Adam Corbin COP 5330 - 003

3.1

a

Encapsulation is important to hide away unnecessary implementation details to the end users. This will allow the user to focus on their end goal without having to worry about all the intermediate details
on how to get to the goal.

b

As part of a contract there could be a precondition that assumes the user understands. If this contact/precondition
is broken then this is where an exception can be valid to be thrown.

Example

There is a class foo which keeps track of a list of items. There can be a method of foo called remove. In this method we
have a precondition/contract that states that you can not remove from an empty list. At this point if there is an empty
list and remove is called, then it would be ok to throw an exception that list is empty and nothing can be removed.

c

Side Effects are basically methods that modify the state of an object or an input parameter. For example the next method in a list/iterator
will change the internal pointer to the current object. Side Effects should be avoid because it really can confuse users
as its not a typical way do implementing code. If Side Effects are used then make sure that its clearly documented.

d

And example of a bad interface design would be to have a Person class that would implement methods such has printAPage
which should belong to a Printer class. This would be an example of Convenience but it conflicts with
Cohesion since printing of pages is not something a Persons class should do.

3.2

Complex.java

```
class Complex{
    private double real = 0;
    private double imaginary = 0;

    /**
     * @param real setting the this.real
     * @param imaginary setting the this.imaginary
     */
    public Complex(double real, double imaginary){
        this.real = real;
        this.imaginary = imaginary;
    }

    /**
     * Constructor that sets the real and leaves imaginary to zero.
     * @param real used to set this.real
     */
    public Complex(double real){
        this.real = real;
    }

    /**
     * @precondition: Expected to have initilized real and imaginary
     * @return formatted string with the real and imaginary parts of Complex.
Considers handling the negative imaginary
     * to ensure correct formatting of 1 - 4i where 4 is negative
     */
    @Override
    public String toString() {
        if(imaginary >= 0){
            return real + " + " + imaginary + "i";
        }else{
            return real + " - " + (imaginary * -1) + "i";
        }
    }

    /**
     * @precondition: None
     * @postcondition: None
     * @return this private real
     */
    public double r(){
        return this.real;
    }

    /**
     * @Precondition: None
     * @Postcondition: None
     * @return the private imaginary
     */
    public double i(){
        return this.imaginary;
    }

    /**
     * Method to add this complex object with a passed in complex object and
output a newly created complex object
```

```
     * @Precondition: None
     * @Postcondition: result of adding 2 complex object
     * @param complex object to be added to this.
     * @return result of adding 2 complex object
     */
    public Complex add(Complex complex){
        return new Complex(this.r() + complex.r(), this.i() + complex.i());
    }

    /**
     * Method to subtract this complex object with a passed in complex object
and output a newly created complex object
     * @Precondition: None
     * @Postcondition: result of subtracting 2 complex object
     * @param complex object to be subtracted to this.
     * @return result of subtracting 2 complex object
     */
    public Complex sub(Complex complex){
        return new Complex(this.r() - complex.r(), this.i() - complex.i());
    }

    /**
     * Method to multiply this complex object with a passed in complex object
and output a newly created complex object
     * @Precondition: None
     * @Postcondition: result of multiplying 2 complex object
     * @param complex object to be multiplied to this.
     * @return result of multiplying 2 complex object
     */
    public Complex mult(Complex complex){
        double newReal = this.r() * complex.r() - this.i() * complex.i();
        double newImaginary = this.i() * complex.r() + this.r() * complex.i();
        return new Complex(newReal, newImaginary);
    }

    /**
     * Method to divide this complex object with a passed in complex object and
output a newly created complex object
     * Source on the formula for dividing:
     *
https://www.khanacademy.org/math/precalculus/x9e81a4f98389efdf:complex/x9e81a4f9
8389efdf:complex-div/v/dividing-complex-numbers
     * @Precondition: The passed in complex should have a real number and
imaginary number both no equal to zero.
     * @Postcondition: result of dividing 2 complex object
     * @param complex object to be divided to this.
     * @return result of dividing 2 complex object
     * @throws : Raise ArithmeticException when passed in complex is 0 + 0i
     */
    public Complex div(Complex complex)throws ArithmeticException{
        if(complex.i() != 0.0 && complex.r() != 0.0) {
            // Multiply the conjugate to get a simpler version
            Complex numerator = this.mult(complex.conj());
            Complex denominator = complex.mult(complex.conj());
            //When you multiple a imaginary number with its conjegate, the
imaginary part goes away leaving us with just the
            // real number. Then we can use that real number ot divide the
numerator.
```

```
            return new Complex(numerator.r() / denominator.r(), numerator.i() /
denominator.r());
        }else{
            throw new ArithmeticException("ERROR: Cant divide by zero. Passed in
Complex number is 0 + 0i.");
        }
    }

    /**
     * Checks to see if the 2 Complex numbers have equivalent real and imaginary
parts.
     * @param complex number to compare with this.
     * @Precondition: None
     * @Postcondition: boolean value of the evaluation between two Complex
objects
     * @return true == they are the same, false == they are different
     */
    public boolean equals(Complex complex){
        return this.r() == complex.r() && this.i() == complex.i();
    }

    /**
     * This method creates a new Complex object with the conjugate.
     * To compute the conjugate by just multiple imaginary by -1
     * @Precondition: None
     * @Postcondition: new complex object created
     * @return new Complex object with conjugate
     */
    public Complex conj(){
        return new Complex(this.real, this.imaginary * -1.0);
    }

    public static void main(String[] args) {

        Complex a = new Complex(6,3);
        Complex b = new Complex(7,-5);

        System.out.println("A:\t\t" + a.toString());
        System.out.println("B:\t\t" + b.toString());
        System.out.println("Add:\t" + a.add(b));
        System.out.println("Sub:\t" + a.sub(b));
        System.out.println("Mult:\t" + a.mult(b));
        System.out.println("Div a/b:\t" + a.div(b));
        System.out.println("Div b/a:\t" + b.div(a));
        System.out.println("A Conj:\t" + a.conj().toString());
        System.out.println("B Conj:\t" + b.conj().toString());

    }
}
```

TestComplex.java

```
import org.junit.Test;

import static org.junit.Assert.assertTrue;
```

```
public class TestComplex {

    /**
     * Creating 2 complex numbers with the same real and imaginary to test the
equals
     */
    @Test
    public void testEquals() {
        System.out.println("Run test equals");
        Complex a = new Complex(1,2);
        Complex b = new Complex(1,2);
        assertTrue(a.equals(b));
    }

    /**
     * Computing internally what an Add would do, and then using 2 complex
numbers to do the same to compare.
     */
    @Test
    public void testAdd() {
        System.out.println("run test add");
        double r1 = 1, i1 = 2, r2 = -3, i2 = 4;
        double rResult = r1 + r2, iResult = i1 + i2;
        Complex x = new Complex(r1,i1);
        Complex y = new Complex(r2,i2);
        Complex w = x.add(y);
        Complex z = new Complex(rResult,iResult);
        // test condition using the Complex equals() method:
        assertTrue(z.equals(w));
    }

    /**
     * Computing internally what an subtract would do, and then using 2 complex
numbers to do the same to compare.
     */
    @Test
    public void testSub() {
        System.out.println("run test sub");
        double r1 = 1, i1 = 2, r2 = -3, i2 = 4;
        double rResult = r1 - r2, iResult = i1 - i2;
        Complex x = new Complex(r1,i1);
        Complex y = new Complex(r2,i2);
        Complex w = x.sub(y);
        Complex z = new Complex(rResult,iResult);
        // test condition using the Complex equals() method:
        assertTrue(z.equals(w));
    }

    /**
     * Computing internally what an multiply would do, and then using 2 complex
numbers to do the same to compare.
     */
    @Test
    public void testMult() {
        System.out.println("run test mult");
        double r1 = 1, i1 = 2, r2 = -3, i2 = 4;
        double rResult = r1*r2 - i1*i2, iResult = i1*r2 + r1 * i2;
        Complex x = new Complex(r1,i1);
```

```
        Complex y = new Complex(r2,i2);
        Complex w = x.mult(y);
        Complex z = new Complex(rResult,iResult);
        // test condition using the Complex equals() method:
        assertTrue(z.equals(w));
    }

    /**
     * Computing internally what an divide would do, and then using 2 complex
numbers to do the same to compare.
     */
    @Test
    public void testDiv() {
        System.out.println("run test div");
        double r1 = 1, i1 = 2, r2 = -3, i2 = 4;
        double divisor = r2*r2 + i2*i2;
        double rResult = (r1*r2 + i1*i2)/divisor, iResult = (i1*r2 - r1*i2) /
divisor;
        Complex x = new Complex(r1,i1);
        Complex y = new Complex(r2,i2);
        Complex w = x.div(y);
        Complex z = new Complex(rResult,iResult);
        // test condition using the Complex equals() method:
        assertTrue(z.equals(w));
    }
}
```

4.1

Student.java

```
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Comparator;
import java.util.Date;

class Student {
    private String name = "";
    private Date whenEnrolled;

    /*@invariant name will not change after object creation*/
    /*@invariant whenEnrolled date will not change after object creation*/

    /**
     * @param name         - String name of the student
     * @param whenEnrolled - Date object of when the student was enrolled
     */
    public Student(String name, Date whenEnrolled) {
        this.name = name;
        this.whenEnrolled = whenEnrolled;
    }

    /**
     * @return retrieves the name from the student
     * @Precondition: None
```

```
     * @Postcondition: Returns back the students name
     */
    public String getName() {
        return name;
    }

    /**
     * @return retrieves the date when enrolled from the student
     * @Precondition: None
     * @Postcondition: the date object is cloned so the caller cant not
manipulate the internal date
     */
    public Date getWhenEnrolled() {
        //Clone the whenEnrolled object so the date cant be changed
        return (Date) whenEnrolled.clone();
    }

    /**
     * This comparator is used to sort students by their name
     */
    public static Comparator<Student> getCompByName() {
        return new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
                return o1.getName().compareTo(o2.getName());
            }
        };
    }

    /**
     * This comparator is used to sort students by their enroll date
     */
    public static Comparator<Student> getCompByDate() {
        return new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
                return o1.getWhenEnrolled().compareTo(o2.getWhenEnrolled());
            }
        };
    }

    public static void main(String[] args) {
        Student s1 = new Student("Adam", new Date(2100, Calendar.JANUARY,1));
        Student s2 = new Student("Nicola", new Date(2010,Calendar.JANUARY,1));
        Student s3 = new Student("Soleil", new Date(2010,Calendar.FEBRUARY,1));
        Student s4 = new Student("Weston", new Date(2020,Calendar.JANUARY,1));
        ArrayList<Student> list = new ArrayList<>();
        list.add(s4);
        list.add(s1);
        list.add(s2);
        list.add(s3);
        list.sort(Student.getCompByName());
        System.out.println("Sorted list by name");
        for(Student s : list){
            System.out.println(s.getName() + " " +
s.getWhenEnrolled().toString() + " year:" + s.getWhenEnrolled().getYear());
        }
```

```
        System.out.println("");
        list.sort(Student.getCompByDate());
        System.out.println("Sorted list by Date");
        for(Student s : list){
            System.out.println(s.getName() + " " +
s.getWhenEnrolled().toString() + " year:" + s.getWhenEnrolled().getYear());
        }
    }
}
```

4.2

q4_2.java

```
/*
 * This Java source file was generated by the Gradle 'init' task.
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class q4_2 {
    /**
     * In the main method a SimpleIcon which will be used in program to control
its color. Then there will be 3 buttons
     * dynamically generated which will have Color names as their text. Each
button will have an Action Listener which
     * when click will change the SimpleIcon to the color corresponding to the
button text.
     * @param args None used
     */
    public static void main(String[] args) {

        final SimpleIcon simpleIcon = new SimpleIcon(150, 150);
        final JFrame frame = new JFrame();

        //With JAva 11 the following line did not work. So I broke it up using
colorStrings and Colors
        //Color[] colors = new String[]{"RED", "GREEN", "BLUE"};
        final String[] colorStrings = {"GREEN", "BLUE", "RED"};
        final Color[] colors = {Color.GREEN, Color.BLUE, Color.RED};
        JButton[] btn = new JButton[3];

        frame.setLayout(new FlowLayout());
        for (int i = 0; i < colorStrings.length; i++) {
            btn[i] = createButton(i, colorStrings, colors, simpleIcon, frame);
            frame.add(btn[i]);
        }

        JLabel label = new JLabel(simpleIcon);
        frame.add(label);
        frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        frame.setVisible(true);
        frame.setSize(200, 300);

    }
    /**
     * The function will generate a button that will have an Action listener
which when clicked
     * will change the color of the simple Icon.
     * @param i Index used to look up the color String an color
     * @param colorStrings List of string color names
     * @param colors List of colors used to update the simpleIcon
     * @param simpleIcon Controlled icon based on the action listener.
     * @param frame JFrame where the simpleIcon lives and buttons live
     * @return the button generated which will have an action Listener to change
the icon color
     */
    private static JButton createButton(int i, String[] colorStrings, Color[]
colors, SimpleIcon simpleIcon, JFrame frame){
        JButton helloButton = new JButton(colorStrings[i]);
        final int index = i;
        helloButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                simpleIcon.setColor(colors[index]);
                frame.repaint();
            }
        });
        return helloButton;
    }
}
```

SimpleIcon.java

```
import javax.swing.*;
import java.awt.*;

public class SimpleIcon implements Icon {
    private int width = 0;
    private int height = 0;
    private Color color = Color.RED;

    /**
     * Main constructor which height its heigth and width
     * @param width Size in pixels
     * @param height Size in pixels
     */
    SimpleIcon(int width, int height){
        this.width = width;
        this.height = height;
    }

    /**
     * @param color Updates the color
     */
    public void setColor(Color color) {
        this.color = color;
```

```
    }

    /**
     * This method will generate the circle and fill it will the provided color
     * @param c
     * @param g
     * @param x position x
     * @param y position y
     * @preconditions: Expected that the color is already initilized
     * @postconditions: the object will update and redraw the shape based off
the size and color predefined
     */
    @Override
    public void paintIcon(Component c, Graphics g, int x, int y) {
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(color);
        g2.fillOval(x,y,this.width, this.height);

    }

    /**
     * @return returns the Width which was based on the constructor
     */
    @Override
    public int getIconWidth() {
        return this.width;
    }

    /**
     * @return returns the Height which was based on the constructor
     */
    @Override
    public int getIconHeight() {
        return this.height;
    }
}
```

4.3

ShapeIcon.java

```
import javax.swing.*;
import java.awt.*;

/**
 An icon that contains multiple moveable shapes.
 */
public class ShapeIcon implements Icon
{
    public ShapeIcon(MoveableShape[] shapes,
                     int width, int height)
    {
        this.shapes = shapes;
        this.width = width;
```

```
        this.height = height;
    }

    public int getIconWidth()
    {
        return width;
    }

    public int getIconHeight()
    {
        return height;
    }

    /**
     * This method will handel drawing all of the movableShapes on the screen
     */
    public void paintIcon(Component c, Graphics g, int x, int y)
    {
        Graphics2D g2 = (Graphics2D) g;
        for (MoveableShape shape : shapes) {
            shape.draw(g2);
        }
    }

    private int width;
    private int height;
    private MoveableShape[] shapes;
}
```

AnimationTester.java

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * This program implements an animation that moves 5 different cars staggered
where
 * the first car is the slowest, the folling cars are 2x faster than the
previous
 */
public class AnimationTester {
    public static void main(String[] args) {
        JFrame frame = new JFrame();

        //Create 5 movable shapes and start them staggered apart
        // on the y coordinate
        final MoveableShape[] shapes = new MoveableShape[5];
        for (int i = 0; i < 5; i++) {
            shapes[i] = new CarShape(0, 0, CAR_WIDTH);
            shapes[i].translate(0, i * 50);
        }

        ShapeIcon icon = new ShapeIcon((MoveableShape[]) shapes,
                ICON_WIDTH, ICON_HEIGHT);
```

```
        final JLabel label = new JLabel(icon);
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);

        final int DELAY = 100;
        // Milliseconds between timer ticks
        Timer t = new Timer(DELAY, new
                ActionListener() {
                    public void actionPerformed(ActionEvent event) {
                        //Ensure that each car following is 2x as the previous
so using i^2
                        for (int i = 0; i < shapes.length; i++) {
                            int speed = i+1;
                            shapes[i].translate(speed * speed, 0);
                        }
                        label.repaint();
                    }
                });
        t.start();

        frame.setSize(750, 400);

    }

    private static final int ICON_WIDTH = 50;
    private static final int ICON_HEIGHT = 50;
    private static final int CAR_WIDTH = 50;
}
```

MoveableShape.java

```
import java.awt.*;

public interface MoveableShape
{
    /**
     Draws the shape.
     @param g2 the graphics context
     */
    void draw(Graphics2D g2);
    /**
     Moves the shape by a given amount.
     @param dx the amount to translate in x-direction
     @param dy the amount to translate in y-direction
     */
    void translate(int dx, int dy);
}
```

CarShape.java

```
import java.awt.*;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
```

```
import java.awt.geom.Point2D;
import java.awt.geom.Rectangle2D;

/**
 A car that can be moved around.
 */
public class CarShape implements MoveableShape
{
    /**
     Constructs a car item.
     @param x the left of the bounding rectangle
     @param y the top of the bounding rectangle
     @param width the width of the bounding rectangle
     */
    public CarShape(int x, int y, int width)
    {
        this.x = x;
        this.y = y;
        this.width = width;
    }

    public void translate(int dx, int dy)
    {
        x += dx;
        y += dy;
    }

    public void draw(Graphics2D g2)
    {
        Rectangle2D.Double body
                = new Rectangle2D.Double(x, y + width / 6,
                width - 1, width / 6);
        Ellipse2D.Double frontTire
                = new Ellipse2D.Double(x + width / 6, y + width / 3,
                width / 6, width / 6);
        Ellipse2D.Double rearTire
                = new Ellipse2D.Double(x + width * 2 / 3, y + width / 3,
                width / 6, width / 6);

        // The bottom of the front windshield
        Point2D.Double r1
                = new Point2D.Double(x + width / 6, y + width / 6);
        // The front of the roof
        Point2D.Double r2
                = new Point2D.Double(x + width / 3, y);
        // The rear of the roof
        Point2D.Double r3
                = new Point2D.Double(x + width * 2 / 3, y);
        // The bottom of the rear windshield
        Point2D.Double r4
                = new Point2D.Double(x + width * 5 / 6, y + width / 6);
        Line2D.Double frontWindshield
                = new Line2D.Double(r1, r2);
        Line2D.Double roofTop
                = new Line2D.Double(r2, r3);
        Line2D.Double rearWindshield
                = new Line2D.Double(r3, r4);
```

```
        g2.draw(body);
        g2.draw(frontTire);
        g2.draw(rearTire);
        g2.draw(frontWindshield);
        g2.draw(roofTop);
        g2.draw(rearWindshield);
    }

    private int x;
    private int y;
    private int width;
}
```