INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN

Year	**Year 3**
Semester	**Semester 1 Autumn Repeat**
Date of Examination	
Time of Examination	

Prog Code	BN013	Prog Title	Bachelor of Science in Computing	Module Code	COMP H3012
Prog Code	BN104	Prog Title	Bachelor of Science (Honours) in Computing]	Module Code	COMP H3012
Prog Code	BN302	Prog Title	Bachelor of Science in Computing in Information Technology	Module Code	COMP H3012

Module Title	**Object Orientation with Design Patterns**

Internal Examiner(s): *Ms. Orla McMahon*
External Examiner(s): *Mr. John Dunnion, Dr. Richard Studdert*

Instructions to candidates:

1) To ensure that you take the correct examination, please check that the module and programme which you are following is listed in the tables above.
2) Section A: Attempt any five parts.
3) Section B: Answer any 3 Questions.

4) All questions carry equal marks.

DO NOT TURN OVER THIS PAGE UNTIL YOU ARE TOLD TO DO SO

Section A
Attempt any 5 parts of this question (5 marks each)

Question 1

a) Describe the difference between **Class Adapters** and **Object Adapters**.

[5 Marks]

b) Graphical representations of design patterns only capture the end product of the design process.
Why?
List and briefly describe **four** essential elements that can be used to describe a design pattern.

[5 Marks]

c) When using the Singleton pattern, one approach is to use a static variable and to make the constructor private.

Explain why you would do this in order to implement the Singleton pattern.

[5 Marks]

d) A common pattern cited in early literature on programming frameworks is the **Model-View-Controller (MVC)** pattern.

Briefly describe the role of the various participants in the **MVC** pattern.

[5 Marks]

e) Describe the role of Design Patterns in software development.

[5 Marks]

f) Use an intuitive example to explain the intent of the Observer design pattern.

[5 Marks]

g) List the role of each of the following participants of the Builder pattern

- **i)** Director
- **ii)** Builder
- **iii)** Concrete Builder
- **iv)** Product

[5 Marks]

(Total Marks 25)

Section B
Candidates should attempt any 3 of the following questions.

Question 2

a) The **Chain of Responsibility Pattern** does not have to use a linear chain.
What does this statement mean?
What, if any, implications are there if a non-linear chain is used?

[4 Marks]

b) Compare the intent of the **Facade** and **Adapter** design patterns.

[6 Marks]

c) Read this case study and answer the questions that follow:

You have been asked to deploy an application that displays details about a person's email address book.
The address book contains two different user interfaces. One interface displays details about people in general such as their name, email address, phone number and company name.
The second interface displays details about groups of people such as the name of the group, its purpose, a list of its members and their email addresses.

i) Identify the design pattern that you would use for this application.

ii) Explain why you would use this design pattern.

iii) Illustrate your answer with the aid of a UML diagram and include a description of the participants.

iv) Describe four consequences of using this pattern.

[15 Marks]

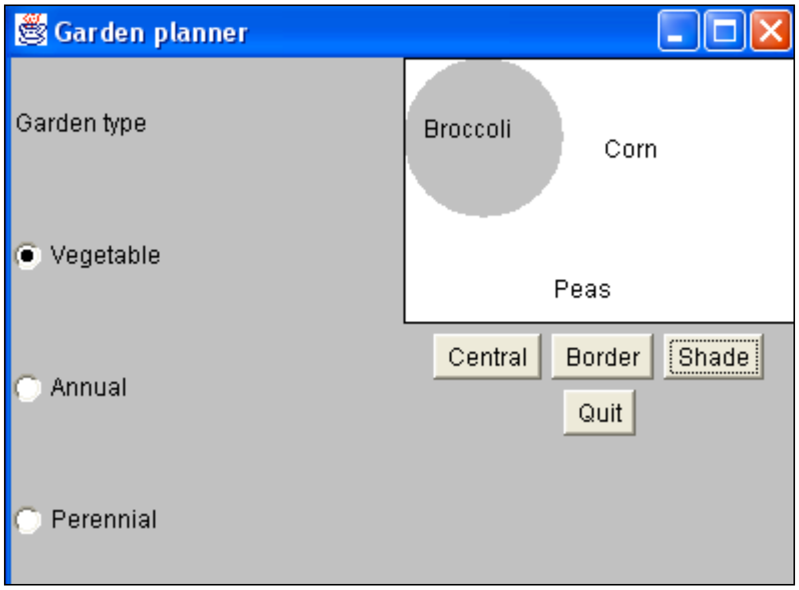
(Total Marks 25)

Question 3

Read this case study and answer the questions that follow:

Assume that you are working as a Java Software Designer. You have been requested to update the Garden Planner software system contained in **code listing 1**. The current system allows a gardener to select a particular garden and a garden area. The plant for that particular area will then be displayed. For example the gardener could select the vegetable garden and then display the plant that is suitable to a shade area. Currently the GUI operates so that the gardener simply selects the garden type, clicks on the appropriate area button (ie central, border or shade) and a suitable plant for the given garden and area appears on the screen.

The following diagram shows the system in operation.



You have been asked to modify the system so that additional gardens can be incorporated into the Garden Planner. The new types of gardens are a Bonsai garden and a Rose garden. In addition to this the garden designer would also like to see the height and watering frequency of each plant that is displayed.

Question 3 (Contd.)

a) Draw an accurate UML diagram that illustrates the structure of the **Abstract Factory** pattern when applied to the updated Garden Planner System.

[5 Marks]

b) Taking the above requirements into account, write new classes for the Plant, BonsaiGarden and RoseGarden participants. Assume that the other garden participants have been already modified.

[8 Marks]

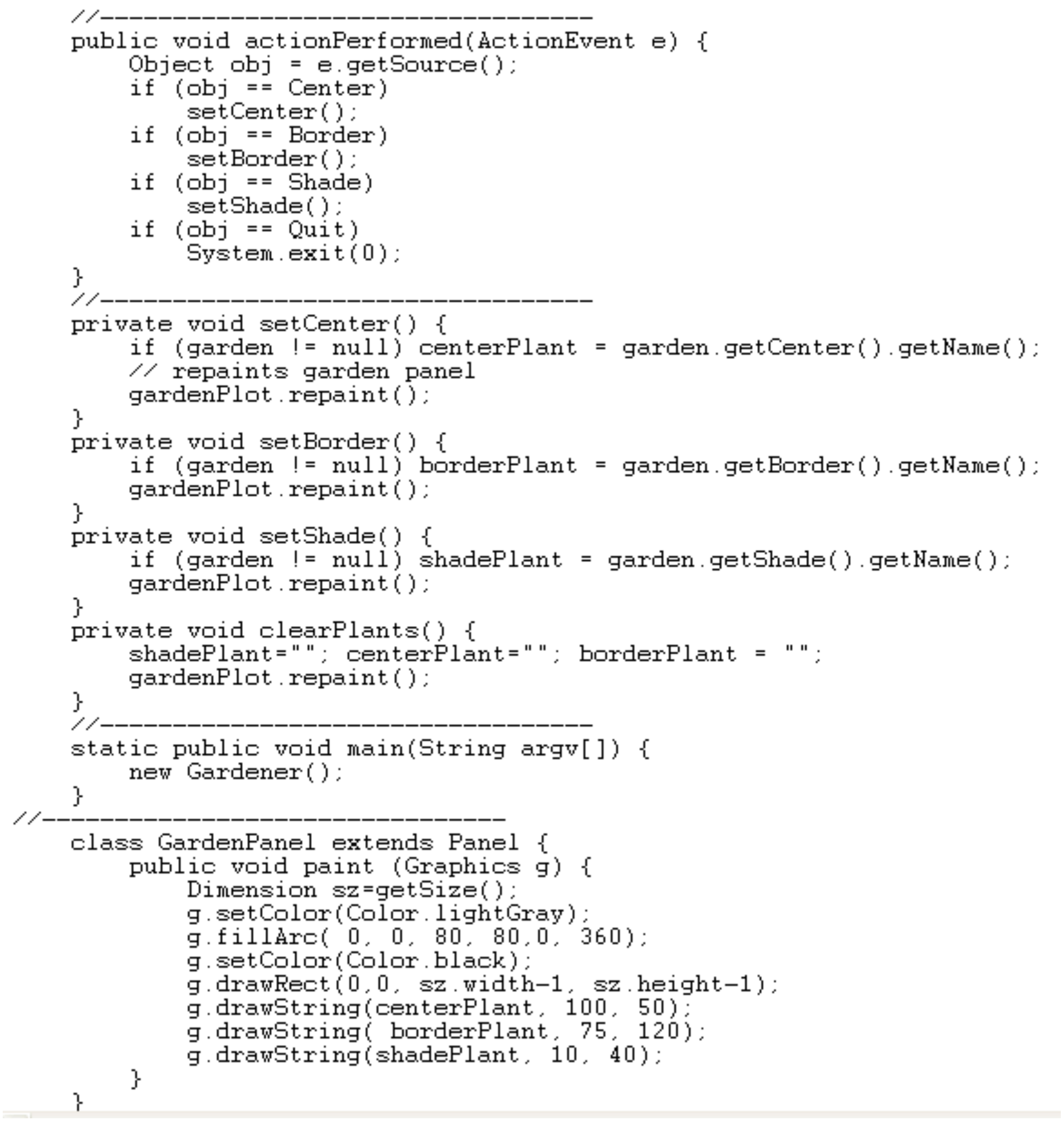
c) Describe in a step-by-step fashion, how you would edit the current Garden Planner (GUI) (Gardener.java) so that a Rose or Bonsai garden can be selected and in addition to the plant name, the height and watering frequency can also be displayed.

[12 Marks]

(Total Marks 25)

Code Listing 1
Gardener.java

```
import java.awt.*;
import java.awt.event.*;

//illustrates use of Abstract Factory pattern
public class Gardener extends Frame
implements ActionListener {
    private Checkbox Veggie, Annual, Peren;
    private Button Center, Border, Shade, Quit;
    private Garden garden = null;
    private GardenPanel gardenPlot;
    private String borderPlant = "", centerPlant = "", shadePlant = "'

    public Gardener() {
        super("Garden planner");
        setGUI();
    }
    //----------------------------------
    private void setGUI() {
        setBackground(Color.lightGray);
        setLayout(new GridLayout(1,2)); // 1 row 2 columns
        Panel left = new Panel();
        add(left);
        Panel right= new Panel();
        add(right);

        //create label and 3 radio buttons on left side

        left.setLayout(new GridLayout(4, 1)); // 4 rows and 1 column
        left.add(new Label("Garden type"));
        CheckboxGroup grp= new CheckboxGroup();
        Veggie = new Checkbox("Vegetable", grp, false);
        Annual = new Checkbox("Annual", grp, false);
        Peren = new Checkbox("Perennial", grp, false);

        left.add(Veggie);
        left.add(Annual);
        left.add(Peren);

        Veggie.addItemListener(new VeggieListener());
        Peren.addItemListener(new PerenListener());
        Annual.addItemListener(new AnnualListener());



        //now create right side
        right.setLayout(new GridLayout(2,1)); // 2 rows 1 column
        gardenPlot = new GardenPanel(); // defined below
        gardenPlot.setBackground(Color.white);
        Panel botRight = new Panel();

        right.add(gardenPlot);
        right.add(botRight);
        Center = new Button("Central");
        Border =  new Button("Border");
        Shade = new Button("Shade");
        Quit = new Button("Quit");
        // add buttons to panel
        botRight.add(Center);
        Center.addActionListener(this);
        botRight.add(Border);
        Border.addActionListener(this);
        botRight.add(Shade);
        Shade.addActionListener(this);
        botRight.add(Quit);
        Quit.addActionListener(this);
        setBounds(200,200, 400,300);
        setVisible(true);

    }
```



```
    //----------------------------------
    public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        if (obj == Center)
            setCenter();
        if (obj == Border)
            setBorder();
        if (obj == Shade)
            setShade();
        if (obj == Quit)
            System.exit(0);
    }
    //----------------------------------
    private void setCenter() {
        if (garden != null) centerPlant = garden.getCenter().getName();
        // repaints garden panel
        gardenPlot.repaint();
    }
    private void setBorder() {
        if (garden != null) borderPlant = garden.getBorder().getName();
        gardenPlot.repaint();
    }
    private void setShade() {
        if (garden != null) shadePlant = garden.getShade().getName();
        gardenPlot.repaint();
    }
    private void clearPlants() {
        shadePlant=""; centerPlant=""; borderPlant = "";
        gardenPlot.repaint();
    }
    //----------------------------------
    static public void main(String argv[]) {
        new Gardener();
    }
//--------------------------------
    class GardenPanel extends Panel {
        public void paint (Graphics g) {
            Dimension sz=getSize();
            g.setColor(Color.lightGray);
            g.fillArc( 0, 0, 80, 80,0, 360);
            g.setColor(Color.black);
            g.drawRect(0,0, sz.width-1, sz.height-1);
            g.drawString(centerPlant, 100, 50);
            g.drawString( borderPlant, 75, 120);
            g.drawString(shadePlant, 10, 40);
        }
    }
    //------------------------------
    class VeggieListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            // creating an instance of garden as VeggieGarden
            garden = new VeggieGarden();
            clearPlants();
        }
    }
    //------------------------------
    class PerenListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            // creating an instance of garden as PerennialGarden
            garden = new PerennialGarden();
            clearPlants();
        }
    }
    //------------------------------
    class AnnualListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            garden = new AnnualGarden();
            clearPlants();
        }
    }


}     //end of Gardener class
```

Garden.java

```
public abstract class Garden {
    public abstract Plant getShade();
    public abstract Plant getCenter();
    public abstract Plant getBorder();
}
```

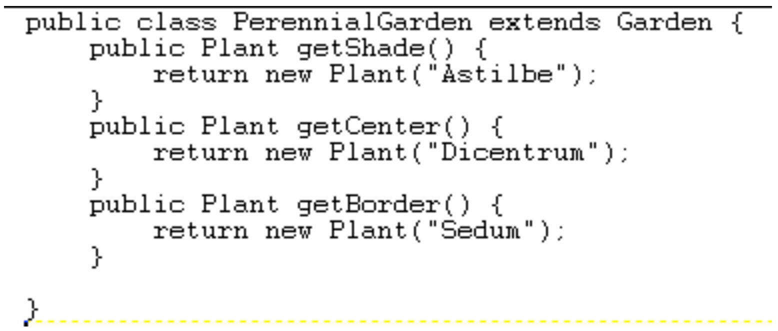
Plant.java

```
public class Plant {
    private String name;
    public Plant(String pname) {
        name = pname;     //save name
    }
    public String getName() {
        return name;
    }
}
```

AnnualGarden.java

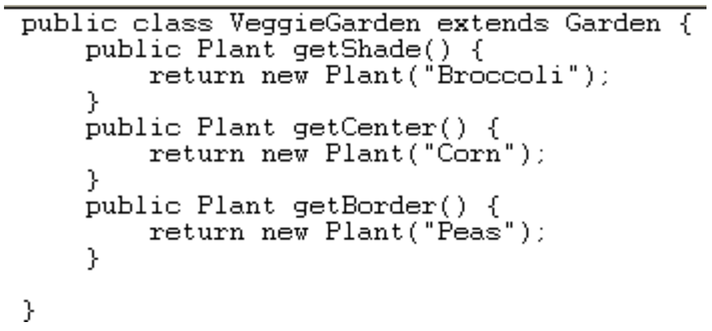
```
public class AnnualGarden extends Garden {
    public Plant getShade() {
        return new Plant("Coleus");
    }
    public Plant getCenter() {
        return new Plant("Marigold");
    }
    public Plant getBorder() {
        return new Plant("Alyssum");
    }

}
```

PerennialGarden.java



```
public class PerennialGarden extends Garden {
    public Plant getShade() {
        return new Plant("Astilbe");
    }
    public Plant getCenter() {
        return new Plant("Dicentrum");
    }
    public Plant getBorder() {
        return new Plant("Sedum");
    }

}
```

VeggieGarden.java



```
public class VeggieGarden extends Garden {
    public Plant getShade() {
        return new Plant("Broccoli");
    }
    public Plant getCenter() {
        return new Plant("Corn");
    }
    public Plant getBorder() {
        return new Plant("Peas");
    }

}
```

Question 4

a) Using some simple UML diagrams together with real-world examples, describe the difference between the **Factory** pattern and the **Factory Method** pattern.

[8 Marks]

b) Explain how the **Proxy** pattern works.
Illustrate your answer with the aid of three examples.

[6 Marks]
]

c) The program given in **code listing 2 (on next page)** creates a simple user interface that allows the user to select menu items, File|Open and File|Exit,. When the user clicks on a button labelled 'Blue', the background colour of the window turns to blue.

As long as there are only a few menu items and buttons this approach works fine, but when there are several menu items and buttons the *actionPerformed* code can get pretty unwieldy.

Using a simple **Command Pattern** re-write the appropriate sections of this program so that the conditional block in the *actionPerformed* code is removed.

[11 Marks]

(Total Marks 25)

Code Listing 2

```
public class SimpleApp extends Frame implements ActionListener
{
  Menu mnuFile;
  MenuItem mnuOpen, mnuExit;
  Button btnBlue;
  Panel p;

  public SimpleApp()
  {
    super("Simple App");

    //Create a new menu bar for the frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // Create the menu items and add them to the bar
    mnuFile = new Menu("File", true);
    mbar.add(mnuFile);
    mnuOpen = new MenuItem("Open...");
    mnuFile.add(mnuOpen);
    mnuExit = new MenuItem("Exit");
    mnuFile.add(mnuExit);

    // Add an actionlistener for each menu item
    // actions will be handled by this class
    mnuOpen.addActionListener(this);
    mnuExit.addActionListener(this);

    // Create a button for the frame
    btnBlue = new Button("Blue");

    // Create a panel and add the button
    p = new Panel();
    add(p);
    p.add(btnBlue);

    // Add an actionlistener for the button
    // actions will be handled by this class
    btnBlue.addActionListener(this);

    setBounds(100,100,200,100);
    setVisible(true);
  }


  // Handle actions from the menu items and button
  public void actionPerformed(ActionEvent e)
  {
    // Determine the source of the action and
    // carry out the appropriate action
    Object obj = e.getSource();
    if(obj == mnuOpen)
      fileOpen();
    if (obj == mnuExit)
      exitClicked();
    if (obj == btnBlue)
      redClicked();
  }

  // Called from actionPerformed when exit selected
  private void exitClicked()
  {
    System.exit(0);
  }
  // Called from actionPerformed when Open selected
  private void fileOpen()
  {
    FileDialog fDlg = new FileDialog(this, "Open a
                    file",FileDialog.LOAD);
    fDlg.show();
  }
  // Called from actionPerformed when button clicked
  private void redClicked()
  {
    p.setBackground(Color.blue);
  }

  static public void main(String argv[])
  {
    new SimpleApp();
  }
}
```

(Total Marks 25)

Question 5

a) Use an intuitive example to explain the intent of the Composite design pattern.

[6 Marks]

b) With the aid of a UML diagram, describe the components of the Composite design pattern

[6 Marks]

a) Examine the classes in **code listing 3 (on next page)**, then answer the following questions:

i) Create a new Decorator class called CrazyDecorator.
The CrazyDecorator class must change a JComponent's background colour to red when the mouse enters the JComponent.
The JComponent's background colour must be reset to it's original colour when the mouse exits the JComponent.

[9 Marks]

ii) Create a simple test program call DecoratorTester which will decorate a JButton with the SlashDecorator **AND** the CrazyDecorator.

[4 Marks

(Total Marks 25)

Code Listing 3
Decorater.java

```
public class Decorator extends JComponent
{
   public Decorator(JComponent c)
   {
      setLayout(new BorderLayout());
      add("Center", c);
   }
}
```

SlashDecorator.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.text.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class SlashDecorator extends Decorator {
   int x1, y1, w1, h1;

   public SlashDecorator(JComponent c) {
      super(c);
   }
   public void setBounds(int x, int y, int w, int h) {
      x1 = x; y1= y;
      w1 = w; h1 = h;
      super.setBounds(x, y, w, h);
   }
   public void paint(Graphics g) {
      super.paint(g);
      g.setColor(Color.red);
      g.drawLine(0, 0, w1, h1);
   }
}
```