Design Patterns

One principle which motivates many design patterns may be stated as "favour composition over inheritance".

a. Explain what the terms inheritance and composition mean, giving a short code example to illustrate each explanation. **4 marks**

b. Give two reasons why you think it might be a good idea to to favour composition over inheritance. Make sure that you justify the reasons you give, using simple code examples where necessary. **4 marks**

c. Give an example of a design pattern which uses composition to extend the capabilities of an object at runtime. **1 marks**

d. Write down the standard summary description for the design pattern you identified in part c above. **1 mark**

e. Draw a simple class diagram which describes the overall structure of the pattern you identified in part c above. **2 marks**

f. Discuss an example which illustrates how the pattern you identified in part c is used in practice. You might draw this example from your experience in the assignments, from examples we've discussed in class or from the Java standard library itself. **4 marks**

Testing

a. Explain one (1) advantage of automated testing processes relative to manual ones. **2 marks**
b. Name two (2) kinds of programmer tests and describe the purpose of each kind of test that you mention. **4 marks**
c. Explain what a mock object is. Your explanation should discuss two (2) common reasons why mock objects might be used in an automated test suite. **4 marks**

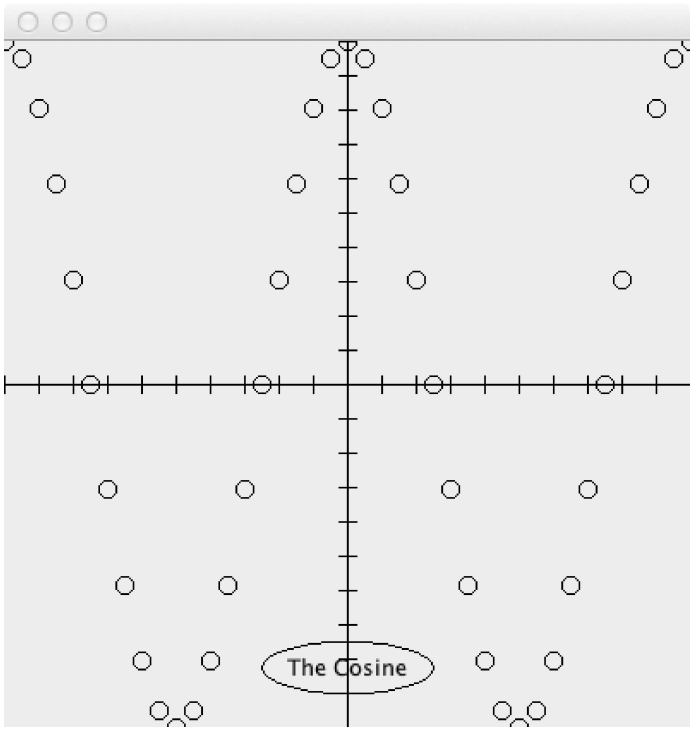
Code Refactoring

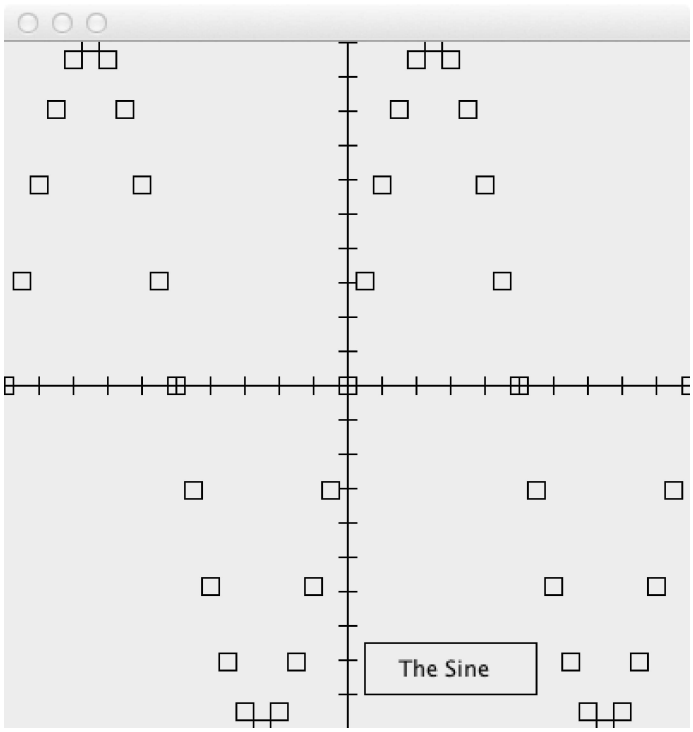
Recently a mathematician friend asked me to write an application which displayed the sine and cosine functions as graphs. As part of that application, I developed two classes which extend the JPanel class to plot one or other of these functions. My rather scrappy code for these classes is displayed on this page and the next:

```
/**
* A simple class which extends a basic panel to display * a plot of the sine function.
*
* @author Dominic Verity
*/
@SuppressWarnings("serial")
public class SinePlotter extends JPanel {
/**
* Default constructor. Initialises the size of this panel. */
SinePlotter() {
setPreferredSize(new Dimension(400, 400));
}
/**
* Paints a pair of axes, the sine function itself and a
* legend onto the panel. *
* @param pGraphics the graphics object upon which
* painting is to take place.
* @see javax.swing.JComponent#paintComponent(java.awt.Graphics) */
@Override
protected void paintComponent(Graphics pGraphics) {
// Draw the x-axis
pGraphics.drawLine(0, 200, 400, 200); for (int i = 0; i <= 400; i += 20) {
pGraphics.drawLine(i, 200 - 5, i, 200 + 5); }
// Draw the y-axis
pGraphics.drawLine(200, 0, 200, 400); for (int j = 0; j <= 400; j += 20) {
pGraphics.drawLine(200 - 5, j, 200 + 5, j); }
// Plot the graph itself.
for
}
// Plot the legend.
pGraphics.drawRect(210, 350, 100, 30);
pGraphics.drawString("The␣Sine", 230, 370); }
}
/**
* A simple class which extends a basic panel to display * a plot of the cosine function.
*
* @author Dominic Verity
*/
(int i = 0; i <= 400; i += 10) {
double pXVal = 2 * Math.PI * (i - 200) / 200; double pYVal = Math.sin(pXVal);
int j = -(int)(pYVal * 200) + 200; pGraphics.drawRect(i - 5, j - 5, 10, 10);
@SuppressWarnings("serial")
public class CosinePlotter extends JPanel {
/**
* Default constructor. Initialises the */
CosinePlotter() {
setPreferredSize(new Dimension(400,
}
size of this panel.
400));
/**
* Paints a pair of axes, the cosine function itself and a
* legend onto the panel. *
* @param pGraphics the graphics object upon which painting
* is to take place.
* @see javax.swing.JComponent#paintComponent(java.awt.Graphics) */
@Override
protected void paintComponent(Graphics pGraphics) {
// Draw the x-axis
pGraphics.drawLine(0, 200, 400, 200); for (int i = 0; i <= 400; i += 20) {
```

```
pGraphics.drawLine(i, 200 - 5, i, 200 + 5); }
// Draw the y-axis
pGraphics.drawLine(200, 0, 200, 400); for (int j = 0; j <= 400; j += 20) {
pGraphics.drawLine(200 - 5, j, 200 + 5, j); }
// Plot the graph itself.
for
}
// Plot the legend.
pGraphics.drawOval(150, 350, 100, 30);
pGraphics.drawString("The Cosine", 165, 370); }
}
```

When objects of these classes are used as the content pane of a main JFrame window, they give rise to the following application displays:





Take a few minutes to understand what this code is doing, and then answer the following questions:

a. Briefly explain what the terms code refactoring and code smell mean. **2 marks**

b. Identify three (3) code smells which occur in the code displayed on pages 11 and 12. For each of the code smells you have written down explain exactly how it is manifested in my code. **3 marks**

c. Refactor the code on pages 11 and 12 to avoid the code smells you identified in part (b) above. In your workbook you should write your refactored code using Java which is as close as you can make it to being correct. **10 marks**

Another

There is another sample exam at

http://ilearn.mq.edu.au/pluginfile.php/2079620/mod_resource/content/2/week12.pdf

(past exam question 2011) The MVarclass provides a way to control access to shared resources in a threaded Java program. In summary form, its declaration is:

public class MVar {

```
// Instance Variables
private T mContent;

// Constructor
public MVar() {
    // .... implementation omitted
}

// Put method
public void put(T pValue) throws InterruptedException {
    // .... implementation omitted
}

// Take method
public T take() throws InterruptedException {
    // .... implementation omitted
}

// .... some more methods.
```

} As you can see, this is a generic class, parameterised by a single type variable, whose objects can be used to store a single value of that parameter type. The interesting feature of this class, from the point of view of using it in threaded code, is the behaviour of its put() and take() methods.

Specifically, each MVar object can be in one of two states, empty, in which case its mContent field contains null, or full, in which case that field contains an actual object (of type T). When it is created each new MVar object starts up in the empty state and after that calls to its put() and take() methods behave as follows:

If our MVar object is empty and we make a call put(vObject) on it then the value vObject will be stored in the mContent field of this MVar and it will switch into the full state. On the other hand if our MVar is already full then the thread that made this call to put() will be put to sleep and will be woken at some point in the future when our MVar is switched back to the empty state - at which point it can complete the task of the last sentence and return.

If our MVar object is full and we make a call take() on it then the value stored in its mContent field will be returned and our MVar will be switched to the empty state (which means that the value of its mContent field will be reset to null). On the other hand if our MVar is empty when this call to take() is made then the thread that made it will be put to sleep and will be woken at some point in the future when our MVar is switched back to the full state - at which point it can complete the task of the last sentence and return.

Of course, it is possible to make a number of calls to take() or put() on the same MVar object from many different threads at the same time - which means that at any time we may have more than one thread waiting to complete these operations. In that case, threads are queued, in separate put and take queues, in the order that they made their calls and each time our MVar makes a transition from empty to full (or back again) only the first thread in the appropriate take or put queue will be awoken.

With this information in mind, answer the following questions:

(part a) Suppose that Java didn' t provide the synchronized keyword and its associated implicit monitor lock mechanism. Under these circumstances, explain how you could use the MVar class to write a class whose methods behaved exactly as if they were synchronized. To simplify your answer, you should illustrate it with an example of a class which uses the implementation strategy you suggest. (8 exam marks)

Remember that it is posible for the code in a synchronized method can use the return statement to make an early exit. It is also possible for exceptions to be raised in the body of such a method, which may also result in early termination. Your code should handle such situations correctly - ensuring that any locking mechanism you have used is correctly unlocked in the even of early exit.

Hint: you might consider using a try....finally... statement for this purpose.

(part b) Use the MVar class to implement a simple producer-consumer simulation in Java. Your simulation should create and start two threads. The first of these should produce the sequence of integers 0,1,2,.... each of which it should transmit to the second thread. The second thread should read (consume) each of these values as it arrives and should print it to the terminal. Both of these threads should sleep for a random period of time (up to 1 second) before it attempts to transmit / read and print a value. You should use an MVar object to ensure that every value that is transmitted by the first thread is successfully received by the second thread. (16 exam marks)

Important: In these questions we are not asking you to provide an implementation of the MVar class. You can assume that such a class is provided for you as part of the standard library.

Consider the following class, whose objects are used to represent flights between European destinations which transit through the hub airport in Frankfurt, Germany:

/** * A class to represent flights between European destinations * which pass through the Frankfurt hub airport. * * @author Dominic Verity * */ *public class Flight { /* Constants */*

```
/*
 * Origin / Destination numbers.
 */
public static final int FRANKFURT = 0;
public static final int LONDON = 1;
public static final int MADRID = 2;
public static final int PARIS = 3;
public static final int STOCKHOLM = 4;

/*
 * Aircraft model numbers.
 */
public static final int AIRBUS_A380 = 0;
public static final int AIRBUS_A340_600 = 1;
public static final int BOEING_747_400 = 2;
public static final int AIRBUS_A321 = 3;

/*
 * Instance variables
 */
/**
 * Flight origin number.
 */
private int mOrigin;
/**
 * Flight destination number.
 */
private int mDestination;
/**
 * Aircraft model number for this flight.
 */
private int mAircraft;
/**
 * Airline flight number.
 */
private int mFlightNumber;

/*
 * Constructors
 */
/**
 * Construct a flight object with specified origin, destination, aircraft
 * and fight number.
 *
 * @param pFlightNumber the flight number, a positive integer.
 * @param pOrigin the origin number for the flight.
 * @param pDestination the destination number for the flight.
 * @param pAircraft the aircraft model number.
 * @throws FlightException thrown if an illegal origin / destination, aircraft model or
 *                         flight number is specified.
 */
public Flight(int pFlightNumber, int pOrigin, int pDestination, int pAircraft) throws FlightException {
    if (pOrigin >= 0 && pOrigin <= 4) {
        mOrigin = pOrigin;
    } else {
        throw new FlightException("Incorrect airport number.");
    }

    if (pDestination >= 0 && pDestination <= 4) {
        mDestination = pDestination;
```

```
    } else {
        throw new FlightException("Incorrect airport number.");
    }

    if (pAircraft >= 0 && pAircraft <= 3) {
        mAircraft = pAircraft;
    } else {
        throw new FlightException("Incorrect aircraft number.");
    }

    if (pFlightNumber >= 0) {
        mFlightNumber = pFlightNumber;
    } else {
        throw new FlightException("Negative flight number");
    }
}

/*
 * Methods
 */

/**
 * Change the origin airport of this flight.
 *
 * @param pOrigin the new origin airport number.
 * @throws FlightException if the specified airport number is illegal.
 */
public void changeOrigin(int pOrigin) throws FlightException {
    if (pOrigin >= 0 && pOrigin <= 4) {
        mOrigin = pOrigin;
    } else {
        throw new FlightException("Incorrect airport number.");
    }
}

/**
 * Change the destination airport of this flight
 *
 * @param pDestination the new destination airport number.
 * @throws FlightException if the specified airport number is illegal.
 */
public void changeDestination(int pDestination) throws FlightException {
    if (pDestination >= 0 && pDestination <= 4) {
        mDestination = pDestination;
    } else {
        throw new FlightException("Incorrect airport number.");
    }
}

/**
 * Change the aircraft used in this flight.
 *
 * @param pAircraft the new aircraft number.
 * @throws FlightException if the specified aircraft number is illegal.
 */
public void changeAircraft(int pAircraft) throws FlightException {
    if (pAircraft >= 0 && pAircraft <= 3) {
        mAircraft = pAircraft;
    } else {
        throw new FlightException("Incorrect aircraft number.");
```

```
    }
}

/**
 * Calculate the cost per passenger for this flight. Assumes that the
 * flight consists of two legs origin -> Frankfurt and Frankfurt -> destination.
 *
 * @return the total cost of this flight to a passenger.
 */
public double calculateCost() {

    // Different aircraft incur different costs per kilometer per passenger.
    double vCostFactor = 0;
    switch (mAircraft) {
    case AIRBUS_A380 : vCostFactor = 0.3; break;
    case AIRBUS_A340_600 : vCostFactor = 0.35; break;
    case BOEING_747_400 : vCostFactor = 0.45; break;
    case AIRBUS_A321 : vCostFactor = 0.4; break;
    }

    // Distance from Frankfurt to origin airport.
    double vOriginDistance = 0;
    switch (mOrigin) {
    case FRANKFURT : vOriginDistance = 0; break;
    case LONDON : vOriginDistance = 609.3; break;
    case MADRID : vOriginDistance = 1420.8; break;
    case PARIS : vOriginDistance = 475.9; break;
    case STOCKHOLM : vOriginDistance = 1195.4; break;
    }

    // Distance from Frankfurt to destination airport.
    double vDestinationDistance = 0;
    switch (mDestination) {
    case FRANKFURT : vDestinationDistance = 0; break;
    case LONDON : vDestinationDistance = 609.3; break;
    case MADRID : vDestinationDistance = 1420.8; break;
    case PARIS : vDestinationDistance = 475.9; break;
    case STOCKHOLM : vDestinationDistance = 1195.4; break;
    }

    // Cost is equal to total distance via Frankfurt airport
    // multiplied by the cost factor.
    return vCostFactor * (vOriginDistance + vDestinationDistance);
}

/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
    String vResult;

    vResult = "Flight number : " + mFlightNumber;

    switch (mOrigin) {
    case FRANKFURT : vResult += ", Origin : Frankfurt (FRA)"; break;
    case LONDON : vResult += ", Origin : London (LHR)"; break;
    case MADRID : vResult += ", Origin : Madrid (MAD)"; break;
    case PARIS : vResult += ", Origin : Paris (CDG)"; break;
    case STOCKHOLM : vResult += ", Origin : Stockholm (ARN)"; break;
```

```
    }

    switch (mDestination) {
    case FRANKFURT : vResult += ", Destination : Frankfurt (FRA)"; break;
    case LONDON : vResult += ", Destination : London (LHR)"; break;
    case MADRID : vResult += ", Destination : Madrid (MAD)"; break;
    case PARIS : vResult += ", Destination : Paris (CDG)"; break;
    case STOCKHOLM : vResult += ", Destination : Stockholm (ARN)"; break;
    }

    switch (mAircraft) {
    case AIRBUS_A380 : vResult += ", Aircraft : Airbus A380."; break;
    case AIRBUS_A340_600 : vResult += ", Aircraft : Airbus A340-400."; break;
    case BOEING_747_400 : vResult += ", Aircraft : Boeing 747-400."; break;
    case AIRBUS_A321 : vResult += ", Aircraft : Airbus A321."; break;
    }

    return vResult;

}
```

} Note: We strongly encourage you to use Eclipse to do this refactoring exercise. The code above is also included in the week 10 samples project on bitbucket.

Now answer the following questions:

(part a) Explain what the terms code smell and refactoring mean. (5 exam marks)

(part b) I wrote this code in somewhat of a hurry (train coding again) so it is badly in need of some refactoring. An obvious code smell in my work is the excessive use of switch statements. Name three (3) other code smells which occur in this code. (4 exam marks)

(part c) Why do you think that these code smells might cause problems in maintaining this code in the future? (5 exam marks)

Hint: You might like to consider what you would need to do to add new airports or aircraft types to this code. What might go wrong when you do that?

(part d) Refactor this code to eliminate all four of the code smells identified in the last part of this question. (20 exam marks)

Hint: in object oriented programming it is usually much more natural to replace switch statements with the use of polymorphism, inheritance and dynamic dispatch. Specifically, we often deploy the strategy pattern to replace code numbers (like the ones used to represent airports and aircraft models above) by strategy classes and to replace each switch statement involving those code numbers with calls to an overridden method of these strategy classes. So we would encourage you to introduce two strategy class hierarchies, whose classes will represent different airports and aircraft respectively, and then to recode the Flight class accordingly.

End of Examination Paper