CS 130 SOFTWARE ENGINEERING

DESIGN PATTERNS PRACTICE QUESTIONS

Professor Miryung Kim

UCLA Computer Science

Based on Materials from Miryung Kim, Christine Julien and Adnan Aziz

Q1 INFORMATION HIDING PRINCIPLE

- Modularization can improve software's ease of change. (TRUE/FALSE)
- Interfaces between modules are not allowed to reveal any volatile information (i.e. design decisions that are likely to change). (TRUE/FALSE)
- Information hiding recommends separating a computer program into distinct sections, such that each section addresses a separate concern. (TRUE/FALSE)
- Information hiding recommends using getters and setters methods in Java (e.g., "getItem()" or "setItem (Item e)"). (TRUE/FALSE)
- By following the information hiding principle, you can reduce the number of inter module dependencies (i.e. dependences between modules) in your program. (TRUE/FALSE)

Q1 INFORMATION HIDING PRINCIPLE

- Modularization can improve software's ease of change. (**TRUE**/ FALSE)
- Interfaces between modules are not allowed to reveal any volatile information (i.e. design decisions that are likely to change). (**TRUE**/FALSE)
- Information hiding recommends separating a computer program into distinct sections, such that each section addresses a separate concern. (TRUE/**FALSE**)
- Information hiding recommends using getters and setters methods in Java (e.g., "getItem()" or "setItem (Item e)"). (TRUE/**FALSE**)
- By following the information hiding principle, you can reduce the number of inter module dependencies (i.e. dependences between modules) in your program. (TRUE/**FALSE**)

Q2 RECOGNIZE DESIGN PATTERN

```
class SquarePeg {
        private double width;
        public SquarePeg( double w )        { width = w; }
        public double getWidth()            { return width; }
        public void   setWidth( double w ) { width = w; }
    }

class RoundHole {
        private int radius;
        public RoundHole( int r ) {
            radius = r;
            System.out.println( "RoundHole: max SquarePeg is "
+ r * Math.sqrt(2) );
        }
        public int getRadius() { return radius; }
    }
```

```
        class DemoSquarePeg {
            public static void main( String[] args ) {
                RoundHole         rh = new RoundHole( 5 );
                SquarePegWrapper spa;
                for (int i=6; i < 10; i++) {
                    spa = new SquarePegWrapper( (double) i );
                    // The client uses (is coupled to) the new
interface
                    spa.makeFit( rh );
                }
            }
        }
```

```
class SquarePegWrapper {
          private SquarePeg sp;
          public SquarePegWrapper( double w ) { sp = new
SquarePeg( w ); }
          public void makeFit( RoundHole rh ) {
              double amount = sp.getWidth() - rh.getRadius()
* Math.sqrt(2);
              System.out.println( "reducing SquarePeg " +
sp.getWidth() + " by " + ((amount < 0) ? 0 : amount) + "
amount" );
              if (amount > 0) {
                 sp.setWidth( sp.getWidth() - amount );
                 System.out.println( "   width is now " +
sp.getWidth() );
              }
          }
       }
```

Q3. MARK ALL THAT APPLY

```
class AbstractSort
{
   // Shell sort
public:
   void sort(int v[], int n)
   {
      for (int g = n / 2; g > 0; g /= 2)
        for (int i = g; i < n; i++)
          for (int j = i - g; j >= 0; j -= g)
            if (needSwap(v[j], v[j + g]))
              doSwap(v[j], v[j + g]);
   }
 private:
   virtual int needSwap(int, int) = 0;
   void doSwap(int &a, int &b)
   {
      int t = a;
      a = b;
      b = t;
   }
};
```

```
class SortUp: public AbstractSort
{
Int needSwap(int a, int b)
   {
      return (a > b);
   }
};
class SortDown: public AbstractSort
{
int needSwap(int a, int b)
   {
      return (a < b);
   }
};
```

Q3. MARK ALL THAT APPLY

- This code implements the Strategy design pattern.
- This design pattern defines the outline of algorithms and let subclasses do some of the work.
- This design pattern keeps control over the algorithm's structure.
- This design pattern defines a family of algorithms and make them interchangeable using composition.
- This design pattern puts all duplicated code into a super class and subclasses share the common code

Q3. MARK ALL THAT APPLY

- This code implements the Strategy design pattern.
  - (FALSE)
- This design pattern defines the outline of algorithms and let subclasses do some of the work.
  - (TRUE)
- This design pattern keeps control over the algorithm's structure.
  - (TRUE)
- This design pattern defines a family of algorithms and make them interchangeable using composition.
  - (FALSE)
- This design pattern puts all duplicated code into a super class and subclasses share the common code (TRUE)

Q4. MARK ALL THAT APPLY

```
public interface IBehaviour {
	public int moveCommand();
}
public class AgressiveBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tAgressive   Behaviour:   if
find another robot attack it");
		return 1;
	}
}
```

Q4. MARK ALL THAT APPLY

```
public class DefensiveBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tDefensive  Behaviour:  if
find another robot run from it");
		return -1;
	}
}
public class NormalBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tNormal    Behaviour:    if
find another robot ignore it");
		return 0;
	}
}
```

```
public class Robot {
	IBehaviour behaviour;
	String name;
	public Robot(String name)
	{	this.name = name;}
	public void setBehaviour(IBehaviour behaviour)
	{	this.behaviour = behaviour;}
	public IBehaviour getBehaviour()
	{	return behaviour;}
	public void move()
	{
		System.out.println(this.name + ": Based on current
position" + "the behaviour object decide the next move:");
		int command = behaviour.moveCommand();
		System.out.println("\tThe   result   returned   by
behaviour object " +"is sent to the movement mechanisms " +
" for the robot '"  + this.name + "'");
	}
	public String getName() {
		return name;}
	public void setName(String name) {
		this.name = name;}
}
```

Q4. MARK ALL THAT APPLY

- This code implements the Strategy design pattern.
- This design pattern defines the outline of algorithms and let subclasses do some of the work.
- This design pattern keeps control over the algorithm's structure.
- This design pattern defines a family of algorithms and make them interchangeable at runtime.
- This design pattern puts all duplicated code into a super class and subclasses share the common code.

Q4. MARK ALL THAT APPLY

- This code implements the Strategy design pattern.
- (TRUE)
- This design pattern defines the outline of algorithms and let subclasses do some of the work.
- (FALSE)
- This design pattern keeps control over the algorithm's structure.
- (FALSE)
- This design pattern defines a family of algorithms and make them interchangeable at runtime.
- (TURE)
- This design pattern puts all duplicated code into a super class and subclasses share the common code.
- (FALSE)

QUESTIONS?