return courseCode.equals("c001") ? 100 : 0;
         }
      });
      enrollmentSet.assertPaymentCorrect(enrollment);
   }
}
```

Note that we are passing a payment object as the third parameter of the constructor for Enrollment. This sounds fine. After all, an enrollment must have a payment. To make the test compile, we need to add this parameter to the constructor:

```
class Enrollment {
   Enrollment(String studentId, String courseCode, Payment payment) {
      ...
   }
}
```

However, this will break all the existing tests:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testAddEnrollment() {
      Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
      ...
   }
   void testStudentRegistered() {
      Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
      ...
      enrollmentSet.assertStudentRegistered(enrollment);
   }
   void testStudentUnregistered() {
      Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
      ...
      enrollmentSet.assertStudentRegistered(enrollment);
      ...
   }
   void testAllSeatsTaken() {
      Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
      ...
      enrollmentSet.assertHasSeat(enrollment);
      ...
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
```

But for all these tests such as testStudentRegistered and testAllSeatsTaken, they don't care about the payment at all. This suggests that they shouldn't work with an Enrollment. They should only work on what they really care. For example, testStudentRegistered should only work with the stuent id. Therefore, we should fix them. However, we do not like having more than one errors in the code. So, comment out the testPaymentCorrect method and keep both constructors for Enrollment for the moment:

```
class Enrollment {
   ...
   Enrollment(String studentId, String courseCode) {
      ...
   }
   Enrollment(String studentId, String courseCode, Payment payment) {
   }
}
```

Note that the new constructor is empty. This looks weird. If it is empty then it is definitely not going to work. There are several options here. We may write the code without a failing test:

```
class Enrollment {
   ...
   Enrollment(String studentId, String courseCode, Payment payment) {
      this.studentId = studentId;
      this.courseCode = courseCode;
      this.payment = payment;
   }
}
```

We may create a new class EnrollmentTest to test it first:

```
class EnrollmentTest extends TestCase {
   void testCreation() {
      Payment payment = ...;
      Enrollment enrollment = new Enrollment("a", "b", payment);
      assertEquals(enrollment.getStudentId(), "a");
      assertEquals(enrollment.getCourseCode(), "b");
      assertTrue(enrollment.getPayment()==payment);
   }
}
```

We may create a test in EnrollmentSetTest that fails until we implement the constructor. Any of these options should be fine. Suppose that in this case we decide to take the last option. However, we can't do it right away because we're in the process of migrating to the new constructor. So, add the task to our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.

Then, change the tests one by one to make sure they don't use the old Enrollment constructor.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Let's consider the first test: testAddToStorage. It needs to pass an Enrollment to addToStorage because addToStorage is required to pass it to the EnrollmentStorage, but the Enrollment object doesn't have to contain meanful data. So, let's update the test to use the new constructor using "null" data:

```
class EnrollmentSetTest extends TestCase {
   public void testAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      final Enrollment enrollment = new Enrollment(null, null, null);
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setStorage(new EnrollmentStorage() {
         public void add(Enrollment enrollmentToStore) {
            callLog.append("x");
            assertTrue(enrollmentToStore == enrollment);
         }
      });
      enrollmentSet.addToStorage(enrollment);
      assertEquals(callLog.toString(), "x");
   }
   ...
}
```

Run the test to make sure it passes. Then consider testStudentRegistered. It should only need the student id instead of an enrollment object:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testStudentRegistered() {
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
         boolean isRegistered(String studentId) {
            return studentId.equals("s001");
         }
      });
      enrollmentSet.assertStudentRegistered("s001");
   }
}
```

This will not compile because currently assertStudentRegistered takes an Enrollment as parameter. We could change it to take the student id as parameter, but this may break other tests. Because we are in the process of migrating from the two-parameter Enrollment constructor to the three-parameter version, we'd like to avoid changing assertStudentRegistered at the same time. Instead, let's copy assetStudentRegistered to create an overloaded version that takes the student id as parameter:

```
class EnrollmentSet {
   ...
   void assertStudentRegistered(Enrollment enrollment) {
      if (!studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
         throw new StudentNotFoundException();
      }
   }
   void assertStudentRegistered(String studentId) {
      if (!studentRegistryChecker.isRegistered(studentId)) {
         throw new StudentNotFoundException();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        }
    }
}
```

After migrating to the new Enrollment constructor, we will get rid of the assertStudentRegistered that takes an enrollment as parameter. So, add this task to our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.
Get rid of assertStudentRegistered that takes an enrollment.

Now run the test again and it should pass. Next consider testStudentUnregistered. It is similar to testStudentRegistered:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testStudentUnregistered() {
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
         boolean isRegistered(String studentId) {
            return false;
         }
      });
      try {
         enrollmentSet.assertStudentRegistered("s001");
         fail();
      } catch (StudentNotFoundException e) {
      }
   }
}
```

Now consider testAllSeatsTaken. It should only need the course id instead of an enrollment object:

```
class EnrollmentSetTest extends TestCase {
   ...
   public void testAllSeatsTaken() {
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
         public int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
         public int getNoSeatsTaken(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      enrollmentSet.setReservationCounter(new ReservationCounter() {
         public int getNoSeatsReserved(String courseCode) {
            return 0;
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        });
        try {
           enrollmentSet.assertHasSeat("c001");
           fail();
        } catch (ClassFullException e) {
        }
     }
}
```

We need to create an overloaded version of assertHasSeat:

```
class EnrollmentSet {
   ...
   void assertHasSeat(Enrollment enrollment) {
      ...
   }
   void assertHasSeat(String courseCode) {
      if (enrollmentCounter.getNoSeatsTaken(courseCode)
         + reservationCounter.getNoSeatsReserved(courseCode)
         >= classSizeGetter.getClassSize(courseCode)) {
         throw new ClassFullException();
      }
   }
}
```

Add a task to delete the old assertHasSeat to our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.
Get rid of assertStudentRegistered that takes an enrollment.
Get rid of assertHasSeat that takes an enrollment.

Run the tests and they should still pass. Similarly, update testHasAFreeSeat to use a course id:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testHasAFreeSeat() {
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
         int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
         }
      });
      enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
         int getNoSeatsTaken(String courseCode) {
            return courseCode.equals("c001") ? 1 : 0;
         }
      });
      enrollmentSet.setReservationCounter(new ReservationCounter() {
         int getNoSeatsReserved(String courseCode) {
            return 0;
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        });
        enrollmentSet.assertHasSeat("c001");
    }
}
```

Run the tests and they should still pass. Similarly, update testAllSeatsReservedOrTaken to use a course id:

```
class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsReservedOrTaken() {
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            public int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 3 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            public int getNoSeatsTaken(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            public int getNoSeatsReserved(String courseCode) {
                return courseCode.equals("c001") ? 1 : 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat("c001");
            fail();
        } catch (ClassFullException e) {
        }
    }
}
```

Run the tests and they should still pass. Now, consider testValidateBeforeAddToStorage. It is only testing the calling sequence. It doesn't need an enrollment object at all. It can happily use a null:

```
class EnrollmentSetTest extends TestCase {
    ...
    public void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            public void assertStudentRegistered(Enrollment enrollment) {
                callLog.append("x");
            }
            public void assertHasSeat(Enrollment enrollment) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        enrollmentSet.add(null);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        assertEquals(callLog.toString(), "xyz");
    }
}
```

Run the tests and they should still pass.

By now we should have eliminated all the calls to the two-parameter version of the Enrollment constructor. Let's delete it. Then run all the tests and they should still pass.

Now, check our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.
Get rid of assertStudentRegistered that takes an enrollment.
Get rid of assertHasSeat that takes an enrollment.

Let's do the second task. First, check if the old version of assertStudentRegistered is still being called by any code at all (some IDEs can do this for you). We find it is still being used at one place:

```
class EnrollmentSet {
   void add(Enrollment enrollment) {
      assertStudentRegistered(enrollment);
      assertHasSeat(enrollment);
      addToStorage(enrollment);
   }
   ...
}
```

That's easy: just change it to use the new version:

```
class EnrollmentSet {
   void add(Enrollment enrollment) {
      assertStudentRegistered(enrollment.getStudentId());
      assertHasSeat(enrollment);
      addToStorage(enrollment);
   }
   ...
}
```

Run the tests but testValidateBeforeAddToStorage fails with a NullPointerException:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
         void assertStudentRegistered(Enrollment enrollment) {
            callLog.append("x");
         }
         void assertHasSeat(Enrollment enrollment) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            callLog.append("y");
         }
      };
      enrollmentSet.setStorage(new EnrollmentStorage() {
         public void add(Enrollment enrollmentToStore) {
            callLog.append("z");
         }
      });
      enrollmentSet.add(null);
      assertEquals(callLog.toString(), "xyz");
   }
}
```

This is because in the add method, we are trying to get the student id. In that case in the unit test we should no longer pass null as the enrollment object. In addition, we find that we are overriding the assertStudentRegistered method in this test. We should change it to use the new version:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
         void assertStudentRegistered(String studentId) {
            callLog.append("x");
         }
         void assertHasSeat(Enrollment enrollment) {
            callLog.append("y");
         }
      };
      enrollmentSet.setStorage(new EnrollmentStorage() {
         public void add(Enrollment enrollmentToStore) {
            callLog.append("z");
         }
      });
      Enrollment enrollment = new Enrollment(null, null, null);
      enrollmentSet.add(enrollment);
      assertEquals(callLog.toString(), "xyz");
   }
}
```

As there should be no more code using the original assertStudentRegistered, let's delete it. Observe that there is no compile error. Run the tests again. Yes, they continue to pass.

Now, check our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.
Get rid of assertHasSeat that takes an enrollment.

Let's do the last task. This is very similar to how we got rid of assertStudentRegistered. Therefore, we will only show the final changes:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class EnrollmentSet {
   void add(Enrollment enrollment) {
      assertStudentRegistered(enrollment.getStudentId());
      assertHasSeat(enrollment.getCourseCode());
      addToStorage(enrollment);
   }
   ...
}
class EnrollmentSetTest extends TestCase {
   ...
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
         void assertStudentRegistered(String studentId) {
            callLog.append("x");
         }
         void assertHasSeat(String courseCode) {
            callLog.append("y");
         }
      };
      enrollmentSet.setStorage(new EnrollmentStorage() {
         public void add(Enrollment enrollmentToStore) {
            callLog.append("z");
         }
      });
      Enrollment enrollment = new Enrollment(null, null, null);
      enrollmentSet.add(enrollment);
      assertEquals(callLog.toString(), "xyz");
   }
}
```

Now, check our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.

This is interesting. We are using the new Enrollment constructor. Even though it is empty (i.e., it does nothing at all), our tests still pass. It probably means our tests are missing something. Let's check the code of EnrollmentSet to see where it gets the student id or course code from an enrollment. It is in the add method:

```
class EnrollmentSet {
   ...
   void add(Enrollment enrollment) {
      assertStudentRegistered(enrollment.getStudentId());
      assertHasSeat(enrollment.getCourseCode());
      addToStorage(enrollment);
   }
}
```

It means these parts of the code are not being tested. At the moment we are testing the call sequence inside the add method, but are not checking the arguments passed to those methods:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
class EnrollmentSetTest extends TestCase {
   ...
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
         void assertStudentRegistered(String studentId) {
            callLog.append("x");
         }
         void assertHasSeat(String courseCode) {
            callLog.append("y");
         }
      };
      enrollmentSet.setStorage(new EnrollmentStorage() {
         public void add(Enrollment enrollmentToStore) {
            callLog.append("z");
         }
      });
      Enrollment enrollment = new Enrollment(null, null, null);
      enrollmentSet.add(enrollment);
      assertEquals(callLog.toString(), "xyz");
   }
}
```

Therefore, we should also check the arguments actually passed:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
         void assertStudentRegistered(String studentId) {
            assertTrue(studentId==null);
            callLog.append("x");
         }
         void assertHasSeat(String courseCode) {
            assertTrue(courseCode==null);
            callLog.append("y");
         }
      };
      enrollmentSet.setStorage(new EnrollmentStorage() {
         public void add(Enrollment enrollmentToStore) {
            callLog.append("z");
         }
      });
      Enrollment enrollment = new Enrollment(null, null, null);
      enrollmentSet.add(enrollment);
      assertEquals(callLog.toString(), "xyz");
   }
}
```

The arguments should be null because we are using null as the student id and course code. However, this is not a good test. A better test is have non-null student id and course code:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            void assertStudentRegistered(String studentId) {
               assertEquals(studentId, "a");
               callLog.append("x");
            }
            void assertHasSeat(String courseCode) {
               assertEquals(courseCode, "b");
               callLog.append("y");
            }
         };
         enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
               callLog.append("z");
            }
         });
         Enrollment enrollment = new Enrollment("a", "b", null);
         enrollmentSet.add(enrollment);
         assertEquals(callLog.toString(), "xyz");
      }
   }
```

Now run the test again and it should fail. This is exactly our purpose. We'd like to force ourselves to create some code in the Enrollment constructor. So, let's do it:

```
class Enrollment {
   Enrollment(String studentId, String courseCode, Payment payment) {
      this.studentId = studentId;
      this.courseCode = courseCode;
   }
   ...
}
```

Note that the payment parameter is still not used because our test doesn't require it. So, update our todo list:

TODO
Write test to force us to store the payment in the Enrollment constructor.

Now run the tests and now they should all pass. Finally we are ready to test the payment. Uncomment the testPaymentCorrect method:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testPaymentCorrect() {
      Payment payment = new CashPayment(100);
      Enrollment enrollment = new Enrollment("s001", "c001", payment);
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
         int getFee(String courseCode) {
            return courseCode.equals("c001") ? 100 : 0;
         }
      });
      enrollmentSet.assertPaymentCorrect(enrollment);
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Learning from the past mistakes we know that it doesn't really need an enrollment object. It only needs a course code (to find course fee) and the payment:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testPaymentCorrect() {
      Payment payment = new CashPayment(100);
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
         int getFee(String courseCode) {
            return courseCode.equals("c001") ? 100 : 0;
         }
      });
      enrollmentSet.assertPaymentCorrect("c001", payment);
   }
}
```

To make it compile, create a CashPayment class and an empty constructor, an empty setCourseFeeLookup method and an empty assertPaymentCorrect method:

```
class CashPayment extends Payment {
   CashPayment(int amount) {
   }
}
class EnrollmentSet {
   ...
   void setCourseFeeLookup(CourseFeeLookup lookup) {
   }
   void assertPaymentCorrect(String string, Payment payment) {
   }
}
```

Now run the test. It passes! It means this test is not enough. This is because our assertPaymentCorrect method does nothing and thus any payment is considered correct. So, write another test that should fail: validate an incorrect payment. We can do it by changing the course fee from 100 to say 101 and expect that an IncorrectPaymentException:

```
class EnrollmentSetTest extends TestCase {
   ...
   void testPaymentIncorrect() {
      Payment payment = new CashPayment(100);
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
         public int getFee(String courseCode) {
            return courseCode.equals("c001") ? 101 : 0;
         }
      });
      try {
         enrollmentSet.assertPaymentCorrect("c001", payment);
         fail();
      } catch (IncorrectPaymentException e) {
      }
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

To make it compile, create an IncorrectPaymentException class. Then run the test and it should fail. Now, write the implementation code so that the test passes:

```
class EnrollmentSet {
   CourseFeeLookup courseFeeLookup;
   ...
   void setCourseFeeLookup(CourseFeeLookup lookup) {
      this.courseFeeLookup = lookup;
   }
   void assertPaymentCorrect(String courseCode, Payment payment) {
      if (courseFeeLookup.getFee(courseCode) != payment.getAmount()) {
         throw new IncorrectPaymentException();
      }
   }
}
```

This will force us to create the getAmount method and quite some related code:

```
abstract class Payment {
   abstract int getAmount();
}
class CashPayment extends Payment {
   int amount;
   public CashPayment(int amount) {
      this.amount = amount;
   }
   int getAmount() {
      return amount;
   }
}
```

Now run the tests again and they should pass. By now it seems that we're done with the payment checking. But we are not.

Maintaining an integrated mental picture of EnrollmentSet

There is a bug in EnrollmentSet. If we run the acceptance tests for this story, we will find that it doesn't check the payment before saving the enrollment to the database. This bug may also be found by inspecting the code of EnrollmentSet from start to end:

```
class EnrollmentSet {
   EnrollmentStorage storage;
   StudentRegistryChecker studentRegistryChecker;
   EnrollmentCounter enrollmentCounter;
   ClassSizeGetter classSizeGetter;
   ReservationCounter reservationCounter;
   CourseFeeLookup courseFeeLookup;

   void setStorage(EnrollmentStorage storage) {
      this.storage = storage;
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    void add(Enrollment enrollment) {
       assertStudentRegistered(enrollment.getStudentId());
       assertHasSeat(enrollment.getCourseCode());
       //Not calling assertPaymentCorrect!
       addToStorage(enrollment);
    }
    void assertStudentRegistered(String studentId) {
       if (!studentRegistryChecker.isRegistered(studentId)) {
          throw new StudentNotFoundException();
       }
    }
    void assertHasSeat(String courseCode) {
       if (enrollmentCounter.getNoSeatsTaken(courseCode)
          + reservationCounter.getNoSeatsReserved(courseCode)
          >= classSizeGetter.getClassSize(courseCode)) {
          throw new ClassFullException();
       }
    }
    void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
       this.studentRegistryChecker = registryChecker;
    }
    void setClassSizeGetter(ClassSizeGetter sizeGetter) {
       this.classSizeGetter = sizeGetter;
    }
    void setEnrollmentCounter(EnrollmentCounter counter) {
       this.enrollmentCounter = counter;
    }
    void setReservationCounter(ReservationCounter counter) {
       this.reservationCounter = counter;
    }
    void addToStorage(Enrollment enrollment) {
       storage.add(enrollment);
    }
    void setCourseFeeLookup(CourseFeeLookup lookup) {
       this.courseFeeLookup = lookup;
    }
    void assertPaymentCorrect(String courseCode, Payment payment) {
       if (courseFeeLookup.getFee(courseCode) != payment.getAmount()) {
          throw new IncorrectPaymentException();
       }
    }
}
```

We did test the behavior of assertPaymentCorrect but we forgot to test that it is indeed called by the add method. In fact, in the whole development process, our idea on the actual code of the whole EnrollmentSet class is moot. Because each behavior is tested in isolation, it is easy to lose sight of the whole picture. Then integration bugs may occur. Therefore, we should regularly review the whole class to maintain an integrated mental picture.

Returning to the bug, as always, write a failing test to show the bug. We do this by amending the testValidateBeforeAddToStorage method:

```
class EnrollmentSetTest extends TestCase {
   void testValidateBeforeAddToStorage() {
      final StringBuffer callLog = new StringBuffer();
      final Payment payment = new CashPayment(5);
      EnrollmentSet enrollmentSet = new EnrollmentSet() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            void assertStudentRegistered(String studentId) {
               assertEquals(studentId, "a");
               callLog.append("x");
            }
            void assertHasSeat(String courseCode) {
               assertEquals(courseCode, "b");
               callLog.append("y");
            }
            void assertPaymentCorrect(
               String courseCode,
               Payment payment2) {
               assertEquals(courseCode, "b");
               assertTrue(payment2 == payment);
               callLog.append("t");
            }

         };
         enrollmentSet.setStorage(new EnrollmentStorage() {
            void add(Enrollment enrollmentToStore) {
               callLog.append("z");
            }
         });
         Enrollment enrollment = new Enrollment("a", "b", payment);
         enrollmentSet.add(enrollment);
         assertEquals(callLog.toString(), "xytz");
      }
   }
```

Run it and it fails. This is good. Write the required implementation code:

```
   class EnrollmentSet {
      void add(Enrollment enrollment) {
         assertStudentRegistered(enrollment.getStudentId());
         assertHasSeat(enrollment.getCourseCode());
         assertPaymentCorrect(enrollment.getCourseCode(), enrollment.getPayment());
         addToStorage(enrollment);
      }
   }
```

This will in turn force us to write the getPayment method in Enrollment and complete its constructor:

```
   class Enrollment {
      String studentId;
      String courseCode;
      Payment payment;

      Enrollment(String studentId, String courseCode, Payment payment) {
         this.studentId = studentId;
         this.courseCode = courseCode;
         this.payment = payment;
      }
      Payment getPayment() {
         return payment;
      }
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Now run the test again and it should pass.

TDD and its benefits

The way we developed the code as shown above is called "Test Driven Development (TDD)" because we always write a failing test before we code the real thing. TDD has the following benefits over writing tests last:

- In order to easily write a unit test, we make extensive use of interfaces (e.g., StudentRegistryChecker and etc.). This makes unit tests very easy to write and read because there is no unneeded data in the tests. If we were not using TDD and just coded the implementation directly, we would easily make use of existing classes (e.g., StudentSet) and would have to pack a lot of unneeded data in the tests (e.g., creating huge objects like Student or Course).

- Due to the extensive use of interfaces, our classes are separate from one another (e.g., EnrollmentSet doesn't know about StudentSet at all) and thus are much more reusable.

- When writing unit tests, it is easy to write a test for one behavior, make it pass, then write another test for another behavior. That is, the whole task is divided into many small pieces to be individually conquered. If we were not using TDD and just coded the implementation directly, we would easily try to implement all the behaviors at the same time. This would take a lot of time and the code would remain untested for quite a long time. In contrast, with TDD, we only implement the code for the behavior being tested. Therefore it only takes a very little time (minutes) to do and we can test the code right away.

Do's and Dont's

- Do not include unneeded data in a test (e.g., if you only need a student id, don't create an enrollment object; if you only need to check the existence of a student, use a StudentRegistryCheck interface instead of the StudentSet class). To do that, you will mostly use interfaces and anonymous classes.

- Do not bear the pain of writing a lot of code in a test to setup the context (e.g., setup a database connection). If it happens, use an interface (e.g., EnrollmentStorage).

- Do not test the same thing in two tests (e.g., when testing validation, do not test the writing to the storage).

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- Do not write any code without a failing test, unless that code is very simple.
- Do not (at least try to avoid) break more than one tests at a time (e.g., if you need to add a parameter to Enrollment's constructor, add a new constructor along with the existing one, migrate all clients to the new one, run all tests and finally delete the old one).
- Do not try to do many things at the same time or do something that takes hours to do. Try to break it down into smaller behaviors. Use a todo list.
- Do write a test and make it pass in a few minutes.
- Do run all the tests after making one or a few small changes.
- Do pick a task that can teach you something before those that are boring.
- Do use a call log to test the calling sequence.
- Do strive to maintain a whole picture at any time.

References

- Kent Beck, Test Driven Development: By Example, Addison-Wesley, 2002.
- David Astels, Test-Driven Development, A Practical Guide by, Prentice Hall, 2003.
- http://www.objectmentor.com/writeUps/TestDrivenDevelopment.
- http://www.objectmentor.com/resources/articles/xpepisode.htm.
- http://www.mockobjects.com.
- http://www.testdriven.com.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Introduction

- You must use TDD to develop the solutions.
- You must use a todo list.

Problems

1. Develop the code to count the number of files directly in a given directory whose sizes are at least 2GB. Sub-directories are ignored and not considered as files.

2. Develop the code to delete a directory and all the files inside. To delete a directory, you must empty it first.

3. Develop the code to generate a unique code for each fax. The code consists of three parts and is like 1/2004/HR. The first part is a sequence number that starts from 1. The next time it will be 2. The second part is the current year. The third part is the id of the department that issues the fax. Each department will have its own sequence number.

4. Develop the code to check if two employees are equal. An employee has an employee id, a list of qualifications, a superior (also an employee). A qualification has a text description and indicates the year when the qualification was achieved.

5. Develop a EnrollmentDBStorage class implementing the EnrollmentStorage interface. It should be able to write an Enrollment object into a database. You can use the schema shown in the text or make up your own. You can assume the name of the database, the driver used and etc.

6. Develop the code to find the number of seats taken for a given course (due to enrollments in it or in its parent). As part of it, you should have a class implementing the EnrollmentCounter shown in the text. When you need to access the database, use an interface instead.

7. Develop the code to find the number of seats reserved for a
reservations for it or its parent). As part of it, you should have a
ReservationCounter shown in the text. When you need to acces
interface instead. You should only count the reservations that are s

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Hints

1. Consider using the following interfaces to replace the file system:

```
public interface FileLookup {
   public String[] getFilesIn(String dirPath);
}
public interface DirChecker {
   public boolean isDir(String path);
}
public interface FileSizeGetter {
   public long getSize(String path);
}
```

2. It is similar to the previous one.

3. Consider using the following interfaces to replace the system clock:

```
public interface YearGetter {
   public int getCurrentYear();
}
```

   To avoid testing the same thing in different tests, you may imagine that the FaxCodeGenerator is like:

```
public class FaxCodeGenerator {
   public String getSequenceNo() {
      ...
   }
   public String generate() {
      return getSequenceNo() + ...
   }
}
```

   and then test the getSequenceNo method and the generate method separately. Alternatively, you may separate the behavior of getSequenceNo into a separate class:

```
public class SequenceNumberGenerator {
   public String getSequenceNo() {
      ...
   }
}
public class FaxCodeGenerator {
   public String generate() {
      ...
   }
}
```

   and then test each class separately.

4. In the tests you will need to create some employee objects, qualification list objects and etc. Try to use as little data as possible as it will turn out to be useless. Try using nulls as long as appropriate.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

5. As this code must access the database, you shouldn't use an interface to replace the database. So, you must create a database and a table in order to run these tests.

6. The following interfaces should make it easier to write the tests:

```
public interface CourseParentGetter {
   public String getParentCode(String courseCode);
}
public interface EnrollmentSource {
   public List getEnrollmentsFor(String courseCode);
}
```

   In addition, you can use the following interface to replace the database:

```
public interface CourseSource {
   public Course getCourse(String courseCode);
}
```

7. It is similar to the previous one. However, the code developed would be similar to the code in the previous problem. It'd be great if you'd remove the duplication (both the tests and the implementation code).

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Develop the code to count the number of files directly in a given directory whose sizes are at least 2GB. Sub-directories are ignored and not considered as files.

   The major problem here is that to setup the test context, we need to have a directory to put some files into. In particular, some of the files in it must be >= 2GB. We must also ensure the directory is initially empty, otherwise the test has to clean up its contents first. We must also ensure that the directory doesn't contain useful files. Therefore, setting up the context will take a lot of care, code, execution time (to create large files) and disk space. To solve this problem, we can use some interfaces to replace the system file:

```
public interface FileLookup {
   public String[] getFilesIn(String dirPath);
}
public interface DirChecker {
   public boolean isDir(String path);
}
public interface FileSizeGetter {
   public long getSize(String path);
}
```

   The tests are:

```
public class FileCounterTest extends TestCase {
   public void testCount() {
      final long KB = 1024;
      final long MB = 1024 * KB;
      final long GB = 1024 * MB;
      FileCounter fileCounter = new FileCounter();
      fileCounter.setFileSizeGetter(new FileSizeGetter() {
         public long getSize(String path) {
            if (path.equals("dir/a")) {
               return 2 * GB;
            }
            if (path.equals("dir/b")) {
               return 2 * GB - 1;
            }
            if (path.equals("dir/c")) {
               return 2 * GB + 1;
            }
            fail();
            return 0;
         }
      });
      fileCounter.setDirChecker(new DirChecker() {
         public boolean isDir(String path) {
            return false;
         }
      });
      fileCounter.setFileLookup(new FileLookup() {
         public String[] getFilesIn(String dirPath) {
            if (dirPath.equals("dir")) {
               return new String[] { "a", "b", "c" };
            }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            return null;
         }
      });
      assertEquals(fileCounter.countLargeFilesIn("dir"), 2);
   }
   public void testIgnoreDirectories() {
      FileCounter fileCounter = new FileCounter();
      fileCounter.setFileSizeGetter(new FileSizeGetter() {
         public long getSize(String path) {
            fail();
            return 0;
         }
      });
      fileCounter.setDirChecker(new DirChecker() {
         public boolean isDir(String path) {
            return true;
         }
      });
      fileCounter.setFileLookup(new FileLookup() {
         public String[] getFilesIn(String dirPath) {
            if (dirPath.equals("dir")) {
               return new String[] { "d1" };
            }
            return null;
         }
      });
      assertEquals(fileCounter.countLargeFilesIn("dir"), 0);
   }
}
```

The implementation code is:

```
public class FileCounter {
   private FileLookup fileLookup;
   private FileSizeGetter fileSizeGetter;
   private DirChecker dirChecker;

   public void setFileLookup(FileLookup lookup) {
      this.fileLookup = lookup;
   }
   public void setDirChecker(DirChecker checker) {
      this.dirChecker = checker;
   }
   public void setFileSizeGetter(FileSizeGetter getter) {
      this.fileSizeGetter = getter;
   }
   public int countLargeFilesIn(String pathToDir) {
      final long KB = 1024;
      final long MB = 1024 * KB;
      final long GB = 1024 * MB;
      String files[] = fileLookup.getFilesIn(pathToDir);
      int noLargeFiles = 0;
      for (int i = 0; i < files.length; i++) {
         if (!dirChecker.isDir(files[i])) {
            if (fileSizeGetter.getSize(pathToDir + "/" + files[i])
               >= 2 * GB) {
               noLargeFiles++;
            }
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      }
      return noLargeFiles;
   }
}
```

Ultimately we have to use the file system to implement these interfaces. This code is not tested by the unit tests so it'd better be as thin as possible:

```
public class FileLookupInJava implements FileLookup {
   public String[] getFilesIn(String dirPath) {
      return new File(dirPath).list();
   }
}
public class FileSizeGetterInJava implements FileSizeGetter {
   public long getSize(String path) {
      return new File(path).length();
   }
}
public class DirCheckerInJava implements DirChecker {
   public boolean isDir(String path) {
      return new File(path).isDirectory();
   }
}
```

The FileCounter should use these implementations by default:

```
public class FileCounter {
   public FileCounter() {
      setFileLookup(new FileLookupInJava());
      setFileSizeGetter(new FileSizeGetterInJava());
      setDirChecker(new DirCheckerInJava());
   }
   ...
}
```

2. Develop the code to delete a directory and all the files inside. To delete a directory, you must empty it first.

   Like the previous one, we'd use some interfaces to replace the system file:

```
public interface FileLookup {
   public String[] getFilesIn(String path);
}
public interface FileRemover {
   public void delete(String path);
}
```

   The tests are:

```
public class RecursiveFileRemoverTest extends TestCase {
   public void testEmpty() {
      final StringBuffer deleteLog = new StringBuffer();
      RecursiveFileRemover remover = new RecursiveFileRemover();
      remover.setFileLookup(new FileLookup() {
         public String[] getFilesIn(String dirPath) {
            return null;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            }
         });
         remover.setFileRemover(new FileRemover() {
            public void delete(String path) {
               deleteLog.append("<" + path + ">");
            }
         });
         remover.delete("dir");
         assertEquals(deleteLog.toString(), "<dir>");
      }
      public void testDeleteFilesFirst() {
         final StringBuffer deleteLog = new StringBuffer();
         RecursiveFileRemover remover = new RecursiveFileRemover();
         remover.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
               if (dirPath.equals("dir")) {
                  return new String[] { "a", "b" };
               }
               return null;
            }
         });
         remover.setFileRemover(new FileRemover() {
            public void delete(String path) {
               deleteLog.append("<" + path + ">");
            }
         });
         remover.delete("dir");
         assertEquals(deleteLog.toString(), "<dir/a><dir/b><dir>");
      }
      public void testRecursion() {
         final StringBuffer deleteLog = new StringBuffer();
         RecursiveFileRemover remover = new RecursiveFileRemover();
         remover.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
               if (dirPath.equals("d1")) {
                  return new String[] { "d2" };
               }
               if (dirPath.equals("d1/d2")) {
                  return new String[] { "f1" };
               }
               return null;
            }
         });
         remover.setFileRemover(new FileRemover() {
            public void delete(String path) {
               deleteLog.append("<" + path + ">");
            }
         });
         remover.delete("d1");
         assertEquals(deleteLog.toString(), "<d1/d2/f1><d1/d2><d1>");
      }
}
```

The implementation code is:

```
public class RecursiveFileRemover {
   private FileLookup fileLookup;
   private FileRemover fileRemover;
   public void setFileLookup(FileLookup lookup) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      this.fileLookup = lookup;
   }
   public void setFileRemover(FileRemover remover) {
      this.fileRemover = remover;
   }
   public void delete(String path) {
      String filesInDir[] = fileLookup.getFilesIn(path);
      if (filesInDir != null) {
         for (int i = 0; i < filesInDir.length; i++) {
            String pathToChild = path + "/" + filesInDir[i];
            delete(pathToChild);
         }
      }
      fileRemover.delete(path);
   }
}
```

The RecursiveFileRemover should by default use implementations of FileLookup and FileRemover that rely on the file system:

```
public class FileLookupInJava implements FileLookup {
   public String[] getFilesIn(String path) {
      return new File(path).list();
   }
}
public class FileRemoverInJava implements FileRemover {
   public void delete(String path) {
      new File(path).delete();
   }
}
public class RecursiveFileRemover {
   public RecursiveFileRemover() {
      setFileLookup(new FileLookupInJava());
      setFileRemover(new FileRemoverInJava());
   }
   ...
}
```

3. Develop the code to generate a unique code for each fax. The code consists of three parts and is like 1/2004/HR. The first part is a sequence number that starts from 1. The next time it will be 2. The second part is the current year. The third part is the id of the department that issues the fax. Each department will have its own sequence number.

   The most obvious problem here is that the code needs to find out the current year. To setup the context so that it generates the code 1/2004/HR, we seem to have to set the system clock first to year 2004, otherwise the test will break when it is run in say 2005. Setting the system clock is a nasty thing to do. A much better way is to use an interface to replace the system clock:

```
public interface YearGetter {
   public int getCurrentYear();
}
```

Then we can write the tests more easily. However, when writing the tests, we will find

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

that we are testing two separate behaviors: one is that the fax code consists of the sequence number, the current year and the department id; the other one is that the sequence number is incremented. So it is easier to test and implement them separately. Here are the tests and implementation code for the fax code behavior:

```
public interface NumberGenerator {
   public void setSeqNo(int seqNo);
   public int generate();
}
public class FaxCodeGeneratorTest extends TestCase {
   public void testCombineFormat() {
      FaxCodeGenerator generator = new FaxCodeGenerator("a");
      generator.setNumberGenerator(new NumberGenerator() {
         public int generate() {
            return 3;
         }
      });
      generator.setYearGetter(new YearGetter() {
         public int getCurrentYear() {
            return 2004;
         }
      });
      assertEquals(generator.generate(), "3/2004/a");
   }
   public void testInitialSeqNo() {
      final StringBuffer callLog = new StringBuffer();
      FaxCodeGenerator generator = new FaxCodeGenerator("a");
      generator.setNumberGenerator(new NumberGenerator() {
         public void setSeqNo(int seqNo) {
            assertEquals(seqNo, 1);
            callLog.append("x");
         }
         public int generate() {
            fail();
            return 0;
         }
      });
      assertEquals(callLog.toString(), "x");
   }
}
public class FaxCodeGenerator {
   private final static int INITIAL_SEQ_NO=1;
   private NumberGenerator numberGenerator;
   private String departId;
   private YearGetter yearGetter;

   public FaxCodeGenerator(String departId) {
      this.departId = departId;
   }
   public void setYearGetter(YearGetter getter) {
      this.yearGetter = getter;
   }
   public String generate() {
      return numberGenerator.generate()
         + "/"
         + yearGetter.getCurrentYear()
         + "/"
         + departId;
   }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public void setNumberGenerator(NumberGenerator generator) {
      this.numberGenerator = generator;
      this.numberGenerator.setSeqNo(INITIAL_SEQ_NO);
   }
}
```

Here are the tests and implementation code for the sequence number behavior:

```
public class SeqNoGeneratorTest extends TestCase {
   public void testInitialValue() {
      SeqNoGenerator generator = new SeqNoGenerator(10);
      assertEquals(generator.generate(), 10);
   }
   public void testIncrementAfterGenerate() {
      SeqNoGenerator generator = new SeqNoGenerator(10);
      generator.generate();
      assertEquals(generator.generate(), 11);
   }
}
public class SeqNoGenerator implements NumberGenerator {
   private int seqNo;
   public SeqNoGenerator(int initialSeqNo) {
      this.seqNo = initialSeqNo;
   }
   public int generate() {
      return seqNo++;
   }
   public void setSeqNo(int seqNo) {
      this.seqNo = seqNo;
   }
}
```

Ultimately we have to use the system clock to implement the YearGetter interface. To make this code as thin as possible, we further divide this into two behaviors: to get the current time from the system clock and to get the year from a given time. The former cannot be tested but the latter can. The code for the latter is:

```
public class YearGetterFromCalendarTest extends TestCase {
   public void testGetCurrentYear() {
      GregorianCalendar calendar = new GregorianCalendar(2003, 0, 23);
      YearGetterFromCalendar getter = new YearGetterFromCalendar(calendar);
      assertEquals(getter.getCurrentYear(), 2003);
   }
}
public class YearGetterFromCalendar implements YearGetter {
   private GregorianCalendar calendar;
   public YearGetterFromCalendar(GregorianCalendar calendar) {
      this.calendar = calendar;
   }
   public int getCurrentYear() {
      return calendar.get(Calendar.YEAR);
   }
}
```

The FaxCodeGenerator should by default use this implementation:

```
public class FaxCodeGenerator {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   public FaxCodeGenerator(String departId) {
      this.departId = departId;
      setYearGetter(new YearGetterFromCalendar(new GregorianCalendar()));
   }
   ...
}
```

It should also use the SeqNoGenerator as the NumberGenerator by default:

```
public class FaxCodeGenerator {
   public FaxCodeGenerator(String departId) {
      this.departId = departId;
      setYearGetter(new YearGetterFromCalendar(new GregorianCalendar()));
      setNumberGenerator(new SeqNoGenerator(INITIAL_SEQ_NO));
   }
   ...
}
```

4. Develop the code to check if two employees are equal. An employee has an employee id, a list of qualifications, a superior (also an employee). A qualification has a text description and indicates the year when the qualification was achieved.

   The difficulty here is that we need to setup a lot of data (two employee objects) but most turn out to be unused. The real behavior in the Employee class here is that when determining whether it is equal to another Employee object, it checks if its three elements (id, qualification list, superior) are equal to their counterparts in that other Employee object. The actual contents of its id, qualification list and superior are totally unimportant to it.

   The tests are:

```
public interface EqualityChecker {
   public boolean eachEquals(Object objList1[], Object objList2[]);
}
public class EmployeeTest extends TestCase {
   public void testEquals() {
      final StringBuffer callLog = new StringBuffer();
      final QualificationList qualiList1 = new QualificationList();
      final QualificationList qualiList2 = new QualificationList();
      final Employee superior1 = new Employee(null, null, null);
      final Employee superior2 = new Employee(null, null, null);
      Employee employee1 = new Employee("id1", qualiList1, superior1);
      Employee employee2 = new Employee("id2", qualiList2, superior2);
      employee1.setEqualityChecker(new EqualityChecker() {
         public boolean eachEquals(Object[] objList1, Object[] objList2) {
            callLog.append("x");
            assertEquals(objList1.length, 3);
            assertEquals(objList1[0], "id1");
            assertSame(objList1[1], qualiList1);
            //assertSame(X, Y) means assertTrue(X==Y). It is provided by JUnit.
            assertSame(objList1[2], superior1);
            assertEquals(objList2.length, 3);
            assertEquals(objList2[0], "id2");
            assertSame(objList2[1], qualiList2);
            assertSame(objList2[2], superior2);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            return true;
         }
      });
      assertTrue(employee1.equals(employee2));
      assertEquals(callLog.toString(), "x");
   }
   public void testNonEmployee() {
      Employee employee = new Employee("id", null, null);
      employee.setEqualityChecker(new EqualityChecker() {
         public boolean eachEquals(Object[] objList1, Object[] objList2) {
            fail();
            return false;
         }
      });
      assertFalse(employee.equals("non-employee"));
      //assertFalse(X) means assertTrue(!X). It is provided by JUnit.
   }
}
```

The implementation code is:

```
public class Employee {
   private String id;
   private QualificationList qualiList;
   private Employee superior;
   private EqualityChecker equalityChecker;

   public Employee(
      String id,
      QualificationList qualiList,
      Employee superior) {
      this.id = id;
      this.qualiList = qualiList;
      this.superior = superior;
   }
   public boolean equals(Object obj) {
      return obj instanceof Employee ? equals((Employee) obj) : false;
   }
   private boolean equals(Employee employee) {
      return equalityChecker.eachEquals(
         getElementsForEqualityCheck(),
         employee.getElementsForEqualityCheck());
   }
   private Object[] getElementsForEqualityCheck() {
      return new Object[] { id, qualiList, superior };
   }
   public void setEqualityChecker(EqualityChecker checker) {
      this.equalityChecker = checker;
   }
}
```

We will need to provide an implementation for EqualityChecker:

```
public class ShortCircuitEqualityCheckerTest extends TestCase {
   public void testOneElement() {
      ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
      assertTrue(
         checker.eachEquals(new Object[] { "a" }, new Object[] { "a" }));
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      assertFalse(
         checker.eachEquals(new Object[] { "a" }, new Object[] { "b" }));
   }
   public void testNotSameLength() {
      ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
      assertFalse(checker.eachEquals(new Object[] { "a" }, new Object[0]));
   }
   public void testFirstElementNotEqual() {
      ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
      assertFalse(
         checker.eachEquals(
            new Object[] { "a", "b" },
            new Object[] { "c", "b" }));
   }
   public void testFirstElementEqual() {
      ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
      assertTrue(
         checker.eachEquals(
            new Object[] { "a", "b" },
            new Object[] { "a", "b" }));
      assertFalse(
         checker.eachEquals(
            new Object[] { "a", "b" },
            new Object[] { "a", "c" }));
   }
}
```

The Employee class should by default use this implementation:

```
public class Employee {
   public Employee(
      String id,
      QualificationList qualiList,
      Employee superior) {
      ...
      equalityChecker = new ShortCircuitEqualityChecker();
   }
   ...
}
```

After implementing Employee, we also need to implement the equality check for the QualificationList class. It is very similar to Employee:

```
public class QualificationListTest extends TestCase {
   public void testEquals() {
      final StringBuffer callLog = new StringBuffer();
      QualificationList list1 = new QualificationList() {
         public List getQualifications() {
            List qualifications = new ArrayList();
            qualifications.add("a");
            qualifications.add("b");
            return qualifications;
         }
      };
      QualificationList list2 = new QualificationList() {
         public List getQualifications() {
            List qualifications = new ArrayList();
            qualifications.add("c");
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            qualifications.add("d");
            return qualifications;
         }
      };
      list1.setEqualityChecker(new EqualityChecker() {
         public boolean eachEquals(Object[] objList1, Object[] objList2) {
            callLog.append("x");
            assertEquals(objList1.length, 2);
            assertEquals(objList1[0], "a");
            assertEquals(objList1[1], "b");
            assertEquals(objList2.length, 2);
            assertEquals(objList2[0], "c");
            assertEquals(objList2[1], "d");
            return true;
         }
      });
      assertTrue(list1.equals(list2));
      assertEquals(callLog.toString(), "x");
   }
   public void testNonQualificationList() {
      QualificationList list = new QualificationList();
      list.setEqualityChecker(new EqualityChecker() {
         public boolean eachEquals(Object[] objList1, Object[] objList2) {
            fail();
            return true;
         }
      });
      assertFalse(list.equals("non-qualification-list"));
   }
}
public class QualificationList {
   private EqualityChecker equalityChecker;

   public QualificationList() {
      setEqualityChecker(new ShortCircuitEqualityChecker());
   }
   public void setEqualityChecker(EqualityChecker checker) {
      this.equalityChecker = checker;
   }
   public boolean equals(Object obj) {
      return obj instanceof QualificationList
         && equals((QualificationList) obj);
   }
   public List getQualifications() {
      return null;  //TODO: Implement when required.
   }
   private boolean equals(QualificationList list) {
      return equalityChecker.eachEquals(
         getQualifications().toArray(),
         list.getQualifications().toArray());
   }
}
```

Next, we also need to implement the equality check for the Qualification class in a similar way:

```
public class QualificationTest extends TestCase {
   public void testEquals() {
      Qualification qualification1 = new Qualification("desc1", 2003);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      Qualification qualification2 = new Qualification("desc2", 2004);
      qualification1.setEqualityChecker(new EqualityChecker() {
         public boolean eachEquals(Object[] objList1, Object[] objList2) {
            assertEquals(objList1.length, 2);
            assertEquals(objList1[0], "desc1");
            assertEquals(objList1[1], new Integer(2003));
            assertEquals(objList2.length, 2);
            assertEquals(objList2[0], "desc2");
            assertEquals(objList2[1], new Integer(2004));
            return true;
         }
      });
      assertTrue(qualification1.equals(qualification2));
   }
   public void testNotQualification() {
      Qualification qualification = new Qualification("desc", 2003);
      qualification.setEqualityChecker(new EqualityChecker() {
         public boolean eachEquals(Object[] objList1, Object[] objList2) {
            fail();
            return false;
         }
      });
      assertFalse(qualification.equals("non-qualification"));
   }
}
public class Qualification {
   private String desc;
   private int yearWhenAchieved;
   private EqualityChecker equalityChecker;

   public Qualification(String desc, int yearWhenAchieved) {
      this.desc = desc;
      this.yearWhenAchieved = yearWhenAchieved;
      setEqualityChecker(new ShortCircuitEqualityChecker());
   }
   public boolean equals(Object obj) {
      return (obj instanceof Qualification) && equals((Qualification) obj);
   }
   public boolean equals(Qualification qualification) {
      return equalityChecker.eachEquals(
         makeElementsForEqualityCheck(),
         qualification.makeElementsForEqualityCheck());
   }
   private Object[] makeElementsForEqualityCheck() {
      return new Object[] { desc, new Integer(yearWhenAchieved)};
   }
   public void setEqualityChecker(EqualityChecker checker) {
      this.equalityChecker = checker;
   }
}
```

5. Develop a EnrollmentDBStorage class implementing the EnrollmentStorage interface. It should be able to write an Enrollment object into a database. You only need to handle cash payments only. You can use the schema shown in the text or make up your own. You can assume the name of the database, the driver used and etc.

```
public class EnrollmentDBStorageTest extends TestCase {
   private Connection conn;
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
   protected void setUp() throws Exception {
      Class.forName("org.postgresql.Driver");
      conn =
         DriverManager.getConnection(
            "jdbc:postgresql://localhost/testdb",
            "testuser",
            "testpassword");
      deleteAllEnrollments();
   }
   protected void tearDown() throws Exception {
      conn.close();
   }
   private void deleteAllEnrollments() throws SQLException {
      PreparedStatement st =
         conn.prepareStatement("delete from enrollments");
      try {
         st.executeUpdate();
      } finally {
         st.close();
      }
   }
   public void testAdd() throws SQLException {
      EnrollmentDBStorage storage = new EnrollmentDBStorage(conn);
      Date enrolDate = new GregorianCalendar(2004, 0, 28).getTime();
      Payment payment = new CashPayment(200);
      Enrollment enrollment =
         new Enrollment("s001", "c001", enrolDate, payment);
      storage.add(enrollment);
      PreparedStatement st =
         conn.prepareStatement(
            "select * from enrollments where studentId=? and courseCode=?");
      try {
         st.setString(1, "s001");
         st.setString(2, "c001");
         ResultSet rs = st.executeQuery();
         assertTrue(rs.next());
         assertEquals(rs.getDate("enrolDate"), enrolDate);
         assertEquals(rs.getInt("amount"), 200);
         assertEquals(rs.getString("paymentType"), "Cash");
         assertNull(rs.getObject("cardNo"));
         //assertNull(X) means assertTrue(X == null). It is provided by JUnit.
         assertNull(rs.getObject("expiryDate"));
         assertNull(rs.getObject("nameOnCard"));
         assertFalse(rs.next());
      } finally {
         st.close();
      }
   }
}
public class EnrollmentDBStorage implements EnrollmentStorage {
   private Connection conn;

   public EnrollmentDBStorage(Connection conn) {
      this.conn = conn;
   }
   public void add(Enrollment enrollment) {
      try {
         PreparedStatement st =
            conn.prepareStatement(
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
                "insert into enrollments values(?,?,?,?,?,?,?,?)");
         try {
            st.setString(1, enrollment.getCourseCode());
            st.setString(2, enrollment.getStudentId());
            st.setDate(
               3,
               new java.sql.Date(enrollment.getEnrolDate().getTime()));
            CashPayment payment = (CashPayment) enrollment.getPayment();
            st.setInt(4, payment.getAmount());
            st.setString(5, "Cash");
            st.setNull(6, Types.VARCHAR);
            st.setNull(7, Types.DATE);
            st.setNull(8, Types.VARCHAR);
            st.executeUpdate();
         } finally {
            st.close();
         }
      } catch (SQLException e) {
         throw new RuntimeException(e);
      }
   }
}
```

6. Develop the code to find the number of seats taken for a given course (due to enrollments in it or in its parent). As part of it, you should have a class implementing the EnrollmentCounter shown in the text. When you need to access the database, use an interface instead.

   The tests are:

```
public interface CourseParentGetter {
   public String getParentCode(String courseCode);
}
public interface EnrollmentSource {
   public List getEnrollmentsFor(String courseCode);
}
public class EnrollmentCounterFromSrcTest extends TestCase {
   public void testConsiderAncestor() {
      EnrollmentCounterFromSrc counter = new EnrollmentCounterFromSrc() {
         public int getNoEnollmentsForSingleCourse(String courseCode) {
            if (courseCode.equals("c001")) {
               return 4;
            }
            if (courseCode.equals("c002")) {
               return 2;
            }
            if (courseCode.equals("c003")) {
               return 1;
            }
            fail();
            return 0;
         }
      };
      counter.setCourseParentGetter(new CourseParentGetter() {
         public String getParentCode(String courseCode) {
            if (courseCode.equals("c003")) {
               return "c002";
            }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            if (courseCode.equals("c002")) {
               return "c001";
            }
            if (courseCode.equals("c001")) {
               return null;
            }
            fail();
            return null;
         }
      });
      assertEquals(counter.getNoSeatsTaken("c003"), 7);
   }
   public void testSingleCourse() {
      EnrollmentCounterFromSrc counter = new EnrollmentCounterFromSrc();
      counter.setEnrollmentSource(new EnrollmentSource() {
         public List getEnrollmentsFor(String courseCode) {
            List enrollments = new ArrayList();
            enrollments.add("e1");
            enrollments.add("e2");
            return enrollments;
         }
      });
      assertEquals(counter.getNoEnollmentsForSingleCourse("c001"), 2);
   }
}
```

The implementation code is:

```
public class EnrollmentCounterFromSrc implements EnrollmentCounter {
   private EnrollmentSource enrollmentSource;
   private CourseParentGetter courseParentGetter;

   public int getNoSeatsTaken(String courseCode) {
      int noSeatsTaken = 0;
      for (;;) {
         noSeatsTaken += getNoEnollmentsForSingleCourse(courseCode);
         courseCode = courseParentGetter.getParentCode(courseCode);
         if (courseCode == null) {
            return noSeatsTaken;
         }
      }
   }
   public int getNoEnollmentsForSingleCourse(String courseCode) {
      return enrollmentSource.getEnrollmentsFor(courseCode).size();
   }
   public void setCourseParentGetter(CourseParentGetter getter) {
      this.courseParentGetter = getter;
   }
   public void setEnrollmentSource(EnrollmentSource source) {
      this.enrollmentSource = source;
   }
}
```

We also need to provide an implementation for the CourseParentGetter that gets its information from a CourseSource:

```
public interface CourseSource {
   public Course getCourse(String courseCode);
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
}
public class CourseParentGetterFromSrcTest extends TestCase {
   public void testHasParent() {
      CourseParentGetterFromSrc getter = new CourseParentGetterFromSrc();
      getter.setCourseSource(new CourseSource() {
         public Course getCourse(String courseCode) {
            Course c002 = new Course() {
               public String getParentCode() {
                  return "c001";
               }
            };
            return courseCode.equals("c002") ? c002 : null;
         }
      });
      assertEquals(getter.getParentCode("c002"), "c001");
   }
   public void testHasNoParent() {
      CourseParentGetterFromSrc getter = new CourseParentGetterFromSrc();
      getter.setCourseSource(new CourseSource() {
         public Course getCourse(String courseCode) {
            Course c002 = new Course() {
               public String getParentCode() {
                  return null;
               }
            };
            return courseCode.equals("c002") ? c002 : null;
         }
      });
      assertNull(getter.getParentCode("c002"));
   }
}
```

The implementation code is:

```
public class CourseParentGetterFromSrc implements CourseParentGetter {
   private CourseSource courseSource;

   public String getParentCode(String courseCode) {
      return courseSource.getCourse(courseCode).getParentCode();
   }
   public void setCourseSource(CourseSource source) {
      this.courseSource = source;
   }
}
```

The CourseParentGetterFromSrc should by default use an implementation for CourseSource that gets the information from the database:

```
public class CourseParentGetterFromSrc implements CourseParentGetter {
   public CourseParentGetterFromSrc() {
      setCourseSource(new CourseDBSource());
   }
   ...
}
```

The EnrollmentCounterFromSrc should by default use the implementations for the EnrollmentSource and CourseParentGetter interfaces that get the information from the

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

database:

```
public class EnrollmentCounterFromSrc implements EnrollmentCounter {
   public EnrollmentCounterFromSrc() {
      setEnrollmentSource(new EnrollmentDBSource());
      setCourseParentGetter(new CourseParentGetterFromSrc());
   }
   ...
}
```

7. Develop the code to find the number of seats reserved for a given course (due to reservations for it or its parent). As part of it, you should have a class implementing the ReservationCounter shown in the text. When you need to access the database, use an interface instead. You should only count the reservations that are still active.

   When we are working on this, we will find that the tests are very similar to those for the previous problem. It suggests that they share some common logic: They are both trying to calculate the sum of an integer for a course, for its parent, for its grand-parent and etc. So, we extract that common logic and test it in isolation:

```
public interface CourseIntGetter {
   public int getInt(String courseCode);
}
public interface CourseParentGetter {
   public String getParentCode(String courseCode);
}
public class CourseAncestorsTraverserTest extends TestCase {
   public void testSum() {
      CourseAncestorsTraverser traverser = new CourseAncestorsTraverser();
      traverser.setCourseParentGetter(new CourseParentGetter() {
         public String getParentCode(String courseCode) {
            if (courseCode.equals("c003")) {
               return "c002";
            }
            if (courseCode.equals("c002")) {
               return "c001";
            }
            if (courseCode.equals("c001")) {
               return null;
            }
            fail();
            return null;
         }
      });
      CourseIntGetter intGetter = new CourseIntGetter() {
         public int getInt(String courseCode) {
            if (courseCode.equals("c001")) {
               return 1;
            }
            if (courseCode.equals("c002")) {
               return 2;
            }
            if (courseCode.equals("c003")) {
               return 5;
            }
            fail();
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            return 0;
         }
      };
      assertEquals(traverser.sum("c003", intGetter), 8);
   }
}
```

The implementation code is:

```
public class CourseAncestorsTraverser {
   private CourseParentGetter courseParentGetter;

   public void setCourseParentGetter(CourseParentGetter getter) {
      this.courseParentGetter = getter;
   }
   public int sum(String courseCode, CourseIntGetter intGetter) {
      int sum = 0;
      for (;;) {
         sum += intGetter.getInt(courseCode);
         courseCode = courseParentGetter.getParentCode(courseCode);
         if (courseCode == null) {
            return sum;
         }
      }
   }
}
```

The CourseAncestorsTraverser should by default use an implementation for CourseParentGetter that gets the information from the database:

```
public class CourseParentGetterFromSrc implements CourseParentGetter {
   ...
}
public class CourseAncestorsTraverser {
   public CourseAncestorsTraverser() {
      setCourseParentGetter(new CourseParentGetterFromSrc());
   }
   ...
}
```

To get the number of enrollments and number of reservations, we just need to provide a different CourseIntGetter implementation for each. For enrollments:

```
public class CourseNoEnrollmentsGetterTest extends TestCase {
   public void testCount() {
      CourseNoEnrollmentsGetter getter = new CourseNoEnrollmentsGetter();
      getter.setEnrollmentSource(new EnrollmentSource() {
         public List getEnrollmentsFor(String courseCode) {
            if (courseCode.equals("c001")) {
               List enrollments = new ArrayList();
               enrollments.add("e1");
               enrollments.add("e2");
               return enrollments;
            }
            fail();
            return null;
         }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      });
      assertEquals(getter.getInt("c001"), 2);
   }
}
public class CourseNoEnrollmentsGetter implements CourseIntGetter {
   private EnrollmentSource enrollmentSource;

   public void setEnrollmentSource(EnrollmentSource source) {
      this.enrollmentSource = source;
   }
   public int getInt(String courseCode) {
      return enrollmentSource.getEnrollmentsFor(courseCode).size();
   }
}
```

The CourseNoEnrollmentsGetter should by default use an implementation for EnrollmentSource that gets the information from the database:

```
public class CourseNoEnrollmentsGetter implements CourseIntGetter {
   public CourseNoEnrollmentsGetter() {
      setEnrollmentSource(new EnrollmentDBSource());
   }
   ...
}
```

Finally the CourseAncestorsTraverser can act as an EnrollmentCounter:

```
public class CourseAncestorsTraverser implements EnrollmentCounter {
   ...
   public int getNoSeatsTaken(String courseCode) {
      return sum(courseCode, new CourseNoEnrollmentsGetter());
   }
}
```

For reservations, it is similar:

```
public class CourseNoReservationsGetterTest extends TestCase {
   public void testCountOnlyActive() {
      CourseNoReservationsGetter getter = new CourseNoReservationsGetter();
      getter.setReservationSource(new ReservationSource() {
         public List getReservationsFor(String courseCode) {
            if (courseCode.equals("c001")) {
               List reservations = new ArrayList();
               reservations.add(new Reservation() {
                  public boolean isActive() {
                     return true;
                  }
               });
               reservations.add(new Reservation() {
                  public boolean isActive() {
                     return false;
                  }
               });
               reservations.add(new Reservation() {
                  public boolean isActive() {
                     return true;
                  }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
                });
                return reservations;
            }
            fail();
            return null;
         }
      });
      assertEquals(getter.getInt("c001"), 2);
   }
}
public class CourseNoReservationsGetter implements CourseIntGetter {
   private ReservationSource reservationSource;

   public CourseNoReservationsGetter() {
      setReservationSource(new ReservationDBSource());
   }
   public int getInt(String courseCode) {
      int result = 0;
      List reservations = reservationSource.getReservationsFor(courseCode);
      for (Iterator iter = reservations.iterator(); iter.hasNext();) {
         Reservation reservation = (Reservation) iter.next();
         if (reservation.isActive()) {
            result++;
         }
      }
      return result;
   }
   public void setReservationSource(ReservationSource source) {
      this.reservationSource = source;
   }
}
public class CourseAncestorsTraverser
   implements EnrollmentCounter, ReservationCounter {
   ...
   public int getNoSeatsReserved(String courseCode) {
      return sum(courseCode, new CourseNoReservationsGetter());
   }
}
```

The Reservation class needs to check if it's active or not. To test it, it is easier to use an interface to replace the system clock:

```
public interface Clock {
   public Date getCurrentDate();
}
public class ReservationTest extends TestCase {
   public void testActive() {
      Reservation reservation = new Reservation();
      reservation.setClock(new Clock() {
         public Date getCurrentDate() {
            return new GregorianCalendar(2004, 0, 22).getTime();
         }
      });
      reservation.setReserveDate(
         new GregorianCalendar(2004, 0, 20).getTime());
      reservation.setDaysReserved(3);
      assertTrue(reservation.isActive());
   }
   public void testInactive() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      Reservation reservation = new Reservation();
      reservation.setClock(new Clock() {
         public Date getCurrentDate() {
            return new GregorianCalendar(2004, 0, 22).getTime();
         }
      });
      reservation.setReserveDate(
         new GregorianCalendar(2004, 0, 20).getTime());
      reservation.setDaysReserved(2);
      assertFalse(reservation.isActive());
   }
}
public class Reservation {
   private Clock clock;
   private Date reserveDate;
   private int daysReserved;

   public boolean isActive() {
      GregorianCalendar lastEffectiveDate = new GregorianCalendar();
      lastEffectiveDate.setTime(reserveDate);
      lastEffectiveDate.add(Calendar.DAY_OF_MONTH, daysReserved - 1);
      return !clock.getCurrentDate().after(lastEffectiveDate.getTime());
   }
   public void setDaysReserved(int daysReserved) {
      this.daysReserved = daysReserved;
   }
   public void setReserveDate(Date reserveDate) {
      this.reserveDate = reserveDate;
   }
   public void setClock(Clock clock) {
      this.clock = clock;
   }
}
```

The Reservation class should by default use an implementation for Clock that gets the data from the system clock:

```
public class ClockInJava implements Clock {
   public Date getCurrentDate() {
      return new Date();
   }
}
public class Reservation {
   public Reservation() {
      setClock(new ClockInJava());
   }
   ...
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 14

Team Development with CVS

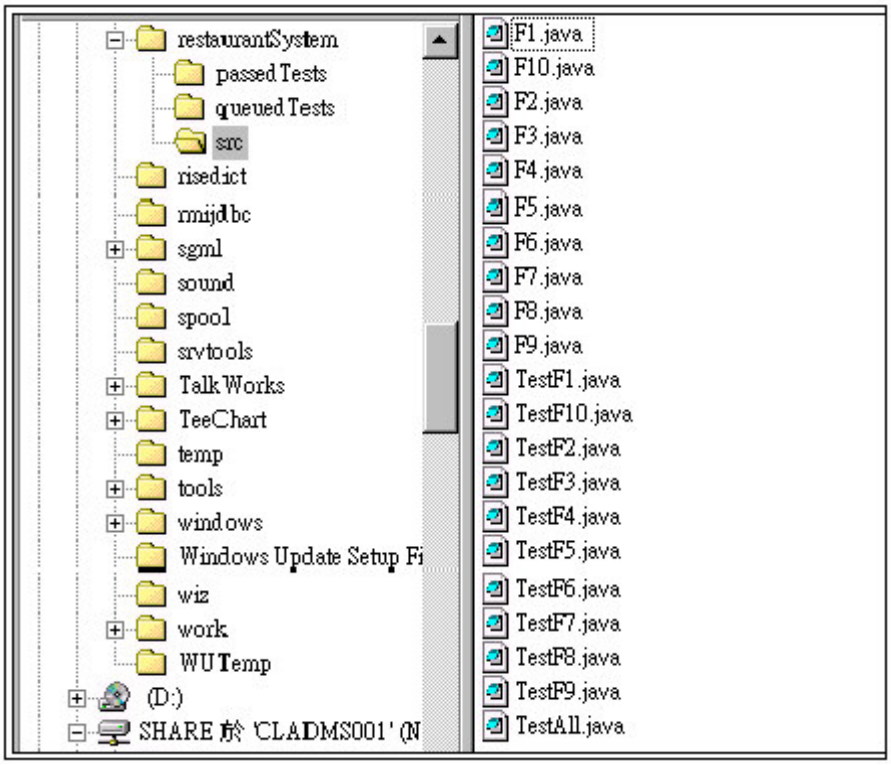
Introduction to a system

Suppose that you are developing a restaurant management system. This system consists of ten source files: F1.java, F2.java, F3.java, ..., F10.java. In addition, you have written the corresponding unit tests like TestF1.java, TestF2.java, ...TestF10.java, as well as a TestAll.java to run all the unit tests in the system. Suppose that all these .java files have been placed into the c:\restaurantSystem\src folder.

Suppose that according to the customer's requirements you have also written twenty acceptance test files such as testAddOrder.test, testAddCustomer.test and etc. At the beginning you place them into c:\restaurantSystem\queuedTests. As development makes progresses, some acceptance tests already pass. So, you move them from c:\restaurantSystem\queuedTests into c:\restaurantSystem\passedTests.

Therefore, your folders look like this:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



Indeed, this way to organize files is pretty good.

Story sign up

Suppose that in order to finish the system sooner, you have hired Paul and John to develop the system with you. In the next iteration, the customer requests that user stories 3, 5, 10, 7 and 12 be implemented. You are very open-minded and hope that they can choose how many and which user stories to do. Finally, Paul chooses to do user stories 5 and 7, because user stories 5 and 7 involves a database, which is exactly Paul's strength; John chooses user story 3, because he is just a fresh university graduate and this is the simplest user story (Of course, John knows very well that in the future he has to choose more user stories, otherwise he will have no hope of any pay raise!); You choose user stories 10 and 12, two of the most difficult user stories in this iteration, because you are the best developer on the team; You must not show any hesitation. This process of the developers signing up for user stories for themselves is called "story sign up".

What are the good things about it? Look, if you insist to assign user stories 10 and 12 to John,

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

assign user story 3 to Paul and do user stories 5 and 7 yourself, will they be happy? When an unhappy developer is assigned to do something unsuitable for him, will he work efficiently?

Difficulty in team development

Now the iteration is started. You start to implement user stories 10; Paul starts to implement user story 5; John starts to implement user story 3. You have shared the restaurantSystem folder so that they can access it. Suppose that you need to modify F1.java and F2.java; Paul needs to modify F3.java; John needs to modify F4.java. Suppose that you are the quickest to get it done and have finished modifying F1.java and F2.java. So you would like to run TestAll and then run the acceptance tests in passedTests and queuedTests. However, because Paul is editing F3.java and John is editing F4.java, the system just won't compile, not to mention running TestAll and the acceptance tests.

Therefore, sharing code directly this way doesn't work. Below we introduce a way that works.

Use CVS to share code

1. You install a software package called "CVS (Concurrent Version System)" on a server. On this CVS server you create a folder designed to share source code. Such a folder is called a "repository".

2. Copy the restaurantSystem folder into the repository.

3. Everyone downloads the restaurantSystem folder from the repository onto their own computer (may put it under c:\temp to become c:\temp\restaurantSystem, or put it anywhere else). This download action is called "checkout".

4. In order to implement user story 10, suppose that you need to make the following two acceptance tests pass: testAddOrder.test and testDeleteOrder.test in c:\temp\restaurantSystem\queuedTests. To do that, you think you need to modify F1.java and F2.java in c:\temp\restaurantSystem\src. For example, you change F1.java from:

```
public class F1 {
   public int foo(int x) {
      return x+10;
   }
}
```

   To:

```
public class F1 {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    public int foo(int x, int y) {
       return x+y;
    }
}
```

5. After the change, you run testAddOrder.test and testDeleteOrder.test in c:\temp\restaurantSystem\queuedTests and they pass. Therefore, you copy testAddOrder.test and testDeleteOrder.test from c:\temp\restaurantSystem\queuedTests into c:\temp\restaurantSystem\passedTests and then delete them.

6. Now, basically you can upload c:\temp\restaurantSystem back to the repository. This upload action is called "commit". However, before committing, you must run TestAll and all the acceptance tests in passedTests. Only when they all pass, can you commit.

7. When you commit, F1.java and F2.java will be uploaded. Because F3.java to F10.java have not been modified, they will not be uploaded. Similarly, testAddOrder.test and testDeleteOrder.test in c:\temp\restaurantSystem\passedTests will be uploaded. The testAddOrder.test and testDeleteOrder.test in restaurantSystem\passedTests in the repository will be deleted. In short, after you commit, the restaurantSystem in the repository will contain the same contents as c:\temp\restaurantSystem.

8. In order to implement user story 5, Paul has modified c:\temp\restaurantSystem\F3.java on his computer. For example, he has changed F3.java from:

```
public class F3 {
   public int bar() {
      F1 f1 = new F1();
      return f1.foo(10);
   }
}
```

   To:

```
public class F3 {
   public int bar() {
      F1 f1 = new F1();
      return f1.foo(20);
   }
}
```

9. Finally Paul succeeds in making the acceptance tests for user story 5 pass. Therefore, he is about to commit. If CVS allowed him to commit, his F3.java would be uploaded and he would not notice any problem. However, the F1.java in the repository would not work with his F3.java (a compile error would occur):

```
public class F1 {
   public int foo(int x, int y) {
      return x+y;
   }
}
public class F3 {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        public int bar() {
           F1 f1 = new F1();
           return f1.foo(20); //COMPILE ERROR!!!
        }
     }
```

Fortunately, CVS will not let him commit. When Paul tries to commit the whole c:\temp\restaurantSystem folder, CVS will find that Paul's F1.java is an older version of the one in the repository. Therefore, it will refuse to commit. Now what should Paul do? Before Paul commits, he should check if the repository contain any updated files. If yes, he should download them. This action of downloading the updated files is called "update". It is different from checkout. checkout performs the first download. Every download after that is an (incremental) update.

10. So Paul performs an update and downloads the updated versions of F1.java and F2.java. At the same time his passedTests folder will get two new files: testAddOrder.test and testDeleteOrder.test. Because F1.java and F2.java have been updated, he needs to compile again and run TestAll and passedTests again. However, currently the system simply won't compile. He has to fix the compile error first. How to do that? He can do it any way he wants, including reverting F1.java back to the original version. However, the precondition is that he must ensure that all the tests in passedTests continue to pass (because you committed first, passedTests now contains testAddOrder.test and testDeleteOrder.test).

11. Suppose that after Paul's hard work, he finally fixes all compile errors, makes TestAll and passedTests pass. He can try to commit. If before him John has committed, once again he will be unable to commit. He needs to do an update, fix all errors, make TestAll and passedTests pass and then try to commit again. From this we can see that we should commit as soon as possible. The person who commits late may have to clean up all the mess. In fact, even before Paul succeeds in making the acceptance tests for user story 5 pass, as long as TestAll and passedTests pass, he can commit to reduce the chances of having to clean up the mess. This way of frequently integrating code is called "continuous integration".

Different people have changed the same file

Suppose that in order to implement user story 5, Paul needs to modify not only F3.java, but also F1.java. For example, he changes F1.java from:

```
public class F1 {
   public int foo(int x) {
      return x+10;
   }
}
```

To:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
public class F1 {
   public int foo(int x) {
      return x+10;
   }
   public int bar(int x) {
      return x;
   }
}
```

When Paul updates, the F1.java in the repository is your version:

```
public class F1 {
   public int foo(int x, int y) {
      return x+y;
   }
}
```

If Paul hadn't changed F1.java, CVS would use your version of F1.java in the repository to overwrite his copy of F1.java (because your version is an update based on his version). Or, if Paul did change F1.java but the F1.java in the repository was still the old version, CVS would simply do nothing (because his version is an update based on the version in the repository). However, now CVS finds that Paul's F1.java is no longer the original F1.java and that the F1.java in the repository has also been updated. None of these two versions is an update based on the other. They are two different updates based on the original version. Therefore, CVS will merge the copy in the repository into Paul's local copy, making Paul's version the most updated version based on these two versions:

```
public class F1 {
   public int foo(int x, int y) {
      return x+y;
   }
   public int bar(int x) {
      return x;
   }
}
```

Of course, after that Paul needs to re-compile, make sure TestAll and passedTests continue to pass and then try to commit.

There are some cases in which CVS cannot merge two versions smoothly. For example, as supposed before, you have changed F1.java to:

```
public class F1 {
   public int foo(int x, int y) {
      return x+y;
   }
}
```

But Paul has changed F1.java to:

```
public class F1 {
   public int foo(int x, int y) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        return x+2*y;
    }
}
```

When CVS tries to merge, it finds that you and Paul have changed the same part of F1.java and therefore can't decide which version it should keep. In this case, the merged F1.java will be like:

```
public class F1 {
  public int foo(int x, int y) {
>>>>>>>>>
      return x+2*y;
<<<<<<<<<
     return x+y;
   }
}
```

Markers like "<<<<<<<" make F1.java fail to compile, advising Paul that he must manually check the code and decide which version to keep. When CVS cannot merge as shown in this case, we say there is a "conflict".

Add or remove files

If you add a new file like F11.java in c:\temp\restaurantSystem\src, when you commit, CVS will not automatically upload it. As a result, the system in the repository probably will not compile (because F11.java is missing). To make it include F11.java, you need to "add F11.java to CVS".

Similarly, if you delete F1.java from c:\temp\restaurantSystem\src, when you commit, CVS will not automatically delete the F1.java in the repository. To make it delete F1.java, you need to "remove F1.java from CVS".

It is very easy to forget to add files to or remove files from CVS. Stay alerted!

Different people have added the same file

Suppose that you have added F11.java and Paul has also added a file with the same name (F11.java). Suppose that you commit first. When Paul updates, CVS will check the relationship between Paul's F11.java and the F11.java in the repository. CVS will find that neither one is an update based on other, nor they are two different updates based on the same version. Instead, they are two completely independent files. In this case, CVS will not merge them. It will simply treat it as an error and refuse to update. To solve this problem, Paul may remove his F11.java from CVS, rename his F11.java (e.g., to F11Old.java), update again to get the F11.java from

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

the repository, manually merge F11Old.java into F11.java and then delete F11Old.java.

Collective code ownership

As mentioned above, you, Paul and John have the right to modify any files (F1-F11.java) in the system without getting approval from anyone as long as TestAll and passedTests continue to pass. Therefore, you, Paul and John collectively own all the code in the system. This is called "collective code ownership".

How to use command line CVS

There are various CVS clients. Some have a command line interface and some have a GUI (e.g., WinCVS). Here we will introduce how to use the command line CVS client in a Linux/Unix environment. Suppose that a repository is in the /var/cvs folder on a server with a DNS hostname c001.cpttm.local.

We need to perform the configurations below:

- Ensure that the developers can use ssh to login to c001.cpttm.local.
- Ensure that the developers have executed ssh-agent, input their passwords and are able to login to c001.cpttm.local repeatedly without inputting a password again. For the details, please see the man page of ssh-agent.
- CVS are installed on the computers used by the developers.
- An environment variable named "CVS_RSH" has been created and set to the value of "ssh" on those computers.
- An environment variable named "CVSROOT" has been created and set to the value of ":ext:c001.cpttm.local:/var/cvs" on those computers.

The commonly used commands are shown below:

checkout	cd /temp cvs checkout restaurantSystem

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

update	cd /temp cvs update restaurantSystem
commit	cd /temp cvs commit restaurantSystem
Add F11.java	cd /temp/restaurantSystem/src cvs add F11.java
Remove F11.java	cd /temp/restaurantSystem/src cvs remove F11.java

References

- Official web site of CVS: http://www.cvshome.org.
- http://www.c2.com/cgi/wiki?CvsTutorial.
- http://www.c2.com/cgi/wiki?ContinuousIntegration.
- http://www.c2.com/cgi/wiki?CollectiveCodeOwnership.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Problems

1. Suppose that you have checked out the system. In each of the following cases, what will CVS do?

   - You modify F1.java and then update.
   - You commit immediately.
   - You modify F1.java and then commit. However, Paul has committed first. He has modified F2.java.
   - You modify F1.java and then commit. However, Paul has committed first. He has modified F1.java.
   - You delete F1.java in Explorer and then commit.
   - You create a new file F12.java and then commit.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Sample solutions

1. Suppose that you have checked out the system. In each of the following cases, what will CVS do?

   - You modify F1.java and then update.

     CVS will do nothing.

   - You commit immediately.

     CVS will do nothing.

   - You modify F1.java and then commit. However, Paul has committed first. He has modified F2.java.

     CVS will refuse to commit and ask you to update first.

   - You modify F1.java and then commit. However, Paul has committed first. He has modified F1.java.

     CVS will refuse to commit and ask you to update first. When you update, if you and Paul has modified different parts of F1.java, CVS will merge the changes. If you two have modified the same part of F1.java, CVS will tell you that there is a conflict. In that case you will have to clean up the markers and merge the changes manually.

   - You delete F1.java in Explorer and then commit.

     CVS will do nothing. You should have removed it from CVS first.

   - You create a new file F12.java and then commit.

     CVS will do nothing. You should have added it to CVS first.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 15

Essential Skills for Communications

Different ways to communicate requirements

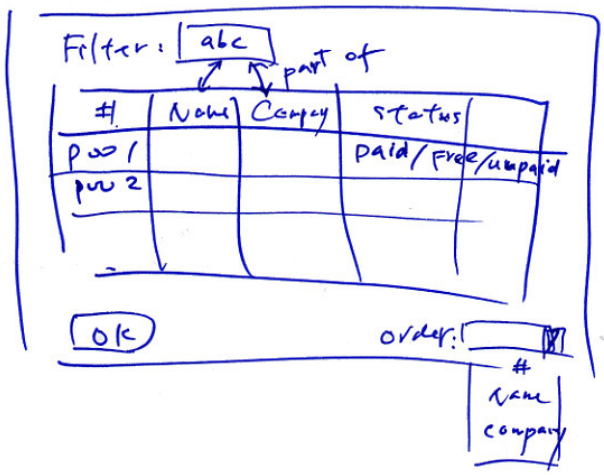
Suppose that we are developing an application for a customer. To do that, we must understand what the requirements are, e.g., what functions it should have or how it can be used to say input some particular data? There are different possible ways to communicate these requirements. For example:

1. We may have a system analyst talk to the customer to understand their requirements. Then the analyst writes down the requirements in a software requirements specification (SRS). The other developers will read the SRS to understand the requirements. If they don't understand something, they will fill out a requirement clarification request and send it to the system analyst. The system analyst may in turn email the customer for answers.

2. All the developers may talk to the customer directly. The requirements are then coded as automated acceptance tests. The customer must be always in the same room as the developers. If the developers have questions, they ask the customer face to face immediately.

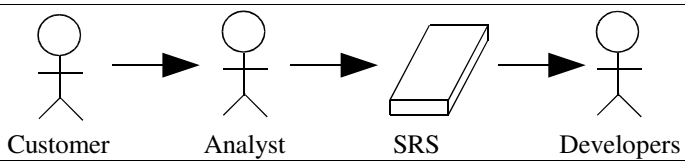
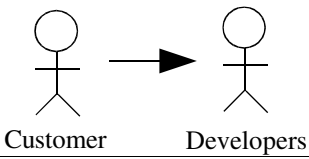
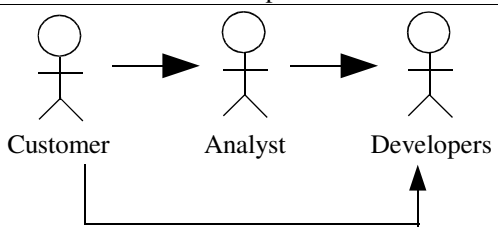
3. A system analyst may talk to the customer to have the major requirements. Then he writes a brief description of each feature on a notepad and brief the developers. For the details, all the developers will talk to the customer directly in regular weekly meetings. As the customer doesn't stay with the developers, between meetings they communicate by phone as required. The important information will be added to a flip chart (an example is shown below). The requirements are also coded as automated acceptance tests but with textual explanations.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



Which method above is the best way to communicate? It really depends the current situation. Method 1 is commonly called a "heavy" method because much of the communication must be done in writing and probably in a fixed format/template (e.g., SRS and requirement clarification request). It has some drawbacks:

First, the communication path between the customer and the developers is very long (i.e., very indirect) when compared to the other two methods (see below). The more indirect the path is, the more time is needed to communicate and the more errors will creep in.

Method 1	
Method 2	
Method 3	

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Second, it takes more time for the analyst to write the SRS and for the other developers to read the SRS then just letting them talk to one another.

Third, no analyst in the world can express the requirements that he knows about 100% completely and correctly through writing. Talking can't either but talking allows easy bi-directional communication, i.e., if the other developers don't understand something or find something ambiguous, they can ask the analyst immediately.

	Support for bi-directional communication
Method 1	Hard (must be done in writing).
Method 2	Easy (talk in the same room).
Method 3	Not as easy as method 2 but easier than method 1 (talk in weekly meetings or on the phone).

So it seems that method 2 (a "light" method) must be better than method 1? If the customer is in US but the developers are in India, then it is not going to work. If there are ten or less developers on the team, if the customer has some important thing to tell, he can simply tell it to all of them ("Hey, guys, listen up! I have something important to tell..."). But if there are more than ten developers (e.g., 20 or 100), they simply can't fit in a room. Even if they can (e.g., use a stadium), the customer simply can't tell something to all of them easily.

Method 2 is probably not the best for someone trying to understand the requirements in the future. If the developers are still there readily accessible, he can just ask them. However, if they are all gone, that person will need to go through the acceptance tests to understand the requirements. This should work, but some well written textual explanations would make it a lot easier for him. Let's compare:

Acceptance test without textual explanations	***Acceptance test with textual explanations***

`SystemInit`
`AddParticipant,p001,Paul`
`AddParticipant,p002,John`
`AddParticipant,p003,Mary`
`AddSeminar,s001,How to go`
`agile?,1000,2`
`EnrolSeminar,s001,p001`
`EnrolSeminar,s001,p002`
`ExpectError,EnrolSeminar,s001,p003`
	`//Make sure no enrollment is`
	`//accepted if a seminar is`
	`//already full.`
	`SystemInit`
	`AddParticipant,p001,Paul`
	`AddParticipant,p002,John`
	`AddParticipant,p003,Mary`
	`//Seminar has only 2 seats.`
	`AddSeminar,s001,How to go`
	`agile?,1000,2`
	`EnrolSeminar,s001,p001`
	`EnrolSeminar,s001,p002`
	`//The third enrollment.`
	`ExpectError,EnrolSeminar,s001,p003`

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

This is because the acceptance test contains much irrelevant data. For example, what is relevant above is that the seminar has only 2 seats and that there are already 2 enrollments. The other data is simply irrelevant. Without some text explanations a reader won't know what data is relevant and what is not. So it takes more time and effort to understand the test. In contrast, the text explanations can point out the relevant data and thus can speed up the understanding process.

Method 3 is somewhere between the method 1 and 2. When compared to method 1, it requires less formality (e.g., use notepad and flip chart instead of an SRS) and encourages face to face (weekly meetings) or oral (phone) communication. When compared to method 2, weekly meetings and oral communication are not as good as constant face to face communication. However, it may make sense if the customer doesn't work in the same building as the developers. It also adds explanations to the acceptance tests. Must this be better? Not necessarily. Adding explanations takes time and effort. If you change the acceptance test, you must also update the explanations. So, you must weigh in the costs and benefits for your team and project.

There are method 4, 5, 6 and lots more subject to your imagination and validation. For example, if you can afford something better than weekly meeting, try changing to daily meeting. Or if it is the opposite, you can change it to bi-weekly meeting. Instead of phone, you may use NetMeeting or ICQ depending on your actual situation.

In summary:

- Do not create documentation just because it feels right. Do it only for the purpose of communication. Even then, there are other ways to communicate and documentation may not be the best way.
- Face to face communication (with a white board or flip chart) is much more effective than documentation, unless the team is quite large or the participants can't come together (not in the same place or not at the same time).
- Keep the communication as frequent as possible.
- Keep the communication path as short as possible.
- Communicate relevant information only.

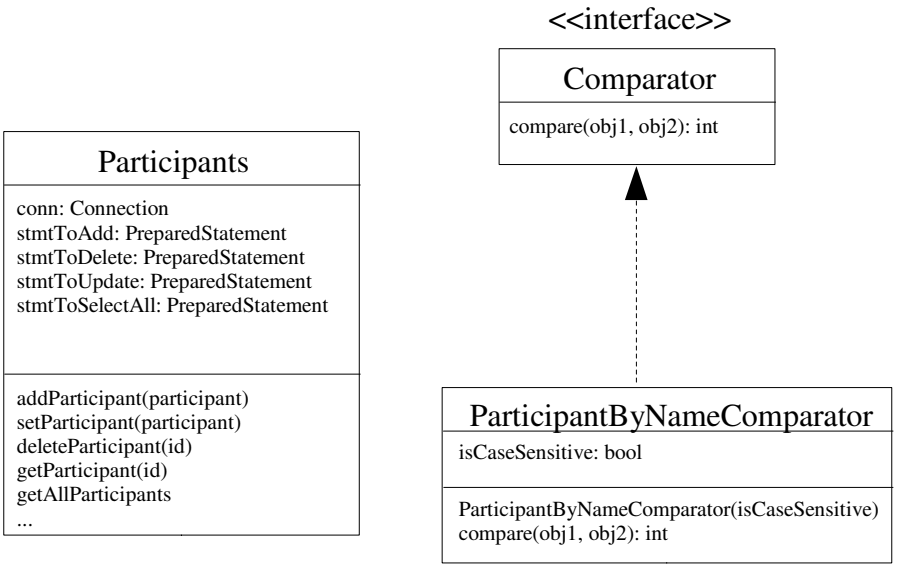
How to communicate designs

During the development, the developers need to communicate the designs of the system. Just like communicating requirements, there are different ways to communicate designs. For example, regarding the design to support the user story of listing the conference participants in

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

a grid, there may be different ways to communicate it:

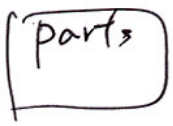
1. The developers can draw a detailed "UML class diagram" like:



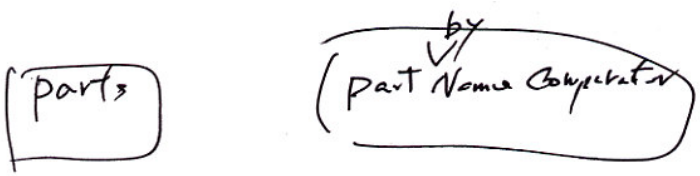
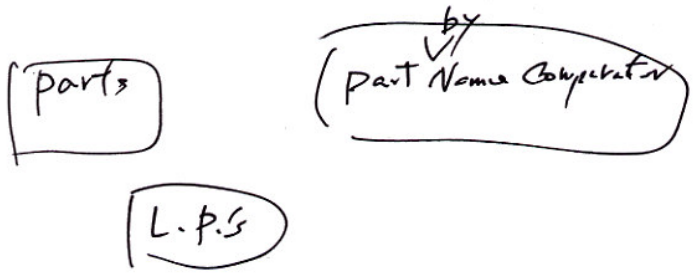
   In a UML class diagram, the symbols are well defined. For example, the dotted arrow shown above means "implement an interface".

2. The developers can make their code very easy to understand and make the design clear from the code. If they need to communicate the design to someone, they let him look at the code and explain what it does. They also show him the client code or the unit tests (just like another client) and explain how a client may use it.

3. The developers can talk about the design while drawing on the white board:

Narration	***White board***
To list the participants, we need to read the participants from the database. This is done by the Participants class.	

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Narration	*White board*
We need to sort them according to their names. All we need is a Comparator. Let's call it ParticipantByNameComparator.	
Then we need a TableModel. Let's call it ListedParticipants and assume each row is a ListedParticipant (without "s" at the end).	

Note that what is said may not be drawn exactly on the white board. For example, the Participants class is simply written as Parts; the ListedParticipants class is simply written as L.P.'s. The ListedParticipant class is simply not drawn at all.

4. The developers can use CRC cards. It is like method 3 but the cards can be moved around to simulate the collaboration.

Which method above is the best? Again, it depends on the situation.

Method 1 is the heaviest method above. It includes a lot of details. Some of the details such as the addParticipant method are irrelevant to the question at hand (how to list the participants). This makes it much harder to understand. It is not much different from just looking at the source code.

Method 2 is the lightest. Code plus oral explanations work very well. However, if it is the maintenance developer who is trying to understand the design, then he will have to get hold of one of those developers. If no one is available, then the oral explanation part will be missing and it will become much harder to understand the code alone. Therefore, if you can foresee who the maintenance developer is, then get him working on the team for the last several months so that the knowledge on the design is transferred.

Method 3 and 4 are also quite light. They are useful when the code is not yet written and thus method 2 cannot be applied. For example, when we need to discuss the design just before we implement the user story. In addition, if the target audience is a maintenance developer who

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

will never be known until after the release of the system, we can keep a copy of the drawings using a flip chart or an electronic white board, or even record the discussions with audio tapes and tape the CRC sessions. Then if the maintenance developer is trying understand how a user story is implemented, he can quickly lookup the relevant design information. However, if the team is large (more than ten people), this method no longer works well and we should find a middle ground with the formal UML class diagrams and the informal class drawings or CRC cards. For example, keep the formal meaning of the symbols, forbid the use of short form for the names and draw the relevant methods and attributes only.

References

- http://www.xprogramming.com/xpmag/docIndex.htm.
- http://www.agilemodeling.com/essays/communication.htm.
- http://www.agilemodeling.com/essays/agileDocumentation.htm.
- Alistair Cockburn, Agile Software Development, Addison-Wesley, 2001.
- Dean Leffingwell, Don Widrig, Managing Software Requirements: A Unified Approach, Addison-Wesley, 1999.
- Jim Highsmith, Agile Software Development Ecosystems, Addison-Wesley, 2002.
- Karl E. Wiegers, Software Requirements, Second Edition, Microsoft Press, 2003.
- Kent Beck, Extreme Programming Explained, Addison-Wesley, 2000.
- Martin Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition, Addison-Wesley, 2003.
- Scott W. Ambler, The Object Primer 3rd Edition: Agile Model Driven Development with UML 2, Cambridge University Press, 2004.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

CHAPTER 16

Pair Programming

How two people program together

Suppose that Andy, one of our developers, is starting to work on improving the quality of an existing system developed by another team. He joined that team some time ago, so he knows the system fairly well. However, he seems to get stuck. Kent, one of our more senior developers, notices that Andy is doing nothing at his computer and the puzzled look on his face.

Kent: "Hi Andy, what are you doing?"

Andy: "I am trying to get rid of the DataAccesser. But I am not sure how to go about it."

Kent is not very familiar with this system. He asks: "What is DataAccesser? Let me take a look." At the mean time he grabs a chair and sits side by side with Andy.

Side note: Andy and Kent are now programming together using one computer. This is called "pair programming".

Andy opens the DataAccesser class in the IDE. Kent is now faced with lots of code. Kent: "What does it do?"

Andy: "It is used to access the database."

This explanation is too vague. Kent has to look at the fields and methods. Kent: "Let's take a look at its fields."

Side note

When explaining some existing design or code, it is best to show the code first and explain on the way. This is far better than explaining just in words.

Andy points to the courseData field on the screen with his finger:

```
public abstract class DataAccesser {
   private CourseData courseData;
   private Table table;
   ...
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

and says: "CourseData is a weird thing. It contains all the data in the system. I really HATE it! OK, let take a look at it." So he opens the CourseData class in the IDE:

```
public class CourseData {
   ...
   private Students students;
   private Courses courses;
   private Enrollments enrollments;
   private IdCounter idCounter;
   ...
}
```

Then he points to the students field:

```
public class CourseData {
   ...
   private Students students;☜ ☜ ☜
   private Courses courses;
   private Enrollments enrollments;
   private IdCounter idCounter;
   ...
}
```

and explains: "For example, students inherits from DataAccesser and represents all the students in the database." At the same time he opens the Students class in the IDE:

```
public class Students extends DataAccesser {
   ...
}
```

Returning to the CourseData class and pointing to the courses field:

```
public class CourseData {
   ...
   private Students students;
   private Courses courses;☜ ☜ ☜
   private Enrollments enrollments;
   private IdCounter idCounter;
   ...
}
```

Andy continues: "Similarly, it represents all the courses in the database; The enrollments field represents all the enrollments in the database and etc."

Kent: "OK, I see. Let's return to DataAccesser."

Andy shows DataAccesser again. Kent points to the table field in DataAccesser:

```
public abstract class DataAccesser {
   private CourseData courseData;
   private Table table;☜ ☜ ☜
   ...
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

and asks: "What's this?"

Andy: "The Table class is simple. It contains the table name and the names of its fields."

Kent: "OK." Now Kent understands that a DataAccesser has a Table and can reference all the other DataAccessers in the system. Kent continues to ask: "What methods it has?"

Andy scrolls down from start to end to find an interesting method and finds one. He points to the deleteAll method:

```
public void deleteAll() throws DataAccessException {
   Vector refs = getRefsAccessersForDeleteAll();
   if (refs != null)
      for (int i = 0; i < refs.size(); i++)
          ((DataAccesser) refs.elementAt(i)).deleteAll();
   PreparedStatement st;
   try {
      st =
         getConnection().prepareStatement("DELETE FROM " + table.getName());
      try {
         executeUpdate(st);
      } finally {
         st.close();
      }
   } catch (SQLException e) {
      throw new DataAccessException(e);
   }
}
```

and says: "For example, the deleteAll method deletes all records in the table."

The call to the getRefsAccessersForDeleteAll method is drawing Kent's attention. The rest of the method is pretty familiar to him. He points to the line:

```
public void deleteAll() throws DataAccessException {
   Vector refs = getRefsAccessersForDeleteAll();☜ ☜ ☜
   if (refs != null)
      for (int i = 0; i < refs.size(); i++)
          ((DataAccesser) refs.elementAt(i)).deleteAll();
   ...
}
```

and asks: "What's this?"

Andy: "It gets all the other DataAccessers that refer to this DataAccesser." Andy points to the loop:

```
public void deleteAll() throws DataAccessException {
   Vector refs = getRefsAccessersForDeleteAll();
   if (refs != null)
☞ ☞ for (int i = 0; i < refs.size(); i++)
☞ ☞     ((DataAccesser) refs.elementAt(i)).deleteAll();
   ...
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }
```

and explains: "Here it will call deleteAll on those DataAccessers first."

Kent doesn't really understand what Andy said. So he asks: "Would you give an example?"

Side note

When we don't understand what someone is saying, the best thing to do is to ask him for an example. This is probably the most important skill in communication (and pair programming).

Andy: "Sure. For example, if you are going to delete all students, because there may be enrollments referring to the students, it will delete all the enrollments first."

Now Kent comes to conclude that a DataAccesser represents a table in a database. The table is "smart" in that it can perform cascade deletes. "I see. Let's look at another method." says Kent.

Andy: "OK." Then he continues to scroll down to find another interesting method. "Here you are." He points to a method named "update":

```
public int update(Object[][] fieldsAndValues, Object[][] keyFieldsAndValues)
   throws DataAccessException {
   try {
      PreparedStatement st =
         getConnection().prepareStatement(
            "UPDATE "
               + table.getName()
               + " SET "
               + getParameterString(fieldsAndValues)
               + getConditionString(keyFieldsAndValues, "="));
      try {
         try {
            setParameters(st, fieldsAndValues, 1);
            setParameters(
               st,
               keyFieldsAndValues,
               fieldsAndValues.length + 1);
         } catch (InvalidArgumentException e1) {
            e1.printStackTrace();
         }
         return executeUpdate(st);
      } finally {
         st.close();
      }
   } catch (SQLException e) {
      throw new DataAccessException(e);
   }
}
```

Andy explains: "This method updates some records in the table." He goes on to point to the fieldsAndValues and keyFieldsAndValues parameters:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
public int update(Object[][] ☞fieldsAndValues, Object[][] ☞keyFieldsAndValues)
   throws DataAccessException {
   ...
}
```

and says: "They are something that I hate very much. Let me show you how they are used." Then he uses the IDE to search for a call to the update method. "Got it. Take a look at this.", he says.

```
private void updateStudentAnyway(Student student) throws DataAccessException {
   Object[][] fieldsAndValues = { {
         StudentsTable.eFirstNameField, student.getEFirstName()}, {
         StudentsTable.cFirstNameField, student.getCFirstName()
      }, {
         StudentsTable.eLastNameField, student.getELastName()
      }, {
         StudentsTable.cLastNameField, student.getCLastName()
      }, {
         StudentsTable.idTypeField, student.getIdType()
      }, {
         StudentsTable.idNoField, student.getIdNo()
      }, {
         StudentsTable.nationalityField, student.getNationality()
      }, {
         StudentsTable.genderField, new Boolean(student.isMale())
      }, {
         StudentsTable.birthDateField, student.getBirthDate()
      }, {
         StudentsTable.regionField, student.getRegionId()
      }, {
         StudentsTable.eAddressField, student.getEAddress()
      }, {
         StudentsTable.cAddressField, student.getCAddress()
      }, {
         StudentsTable.telField, student.getTel()
      }, {
         StudentsTable.mobileField, student.getMobile()
      }, {
         StudentsTable.faxField, student.getFax()
      }, {
         StudentsTable.emailField, student.getEmail()
      }
   };
   Object[][] keyFieldsAndValues = { { StudentsTable.studentNoField,
student.getStudentNo()}
   };
   update(fieldsAndValues, keyFieldsAndValues);
}
```

Andy: "Look, this method is calling update to update a student." Then he points to:

```
private void updateStudentAnyway(Student student) throws DataAccessException {
☞  Object[][] fieldsAndValues = { {
         StudentsTable.eFirstNameField, student.getEFirstName()}, {
         StudentsTable.cFirstNameField, student.getCFirstName()
      },
   ...
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

and says: "It is a 2D array holding the fields to be set and their new values." Then he points to:

```
private void updateStudentAnyway(Student student) throws DataAccessException {
   Object[][] fieldsAndValues = { {
      ...
   };
☞ Object[][] keyFieldsAndValues = { { StudentsTable.studentNoField,
student.getStudentNo()}
   };
   update(fieldsAndValues, keyFieldsAndValues);
}
```

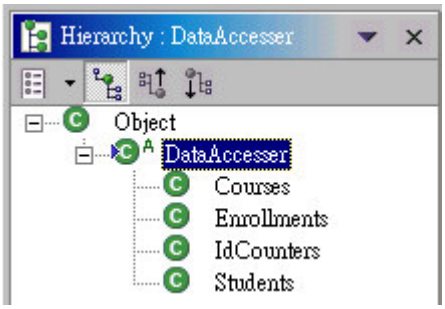
and says: "It is also a 2D array. It specifies the SQL condition. The 2D arrays suck."

Kent agrees that the 2D arrays suck and is thinking to replace each with a list of field-value pairs. "Right, we'll get rid of them." He thinks that he has seen enough of the code. "So, you'd like to get rid of DataAccesser?"

Andy: "Yeah, I think it is ugly." He is waiting for Kent's advice.

After thinking for a while about to get rid of DataAccesser, Kent says: "OK. Let's take a look at the simplest class that is extending DataAccesser."

Andy: "OK." He right clicks DataAccesser and choose "Open Type Hierarchy" in the IDE and instantly all the derived classes of DataAccesser are listed:



Kent is impressed by this powerful feature of the IDE. He keeps it in mind for future use.

Side note
Knowledge has just been transferred. As shown in this case, a more experienced developer can still learn from a less experienced one.

Andy points to a class named "IdCounters" and says "This should be the simplest one." Then he opens it in the IDE.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Kent browses the code of IdCounters and finds that it only uses a few methods of DataAccesser. "OK. Let's do it. Let's create a DBTable class."

Andy doesn't understand what Kent is trying to do. "OK, but what are you trying to do?"

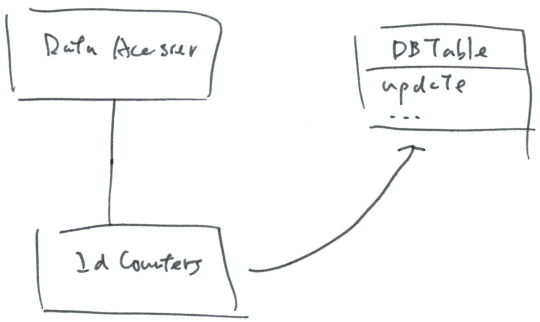
Kent: "At the moment IdCounters is inheriting DataAccesser. Right?" Kent draws a sketch on a piece of scrap paper:

Side note

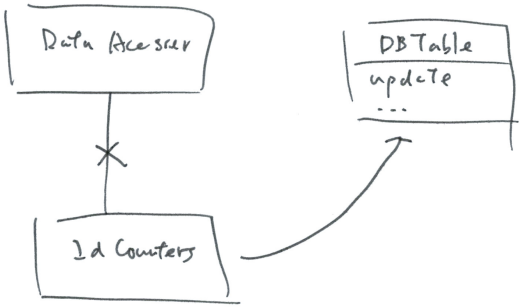
Kent is trying to show Andy how the design in his mind works. Usually, the best way to communicate a design is to show him the code and explain on the way. However, as there is no code to show yet, diagrams can be a good tool in explanations. Note that the diagram above looks like UML but doesn't really obey the UML rules (e.g., in UML inheritance is represented by a line ended with a hollow arrow). This is OK because the inheritance relationship is already clearly spoken in the conversation.

He continues: "I'd like to create a DBTable class to provide the functionality of DataAccesser and let IdCounters use a DBTable instead of inheriting DataAccesser." He continues to draw:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



"In the process I don't want to change DataAccesser at all. When IdCounters no longer uses any of DataAccesser's service, we can break the inheritance." He crosses out the inheritance line in the drawing:



Side node
Again, UML rules are not obeyed. There is no such a cross symbol in UML to mean canceling an inheritance. But it is so obvious to an average person.

Andy now feels much more happier because he knows that he can replace the reliance on DataAccesser bit by bit. He is excited: "Great! Let's do it."

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Side note

- *Andy and Kent have just performed some design together. In pair programming, designing together is one of the major activities. They have also worked out a refactoring strategy together. Andy offered his knowledge on the classes in the system and suggested that DataAccesser was ugly and should be replaced (what to refactor). Based on Andy's knowledge Kent suggested the strategy to reduce the reliance on DataAccesser bit by bit by migrating the clients to use DBTable gradually (how to refactor).*

- *In pair programming, different people with different knowledge/skills can combine their skills to solve a single difficult problem. For example, Andy knows the system but not as good in OO and refactoring, while Kent is better in OO and refactoring but knows little about the system. Obviously, letting either of them refactor the system alone will be extremely difficult. But if they work together, their knowledge is combined and refactoring the system becomes much easier. The above scenario is just one kind of fruitful combination. There are others such as: Database administrator plus programmer (optimizing code to access the database), UI designer plus programmer (developing UI), fellow programmer plus fellow programmer (one is developing code that uses the code developed by the other), junior programmer plus designer/architect (implementing the design or checking if the design works) and etc.*

- *In pair programming, knowledge and best practices are frequently transferred. Now Andy knows more about how to refactor in small steps and Kent knows more about the existing system.*

- *In pair programming, programmers are more confident and happier. For example, Andy was very worried at the beginning. After pairing with Kent, he is much happier. In fact, Kent also feels more confident after having Andy double check his ideas and show the way around the system.*

Kent: "Good. Let's create the DBTable class."

Andy looks puzzled. He asks: "Shouldn't we start with a failing test first?"

Kent: "Ah, you're right! Let's do the test first."

So Andy creates a DBTableTest class:

```
public class DBTableTest extends TestCase {
}
```

Side note

Andy and Kent are going to write code to test DBTable. Right, in addition to designing together, testing together is another major activity in pair programming.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Andy starts typing a method to test the deleteAll method. He types "public", and then a method name starting with "test":

```
public class DBTableTest extends TestCase {
   public test
}
```

Kent points to the location:

```
public class DBTableTest extends TestCase {
   public ☞ test
}
```

and remind him: "void is missing."

Side note

Kent detected and corrected the bug immediately after it was created. Right, in addition to designing together and testing together, debugging together is another major activity in pair programming.

Andy promptly corrects the error and then continues:

```
public class DBTableTest extends TestCase {
   public void test
}
```

Andy: "test what? How about testDeleteAll?"

Kent: "Fine." Andy types it in:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
   }
}
```

Andy: "Then we should create a DBTable."

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
   }
}
```

Andy: "Then we can call deleteAll."

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.deleteAll();
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Kent has made no comments for a while. He has been thinking about how to test deleteAll and has come to a solution.

Andy: "So, how to test it?"

Kent: "Let's create an interface to simulate the database server. It will can execute an SQL statement sent to it."

Andy has never seen unit testing done this way. Therefore he doesn't really understand what Kent is saying. He says: "Sorry, I don't understand."

Kent: "To test DBTable, it seems that we need to setup a database and a test table. This is troublesome. In addition, running the tests will take a long time. Therefore, we should instead simulate the database."

Andy still doesn't really understand because the description is too abstract. He says: "I am really sorry, but I still don't understand."

Kent says: "It's OK. Just type what I say. You will understand very soon. Now, create an interface named DBServer." Andy creates it as told:

```
public interface DBServer {

}
```

and is ready to type the first method signature.

Side note

If we just can't communicate a proposed design to another person, it is best to just type the code. Who should type the code? Let the weaker partner do the typing. If Kent had done the typing, Andy would have complained about not knowing what had been going on or simply have fallen asleep.

Kent says: "public, int, executeUpdate, parenthesis." Andy types accordingly:

```
public interface DBServer {
   public int executeUpdate()
}
```

Note that the IDE automatically adds the closing parenthesis. Kent continues, "string, SQL. That's it." Andy types accordingly:

```
public interface DBServer {
   public int executeUpdate(String sql);
}
```

Kent says: "Return to the test." Andy does it accordingly:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.deleteAll();
   }
}
```

Kent points to the line:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
☞ ☞ table.deleteAll();
   }
}
```

and says: "Add a line before it." Andy does it accordingly. Kent continues, "table, dot, setDBServer, parenthesis." Andy types accordingly:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.setDBServer()
      table.deleteAll();
   }
}
```

Kent continues, "new, DBS, auto-complete, parenthesis." Kent is using the auto-complete feature of the IDE to expand "DBS" to "DBServer". Andy uses the auto-complete feature day in and day out, so he has no trouble using it. Actually "DBS" matches a few other classes the system like DBSanityChecker, DBSuperUser, but Andy quickly chooses DBServer without being told at all:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.setDBServer(new DBServer())
      table.deleteAll();
   }
}
```

It means that Andy is actively engaged and is following closely what Kent is trying to do.

Kent points to the following location:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.setDBServer(new DBServer(☞))
      table.deleteAll();
   }
}
```

and says: "Go to here. Auto-complete." Kent is using the auto-complete feature to generate a

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

skeleton implementation of an interface. Andy didn't know that it can be used this way, but he does it anyway:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            return 0;
         }
      })
      table.deleteAll();
   }
}
```

"Wow! This is great! Only if I knew it earlier!", Andy is impressed.

Side note

Knowledge is transferred again. After learning this trick, Andy's productivity is increased for the rest of his life. If he later pairs with other colleagues, he will also transfer this trick to them. This way, knowledge is spread throughout the whole organization.

Kent points to the line:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
☞ ☞ ☞ ☞  return 0;
         }
      })
      table.deleteAll();
   }
}
```

and says: "Add a line before it." Andy does it accordingly:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {

            return 0;
         }
      })
      table.deleteAll();
   }
}
```

Kent continues, "assertEquals, sql, double quote, delete, from."

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from")
            return 0;
         }
      })
      table.deleteAll();
   }
}
```

Kent hesitates for a second and then points to the location:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable();☜ ☜ ☜
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from")
            return 0;
         }
      })
      table.deleteAll();
   }
}
```

and says: "Pass a string 'abc' to it." Andy does it accordingly:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable("abc");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from")
            return 0;
         }
      })
      table.deleteAll();
   }
}
```

Kent points to the location:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable("abc");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from")☜ ☜ ☜
            return 0;
         }
      })
      table.deleteAll();
   }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

and says: "from, space, abc." Andy types accordingly and now understands that "abc" is the table name:

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable("abc");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from abc");
            return 0;
         }
      })
      table.deleteAll();
   }
}
```

Andy asks: "Why not call it 't'? I think it is better than 'abc'." Kent think "t" is indeed more descriptive than "abc". So, he says: "Yeah, that's a good idea." So, Andy changes it to "t":

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
         }
      })
      table.deleteAll();
   }
}
```

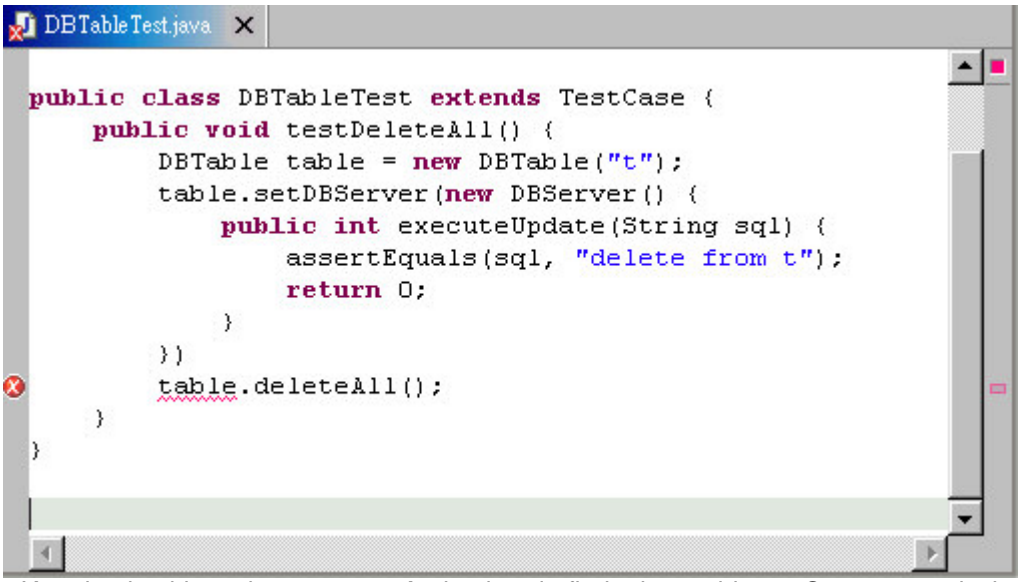
Side note

- *After typing all the code dictated by Kent, Andy finally understands how to use DBServer to simulate a database to unit test DBTable. This is a technique called "mock objects". Knowledge is transferred once again.*

- *Andy and Kent just made another design decision together regarding the table name. But this time it was Andy who made the suggestion. It means that even though one person may be very good at something (e.g., design), there is still room for others to make significant contributions.*

However, there is a syntax error detected by the IDE:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



While Kent is checking what's wrong, Andy already finds the problem: "Oops, a semicolon is missing!" So he corrects it:

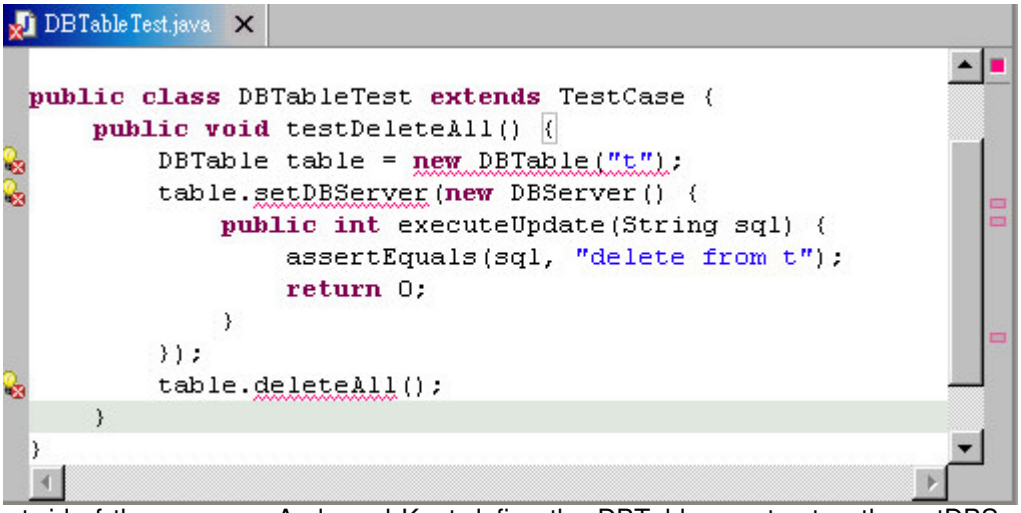
```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
         }
      });
      table.deleteAll();
   }
}
```

Side note

This time Andy found the bug first. It means that being more experienced doesn't necessarily mean that he can always find the bug faster. Each person is good at detecting certain types of bugs but poor at others. Fortunately, the "blind zone" of one person usually doesn't overlap much with that of another. For example, Andy didn't notice that he missed "void" when creating the testDeleteAll method, but this was very obvious to Kent. In contrast, Kent didn't notice that a semicolon was missing, but this was quite visible to Andy. Without a pair partner, it would have taken much longer for each of them to find the problems.

Then the IDE detects other syntax errors:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org



To get rid of these errors, Andy and Kent define the DBTable constructor, the setDBServer method and an empty deleteAll method:

```
public class DBTable {
   private String tableName;
   private DBServer dbServer;

   public DBTable(String tableName) {
      this.tableName = tableName;
   }
   public void setDBServer(DBServer dbServer) {
      this.dbServer = dbServer;
   }
   public void deleteAll() {
   }
}
```

They run the test and it passes!

Side note
As something unusual has occurred, they will debug together again.

Kent has made this kind of mistakes before, so he finds the problem in just a few seconds. He points to the line:

```
public void testDeleteAll() {
☞  DBTable table = new DBTable("t");
   table.setDBServer(new DBServer() {
      public int executeUpdate(String sql) {
         assertEquals(sql, "delete from t");
         return 0;
      }
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
      });
      table.deleteAll();
   }
```

and says: "Add a line before it." Andy does it accordingly:

```
   public void testDeleteAll() {

      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
         }
      });
      table.deleteAll();
   }
```

Then Kent says: "final, StringBuffer, callLog, equal, new, StringBuffer" and Andy types accordingly:

```
   public void testDeleteAll() {
      final StringBuffer callLog = new StringBuffer();
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
         }
      });
      table.deleteAll();
   }
```

Then Kent points to the line:

```
   public void testDeleteAll() {
      final StringBuffer callLog = new StringBuffer();
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
☞ ☞ ☞    assertEquals(sql, "delete from t");
            return 0;
         }
      });
      table.deleteAll();
   }
```

and says: "Add a line before it." Andy does it accordingly:

```
   public void testDeleteAll() {
      final StringBuffer callLog = new StringBuffer();
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {

            assertEquals(sql, "delete from t");
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
            return 0;
         }
      });
      table.deleteAll();
   }
```

Kent says: "callLog, dot, append, double quote, x, double quote" and Andy types accordingly:

```
   public void testDeleteAll() {
      final StringBuffer callLog = new StringBuffer();
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
         }
      });
      table.deleteAll();
   }
```

Kent points to the following line:

```
   public void testDeleteAll() {
      final StringBuffer callLog = new StringBuffer();
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
         }
      });
      table.deleteAll();☜ ☜ ☜
   }
```

and is about to tell Andy to add a line after it, but Andy already knows what he is trying to do and add the line by himself:

```
   public void testDeleteAll() {
      final StringBuffer callLog = new StringBuffer();
      DBTable table = new DBTable("t");
      table.setDBServer(new DBServer() {
         public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
         }
      });
      table.deleteAll();
      assertEquals(callLog.toString(), "x");
   }
```

Side note

After typing the code, Andy now understands how to use a call log to track the call sequence. Knowledge is transferred once again.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

They run the test and now it fails. They feel that they have made good progress.

> ***Side note***
>
> *Now they are about to implement deleteAll in DBTable. Right, in addition to designing together and testing together, coding together is another major activity in pair programming.*

Andy says: "OK, let's copy the code from DataAccesser."

```
public void deleteAll() throws DataAccessException {
   Vector refs = getRefsAccessersForDeleteAll();
   if (refs != null)
      for (int i = 0; i < refs.size(); i++)
          ((DataAccesser) refs.elementAt(i)).deleteAll();
   PreparedStatement st;
   try {
      st =
         getConnection().prepareStatement("DELETE FROM " + table.getName());
      try {
         executeUpdate(st);
      } finally {
         st.close();
      }
   } catch (SQLException e) {
      throw new DataAccessException(e);
   }
}
```

Andy deletes the code about cascade deletes and then uses DBServer instead of the JDBC connections and prepared statements:

```
public void deleteAll() {
   dbServer.executeUpdate("DELETE FROM " + tableName);
}
```

Then they run the test again, but it still fails! Kent points to the line:

```
public void testDeleteAll() {
   final StringBuffer callLog = new StringBuffer();
   DBTable table = new DBTable("t");
   table.setDBServer(new DBServer() {
      public int executeUpdate(String sql) {
         callLog.append("x");
         assertEquals(sql, "delete from t");
         return 0;
      }
   });
   table.deleteAll();
☞  assertEquals(callLog.toString(), "x");
}
```

and says: "Let's set a breakpoint here." However, Andy suddenly shouts, "Ah! The character cases are different! The test is using lowercase but the code is using uppercase!" So they change the test to:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
public void testDeleteAll() {
   final StringBuffer callLog = new StringBuffer();
   DBTable table = new DBTable("t");
   table.setDBServer(new DBServer() {
      public int executeUpdate(String sql) {
         callLog.append("x");
         assertEquals(sql.toLowerCase(), "delete from t");
         return 0;
      }
   });
   table.deleteAll();
   assertEquals(callLog.toString(), "x");
}
```

Then they run the test and it passes. Now the task is done and they run all the existing tests. During this time, they take a break to go get a drink (also done together because they are now a better team).

Side note

- Because pair programming is a mentally intensive exercise, regular breaks are good for the productivity.
- After performing this pair programming session, both Andy and Kent are familiar with this part of the system (DBTable, DBServer, etc.). If anyone of them is on vacation or leaves the company, the other can still maintain it. It means pair programming enhances the company's resistance to staff turn-over.

Later, another colleague John comes over and asks: "Andy, I'll start to work on the transcript printing story. I know that you wrote the part about the inputing of marks, would you like to pair with me?"

Andy: "Sure." At the same time Andy moves to pair with John. Kent is now free to pair with someone else.

Side note

- *It is good to switch parters regularly (probably at the start of a new task). Now Andy will further transfer the knowledge he learned to John. He will also learn something from John (e.g., how transcript printing works).*
- *As Andy knows how the marks are saved, he can probably help in retrieving the marks to print transcripts. Therefore, it is wise for John to ask Andy to pair with him to work on this task.*

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Summary on the benefits of pair programming

- Combine different knowledge to tackle a difficult task.
- Knowledge transfer.
- Efficient bug detection and removal.
- Programmers are happier.
- Resistance to staff turn-over.

Summary on the skills for pair programming

- Use code to explain existing designs.
- Use examples to explain.
- Use diagrams to explain proposed designs.
- If you just can't communicate a proposed design, type the code in.
- Let the weaker partner do the typing to keep him engaged.
- Take a break regularly.
- Switch partners regularly.

Do we have to double the number of programmers

Even though pair programming has quite many benefits, if we let two programmers work on a single task, do we have to double the number of programmers?

A research conducted by Laurie Williams shows that in a particular university environment but otherwise without any particular arrangements, it takes 15% more time for a pair to do as much work as two individuals working alone. It means we don't have to double the number of programmers, but just 15% more programmers or let the same number of programmers work for 15% longer.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

So, on one hand pair programming has some great benefits, but at the same time it needs 15% more development time. Is it worth it? The above research shows that the software delivered by a pair contains 15% fewer bugs than that of the individuals. These bugs will have to be fixed. Based on industry statistics, Alistair Cockburn and Laurie Williams conclude that the extra time the individuals will take to fix those bugs is 15-60 times of that 15% extra development time spent by the pair. So, just one benefit of pair programming (more efficient bug removal) alone already justifies the 15% increase in development time.

When will pair programming not work

Pair programming doesn't always work. Because it requires two people communicating and making decisions together, if somehow communication or decision making is not happening, then pair programming will not work.

When will communication not happen? For example, suppose that Kent is busying working hard to meet the deadline for his project, but the director of development Paul insists that he help Andy. So Kent reluctantly pairs with Andy. He asks Andy to show him the code. After seeing that, although he is not happy with DataAccesser either, he is so concerned with his own deadline that he says: "Oh my! It is a lot of work to remove it. You can simply rename it to DBTable." Andy says: "No! The current system is so database centric that it is too difficult to work on it any more. We must do something! Paul says that you would help me...' After that, when Andy is typing, Kent is not paying attention at all. He is not giving any advice either.

In this case above, communication is not happening because Kent is reluctant to pair with Andy. They seem to be pairing but in fact they are not. They don't share a common goal (i.e., improving the quality of that system). When people don't share a common goal, they have no incentive to communicate.

This is not the only possible reason for lack of communication. For example, if a partner is emotionally rejecting everything communicated, then there is no communication. When Andy and Kent were working on the testDeleteAll method, Andy suggested to use "t" instead of "abc" as the table name. What if Kent had responded like this: "abc is just fine. I have been using this kind of constants in tests for years. How dare you challenge me? You are just a fresh graduate. What do you know? When I started programming, you were still in kindergarten."

The problem may also occur at the sending end of the communication path. That is, a partner lacks confidence so much that he doesn't dare to raise anything. When Andy and Kent were starting to write DBTable, Andy (rightly) questioned Kent why he suggested writing DBTable without writing a failing test first. What if Andy had been so impressed by Kent's seniority and OO skills that he hadn't questioned him? Then the system would not have benefited from his valid suggestions.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Lack of confidence is just one reason for not raising anything. Another one is to avoid looking stupid. When Andy was explaining to Kent about the cascade delete behavior in the deleteAll method of DataAccesser, Kent didn't understand his explanation. So he asked Andy to give an example. What if Kent had felt that he would look stupid to not understand such a simple method? Then Kent would not have understood the system to suggest a good way to refactor it.

In summary, here are some common problems that can stop pair programming from working:

- Reluctant pairing.
- Rejecting every single suggestion from the partner or attacking or demeaning the partner.
- Not making suggestions to avoid "challenging" the "gurus".
- Not asking questions to avoid looking stupid.

What if these problems really occur? As these have more to do with management and personality than technical skills, the best way is probably to allow them not to pair or to pair with somebody else (e.g., guru plus guru).

References

- Costs and benefits of pair programming:
  - Laurie Williams' dissertation at http://www.cs.utah.edu/~lwilliam/Papers/dissertation.pdf details a research done in a university environment regarding pair programming.
  - Alistair Cockburn and Laurie Williams' paper at http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF analyzes the costs and benefits of pair programming.
  - http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF.
  - http://www.agilealliance.org/articles/articles/QuantitativeAssessment.pdf.
- How to pair program effectively:
  - http://www.agilealliance.org/articles/articles/Kindergarten.pdf.
  - http://www.pairprogramming.com/conversantpairing.htm.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

  - http://www.agilealliance.org/articles/articles/PairedProgrammingandPersonalityTraits.pdf .

- Demonstrations:
  - Robert Martin and Robert Koss' article at http://www.objectmentor.com/publications/xpepisode.htm demonstrates how Extreme Programming, including pair programming, works in practice.

- Further references:
  - Laurie Williams' web site about pair programming: http://www.pairprogramming.com/.
  - A website about Agile Development: http://www.agilealliance.org.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Chapter exercises

Problems

1. Find someone to pair with you to develop an "update" method in DBTable that models after its counterpart in DataAccesser without using arrays.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org