American University of Sharjah
School of Engineering
Department of Computer Engineering
P. O. Box 26666
Sharjah, UAE

Instructor: Imran A. Zualkernan
Office: ESB-2066
Phone: 971-6-515 2953
Fax: 971-6-515 2979
e-mail: izualkernan@aus.edu
Semester: Fall 2020

Failure to put your name and ID will result in a 2-point deduction from you grade.

Name: Amir_Mohideen________________
ID : _b00074559___________________

COE312 – Final Examination

Fall 2020

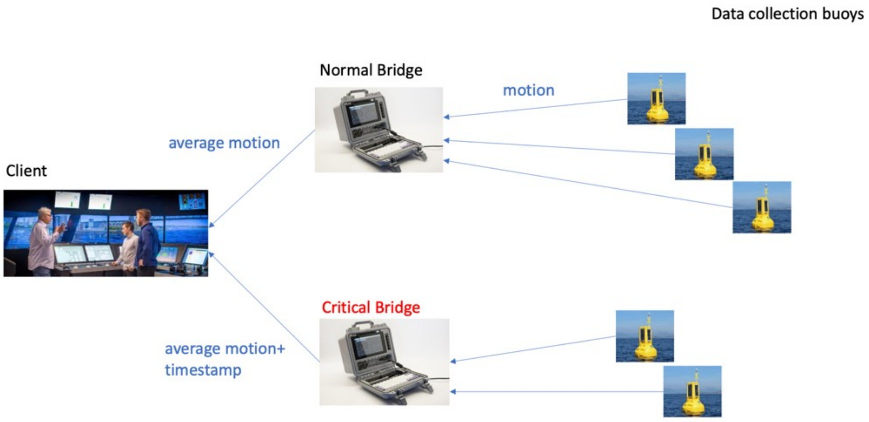
Time: 2 hours

Score: /100

The exam open book and notes. Please paste screenshot of the output with your solution.

The question(s) have been written in a manner such that it is not possible for two students to have the same solution. Therefore, please refrain from the temptation of copying code and changing order of and names of variables, etc. AUS code of conduct will be strictly enforced, and no violation will be tolerated as per AUS policy. Please note that the AUS code of conduct does not discriminate between who copied from whom so it is not advisable to share your solution with others.

Q1 (50 points). Tsunami detection system



Implement *classes* and design for a *tsunami detection system*. The system incorporates data collection buoys that send a floating-point number (representing motion) once every second to a bridge which is an intermediate computer. A bridge calculates the running average of the values received so far from all of its attached buoys and forwards the average motion value to a client which is the control center. After receiving the motion value from a buoy, the bridge clips the value to under 50. In other words, if the value is more than 50 then the bridge changes the value to 50. If the motion value is less than 50, then it is forwarded as is. There are two types of bridges: normal and critical. A normal bridge forwards only the average clipped motion value while the critical bridge forwards the average clipped motion value and the current timestamp (Date).

While designing the system, you must:

1. Use the only the two most appropriate design patterns. You are not allowed to use more than two design patterns. Using more than two design patterns will result in a 50% deduction from the grade.
2. Use a message-oriented design.

Show your output on the sample program below.

Sample Program

```
public static void main(String[] args) {

        Buoy b1 = new Buoy();
        Buoy b2 = new Buoy();
        Buoy b3 = new Buoy();
        Buoy b4 = new Buoy();

        Buoy [] bs1 = {b1, b2};

        NormalBridge bridge1 = new NormalBridge(bs1);


        Buoy [] bs2 = {b3, b4};

        CriticalBridge bridge2 = new CriticalBridge(bs2);

        Bridge [] bridges= {bridge1,bridge2};

        Client c1 = new Client(bridges);

}
```

Expected Output:

```
origin:NormalBridge@2ea0d703
topic:motion
payload:NormalBridgePayload@20673ca7
origin:NormalBridge@2ea0d703
topic:motion
payload:NormalBridgePayload@2f0494ac
received:0.25166315 with no timestamp
received:0.049928457 with no timestamp
origin:CriticalBridge@53b3417
topic:motion
payload:CriticalBridgePayload@4970c6f6
origin:CriticalBridge@53b3417
topic:motion
payload:CriticalBridgePayload@6a061f28
received:0.22669894 with timestamp Sat Dec 12 15:02:22 GST 2020
received:0.16327792 with timestamp Sat Dec 12 15:02:22 GST 2020
origin:NormalBridge@2ea0d703
topic:motion
payload:NormalBridgePayload@71a0a495
received:0.46500286 with no timestamp
origin:NormalBridge@2ea0d703
. . .
```

Grading Rubric:

0-3	4-6	7-8	9-10
Program does not compile or run. OR Program is not related to the problem at hand. OR	• Class hierarchy is correct. AND • Interfaces are not implemented or implemented incorrectly. AND • Most attributes for	• Class hierarchy is correct. AND • Most interfaces are implemented correctly. AND • Some obvious attributes in	• Class hierarchy is correct. AND • All the interfaces are implemented correctly. AND • All the attributes are mentioned. AND

No screenshot is provided. OR Sample program has been changed. OR No design patterns have been used	interfaces are missing. AND • At least one design pattern is appropriately used somewhat. AND • Message-based style is not used.	interfaces are missing. AND • At least one of the design pattern is correctly used but the second design pattern is not appropriate. AND • Message-based style is used.	• Both the design patterns are the correct ones and implemented correctly. AND • Message-based style is used. AND • The output matches exactly for the sample program.

Solution Approach (describe):

<in a few lines explain which two design patterns were used and why>

We need to use the Observer Pattern here since we clearly have entities giving data and entities observing the data

We also need to use singleton pattern for client since in the question there is only one client

Solution (Paste your formatted code here):

```
import OneSubjectMultipleObserver.Bridge;
import OneSubjectMultipleObserver.Buoy;
import OneSubjectMultipleObserver.Client;
import OneSubjectMultipleObserver.CriticalBridge;
import OneSubjectMultipleObserver.NormalBridge;

public class Driver {


        public static void main(String[] args) {

                Buoy b1 = new Buoy();
                Buoy b2 = new Buoy();
                Buoy b3 = new Buoy();
                Buoy b4 = new Buoy();

                Buoy [] bs1 = {b1, b2};

                NormalBridge bridge1 = new NormalBridge(bs1);


                Buoy [] bs2 = {b3, b4};

                CriticalBridge bridge2 = new CriticalBridge(bs2);

                Bridge [] bridges= {bridge1,bridge2};

                Client c1 = new Client(bridges);

        }
```

no singleton here!

```
}

package OneSubjectMultipleObserver;

import java.util.Date;
import java.util.Random;

public class Buoy extends ConcreteSubject implements
Runnable{


    float float_random;
    Random rand = new Random();
    Date d=null;


    public Buoy()
    {

        Thread t = new Thread(this);
        t.start();

        float_random=0;
    }


    public void run()
    {

        float_random=rand.nextFloat();

        if(float_random>0)
        {
            // send a message to whoeever wants it
            Message m = new Message(this, "motion", this ,
float_random, d);
            publishMessage(m);
        }

        try {
            Thread.sleep(1000); //sleep one sec
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

```
package OneSubjectMultipleObserver;

public class Client extends ConcreteObserver implements
Runnable{

private static Client instance;

    public Client(ConcreteSubject[] subject) {
        super(subject);

        Thread t = new Thread(this);
        t.start();

    }

    public static synchronized Client                X No need!
getInstance(ConcreteSubject[] subject)
    {
        if(instance == null)
        {
            instance = new Client(subject);
        }
        return instance;
    }

    public synchronized void update(Message m)
    {
        //System.out.println("received a "+ m.payload+"
message from "+m.origin);


            if(m.topic =="motion")
            {                                                why else
                if( m.d != null)
                {
                    //this.weapon = m.payload;
                    //System.out.println("The GhostBuster
changed weapon to ");
                    System.out.println( m);
                }                                        ?

                else
                {
                    System.out.println( m);

                }
            }
    }

    public void run()
    {

        while(true)
        {
            //System.out.println("The GhostBuster changed
weapon to ");

        }
    }

}

package OneSubjectMultipleObserver;

public class ConcreteObserver implements Observer {

    ConcreteSubject [] subject = null;

    public ConcreteObserver(ConcreteSubject [] subject)
    {
        this.subject = subject;
        for(int i = 0; i<subject.length; i++)
        {
        subject[i].registerObserver(this);
        }
    }

    public void update(Message m)
    {
        System.out.println(m);
    }

}

package OneSubjectMultipleObserver;

import java.util.ArrayList;

public class ConcreteSubject implements Subject{

    private ArrayList observers;

    public ConcreteSubject(){
        observers = new ArrayList();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObsever(Observer o)
    {
        int i = observers.indexOf(o);
        if (i>=0) observers.remove(i);
    }


    public void publishMessage(Message m)
    {
        for (int i = 0; i < observers.size(); i++)
        {
            Observer observer = (Observer) observers.get(i);
            observer.update(m);
        }
    }
}

package OneSubjectMultipleObserver;

import java.sql.Date;

public class CriticalBridge extends ConcreteSubject implements
Runnable,Observer{

    ConcreteSubject[] subjects;

    float motion;

    public CriticalBridge(ConcreteSubject[] subjects)
    {
        super();
        this.subjects = subjects;

        for(int i = 0; i<subjects.length; i++)
        {
            subjects[i].registerObserver(this);
        }

        Thread t = new Thread(this);
        t.start();
    }


    public void run()
```

```
        {
                while(true)
                {
                                // do nothing because the message will be
automatically printed
                            // through update.
                }
        }
```

?

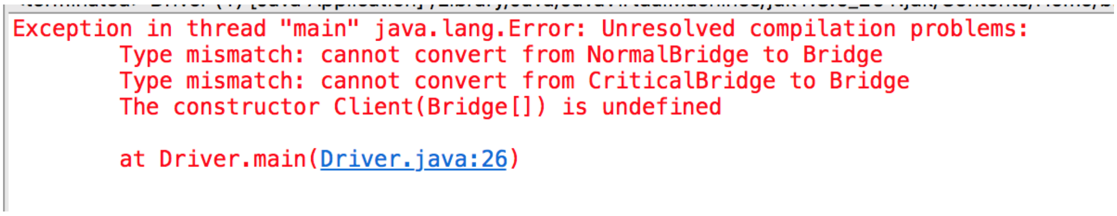
```
        public void update(Message m)
        {
                //System.out.println("received a "+ m.payload+"
message from "+m.origin);
                                if(m.recieved > 50)
                                {

                                        Date d=null;

                                        m.origin= this;
                                        m.payload = this;
                                        m.recieved = 50;
                                        m.d = d;
                                        publishMessage(m);


                                        publishMessage(m);
                                }

        }

}
```

why twice!

```
package OneSubjectMultipleObserver;

import java.util.Date;

public class Message
{
                Object origin;
                String topic;
                Object payload;
                float recieved;
                Date d;

                Message(Object origin, String topic, Object payload,float
recieved,Date d)
```

```
        {
            this.origin=origin;
            this.topic=topic;
            this.payload=payload;
            this.recieved=recieved;
            this.d = d;
        }

        public String toString()
        {

            return "origin:"+origin+"\ntopic:"+topic+"\
npayload:"+payload+"\nrecieved:"+recieved;
        }


}

package OneSubjectMultipleObserver;

public class NormalBridge extends ConcreteSubject implements
Runnable,Observer{

    ConcreteSubject[] subjects;

    float motion;

    public NormalBridge(ConcreteSubject[] subjects)
    {
        super();
        this.subjects = subjects;

        for(int i = 0; i<subjects.length; i++)
        {
            subjects[i].registerObserver(this);
        }

        Thread t = new Thread(this);
        t.start();
    }


    public void run()
    {
        while(true)
        {
                // do nothing because the message will be
automatically printed
            // through update.
        }
    }


    public void update(Message m)
    {
        //System.out.println("received a "+ m.payload+"
message from "+m.origin);

                if(m.recieved > 50)
                {
                    m.origin= this;
                    m.payload = this;
                    m.recieved = 50;
                    publishMessage(m);
                }

    }

}
package OneSubjectMultipleObserver;

public interface Observer {

    void update(Message m);

}
package OneSubjectMultipleObserver;

public interface Subject {

    public void registerObserver(Observer o);
    public void removeObsever(Observer o);
    public void publishMessage(Message m);

}

package OneSubjectMultipleObserver;

public class Bridge {



    Bridge()
    {
        }

}
```

Screenshot (Paste your screenshot here).



```
Exception in thread "main" java.lang.Error: Unresolved compilation problems: 
	Type mismatch: cannot convert from NormalBridge to Bridge
	Type mismatch: cannot convert from CriticalBridge to Bridge
	The constructor Client(Bridge[]) is undefined

	at Driver.main(Driver.java:26)
```

Q2 (50 points). The Zombie Game

Design a Java class called *Zombie* that will be used in a Zombie game. The game will have a large number of Zombies interacting with various characters in the game. A *Zombie* is either dead or alive. A *Zombie* has significantly different behaviors depending on whether it is dead or alive. If alive, a *Zombie* will attack you if you say “hi” to it. On the other hand, when dead, a *Zombie* will shiver in response to a “hi.” Multiple characters in the game can hit a *Zombie* at any time. You kill a *Zombie* by hitting them on the head, but the *Zombie* become alive again if you slap them while they are dead. Regardless, if you kill a Zombie three times by hitting them on the head, then the *Zombie* dies forever and can never be brought back to life via slapping. Note that the three hits do not have come from the same character in the game; any three hits will kill the *Zombie* forever.

While designing the class, you must:

1. Use the only two most appropriate design patterns. You are not allowed to use more than two design patterns. Using more than two design patterns will result in a 50% deduction from the grade

Show your output on the sample program below.

Sample Program

```
public class Main {

        public static void main(String[] args) throws Exception {

                Zombie z1 = new Zombie();

                z1.say_hi();
                z1.hit();
                Thread.sleep(100);
                z1.say_hi();
                z1.slap();
                Thread.sleep(100);
                z1.say_hi();
                z1.hit();
                z1.slap();
                z1.hit();
                Thread.sleep(100);
                z1.say_hi();
                z1.slap();
                z1.hit();
                z1.slap();
                z1.hit();

}
}
```

Expected Output:

```
zombie is attacking.
zombie is shivering...
zombie is attacking.
Hasta lavista baby -- you killed me!
```

```
Why are you talking to the dead?
Exception in thread "main" java.lang.Exception: Please do not desecrate the dead by slapping them!
	at Zombie.slap(Zombie.java:48)
	at Main.main(Main.java:20)
```

Grading Rubric:

0-3	4-6	7-8	9-10
Program does not compile or run. OR Program is not related to the problem at hand. OR No screenshot is provided. OR Sample program has been changed. OR No design patterns have been used	- Class hierarchy is correct. AND - Interfaces are not implemented or implemented incorrectly. AND - Most attributes for interfaces are missing. AND - At least one design pattern is appropriately used somewhat.	- Class hierarchy is correct. AND - Most interfaces are implemented correctly. AND - Some obvious attributes in interfaces are missing. AND - At least one of the design patterns is correctly used but the second design pattern is not appropriate.	- Class hierarchy is correct. AND - All the interfaces are implemented correctly. AND - All the attributes are mentioned. AND - Both the design patterns are the correct ones and implemented correctly. AND - The output matches *exactly* for the sample program.

Solution Approach (describe):

<in a few lines explain which two design patterns were used and why>

We need to use state pattern since there is significantly different behaviours between states (alive, killed & dead) of zombie

We also need to use strategy pattern since each character have different behaviours based on context

Solution (Paste your code here):

```java
import State.ZombieContext;
import Strategy.say_hiBehaviour;

public class Zombie implements Runnable{

	ZombieContext z1 = new ZombieContext();
	say_hiBehaviour s;

	public Zombie()
	{

		Thread t = new Thread(this);
		t.start();


	}


	public void run()
	{
        while(!(z1.state.getClass().getName() ==
"DeadForeverState") )
                {

                }
    }

    void hit()
    {
        z1.setHit();
    }

    void slap()
    {
        z1.setSlap();
    }

    public void say_hi() {

        }

}

import State.ZombieContext;

public class Driver {


        public static void main(String[] args) throws Exception
{

            Zombie z1 = new Zombie();

            z1.say_hi();
            z1.hit();
            Thread.sleep(100);
            z1.say_hi();
            z1.slap();
            Thread.sleep(100);
            z1.say_hi();
            z1.hit();
            z1.slap();
            z1.hit();
            Thread.sleep(100);
            z1.say_hi();
            z1.slap();
            z1.hit();
            z1.slap();
            z1.hit();
    }


}
package State;

public class AliveState implements State {

    public void prev(ZombieContext context)
    {
        System.out.println("The package is in its root state.");
    }

    public void next(ZombieContext context)
    {

        // change state on the boolean
        if(context.hit)
            context.setState(new DeadState());
    }

    public void printStatus()
    {
        System.out.println("Zombie Alive");
    }

}

package State;

public class DeadState implements State {

    public void prev(ZombieContext context)
    {
        if(context.slap)
        context.setState(new AliveState());
    }

    public void next(ZombieContext context)
    {
        // make the package ready
        context.PackageProcessed = true;

        // change state on the boolean
        if( context.hitcount == 3 )
        {
        context.setState(new DeadForeverState());
```

```
                printStatus();
                }
        }

        public void printStatus()
        {
                System.out.println("Hasta lavista baby -- you killed
me!");
        }

}


package State;

public class DeadForeverState implements State {

        public void prev(ZombieContext context)
        {
                System.out.println("Zombie Dead Forever");
        }

        public void next(ZombieContext context)
        {
                System.out.println("Zombie Dead Forever");
        }

        public void printStatus()
        {
                System.out.println("Why are you talking to the dead?");
        }

}


package State;

public interface State {

        public void prev(ZombieContext context);
        public void next(ZombieContext context);
        public void printStatus();

}

package State;

public class ZombieContext {
```

```java
    int hitcount = 0;
    boolean hit,slap = false;

    public State state = new AliveState();

    int day = 1; //or randomize
    // booleans
    boolean PackageReady = false;
    boolean PackageProcessed = false;

    public void previousState()
    {
        state.prev(this);
    }

    public void nextState()
    {
        state.next(this);
    }

    public void printStatus()
    {
        state.printStatus();
    }

    public void setState(State state)
    {
        this.state = state;
    }

    //optional
    public String toString()
    {
        return "Zombie\n"+"State=
"+state.getClass().getName();
    }

    public void setHit() {
        this.hit = true;
        hitcount++;
    }

    public void setSlap() {
        this.slap = true;
    }

}
```

//strategy

```
package Strategy;

public abstract class say_hiBehaviour
{

		abstract void say_hi();

}
```

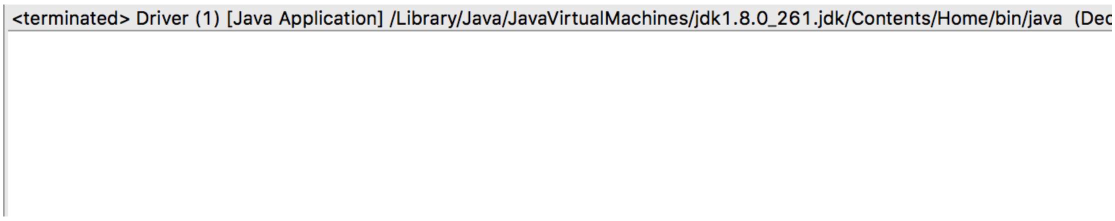
```
package Strategy;

public class attack_say_hi extends say_hiBehaviour{



		void say_hi() {
		System.out.println( "zombie is  attacking");
		}




}
```

```
package Strategy;

public interface say_hi {

}
```

```
package Strategy;

public class shiver_say_hi extends say_hiBehaviour{

	void say_hi() {
		System.out.println("zombie is shivering...");
		}

}
```

```
package Strategy;

public class shiver_say_hi extends say_hiBehaviour{

	void say_hi() {
		System.out.println("zombie is shivering...");
		}
}
```

Screenshot (Paste your screenshot here).



```
<terminated> Driver (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java  (Dec
```