5.1

a) How does a software engineer use design patterns?

A software engineer uses patterns to help solve a problem they have encountered. A pattern gives the engineer a solution to a problem that has already been solved that can possibly be applied to the same situation. It saves time and prevents the programmer from using a bad design and having to refactor later on.

b) When do you apply the Strategy pattern ?

We use the strategy pattern when we need an object– the context to behave differently but in similar circumstances. For example, the layout manager, depending on the size of each container the layout manager will arrange the components in various ways.

c) The TitledBorder class can give a title to a border. Consider the code panel.setBorder(new TitledBorder(new BevelBorder(BevelBorder.RAISED), "Confirm Selection")); What design pattern(s) are at work?

The decorator pattern is at work here. The decorator pattern acts like a wrapper that adds functionality without changing the original class. In this case the original panel class is unchanged, but it gets extended with the new title property on it. When the class needs to be used again it will not contain the title property.

5.2a

Diagram

Description automatically generatedDiagram

Description automatically generated

5.2.b

The decorator pattern is at work in this application. Although there is some composition at work with taking the lecture video and placing it in the captioned video, the main intent is not to stack these but rather to decorate the video with captions without changing the original lecture video class. Instead, the captioned Video class extends the features of the video clip without changing the original video.

5.2.c

MainFrame.java

import javax.swing.\*;  
  
*/\*\*  
 \* Sets up initial frame that contains panel  
 \*/*public class MainFrame {  
 */\*\*  
 \* driver code to program  
 \** ***@param*** *args  
 \*/* public static void main(String[] args) {  
 JFrame frame = new JFrame("Bar Graph by JaroldSabillon"); //initializing frame  
 frame.setSize(500, 250); //setting dimensions of frame  
  
 BarChart chart = new BarChart(); // initializing panel  
 frame.add(chart); //adds bar chart to JFrame  
  
 frame.setVisible(true);  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*); //Stops program on exit  
 }  
}

BarChart.java

import java.awt.\*;  
import java.awt.event.KeyEvent;  
import java.awt.event.KeyListener;  
import javax.swing.\*;  
//class definition  
  
*/\*\*  
 \* the main panel for the chart  
 \*/*public class BarChart extends JPanel implements KeyListener {  
  
  
 */\*\*  
 \* creates the panel that contains the charts and text fields  
 \*/* public BarChart() {  
 panel = new JPanel();  
 field[0] = new JTextField("0");  
 field[1] = new JTextField("0"); //initializes text field  
 field[2] = new JTextField("0");  
  
 setLayout(null);  
  
 field[0].setBounds(420,50,50,30); //boundaries of text field  
 field[1].setBounds(420,100,50,30);  
 field[2].setBounds(420,150,50,30);  
  
 add(field[0]); //adds to panel  
 add(field[1]);  
 add(field[2]);  
  
 field[0].addKeyListener(this); //updates when value is changed  
 field[1].addKeyListener(this);  
 field[2].addKeyListener(this);  
 }  
  
 */\*\*  
 \*  
 \** ***@param*** *g the <code>Graphics</code> object to protect  
 \*/* public void paintComponent(Graphics g) {  
 super.paintComponent(g);  
  
 g.drawString("Range is from[0-100]", 25, 25);  
 g.setColor(Color.*RED*);  
 g.fillRect(x[0], y[0], z[0], h[0]);  
 g.setColor(Color.*YELLOW*); //colors the bars  
 g.fillRect(x[1], y[1], z[1], h[1]);  
 g.setColor(Color.*BLUE*);  
 g.fillRect(x[2], y[2], z[2], h[2]);  
 }  
 @Override  
 public void keyTyped(KeyEvent e) {  
 }  
  
 @Override  
 public void keyPressed(KeyEvent e) {  
 }  
  
 @Override  
 public void keyReleased(KeyEvent e) {  
 int vals[] = new int[3];  
 try{  
 vals[0]=Integer.*parseInt*(field[0].getText());  
 vals[1]=Integer.*parseInt*(field[1].getText()); //parses text field to integers  
 vals[2]=Integer.*parseInt*(field[2].getText());  
 }  
 catch(NumberFormatException nfe){  
 vals[0]=0;  
 vals[1]=0; //if fails to parse, sets value to 0  
 vals[2]=0;  
 }  
 z[0]=vals[0]\*2;  
 z[1]=vals[1]\*2; //doubles values to use smaller numbers, halves precision  
 z[2]=vals[2]\*2;  
 repaint();  
 }  
 private final int x[] = {10,10,10}; //setting up values  
 private final int y[] = {50,90,130};  
 private int z[] = {100,100,100};  
 private final int h[] = {40,40,40};  
 private JPanel panel;  
 private JTextField[] field = new JTextField[3];  
}

6.1

a) You should use an abstract class when you do not know exactly what the methods are required to do just yet. The method body can be implemented later by their children but also ensures that the children must have those methods. For example, we may create a class called animal with a method such as makeSound(). It is useful to make an abstract class so that other classes can then derive from it and implement the makeSound() method in their own ways.

b) Only an interface should be used over an abstract class when you want any other class that will implement the interface to HAVE to use its methods. Like a contract, where it is required to incorporate the method when using the interface.

c) This uses the composite pattern. It creates a class which will then contain a group of objects which can then be modified.

6.2

a) The template method works by works by having the parent class as a skeleton of the methods and leaving the details of the methods to its children. In the case of Rectangle2d/.float/.double because they all contain similar code, methods, variables. And when needed the children will over ride the parent as it is the superclass.

b) CompoundShape.add is protected so as not to let other shapes draw over it. Preventing ‘graffiti’

CompoundShape.path is private so that its features are not accessible by subclass methods and by methods in the same package, which would occur if It was protected instead.

6.3.