Com S 362 Practice Final

NOTE:

In this practice exam I have focused on patterns and UML drawings. You should also be prepared for questions related to basic design patterns, scenarios, and other topics from the first half of the semester.

1. (PROBLEM CORRECTED) For each term on the left, select the best description on the

__j_ accidental complexity	a. A methodology-independent graphical modelling language.
__x_state diagram	b. One of the UML interaction diagrams.
__k_delegation	c. A technique for identifying candidate domain concepts.
_f__functional design paradigm	d. Another name for activity diagram.
_l__CRC Cards	e. The long term consequence of fixing bugs.
_a__UML	f. Related to mathematical composition.
_m__Flyweight Pattern	g. An architectural style made famous by UNIX.
_o__Strategy Pattern	h. An analytic technique that simplifies complex systems by focusing on important aspects while obscuring or ignoring less important details.
_i__EARS	i. A specific syntax for writing clear requirements.
_b__Sequence diagram	j. Problems created or aggravated by technological choices and, thus, which are possible to 'fix' with improved technology.
_g__Pipes and Filters	k. Forwarding a method call to another object for implementation/execution.
	l. Introduced in a paper by Kent Beck.
	m. Uses sharing to support a large number of fine-grained objects efficiently.
	n. A line between two boxes.
	o. Lets you define a new operation without changing the classes of the elements to which the operation is applied.
	x. None of the Above

right. Put `

2. Fill in the blanks.

Creational Patterns:
Prototype
Abstract Factory
Builder
Singleton
Factory Method

a. _any of 5____________________, ________________________________, and ________________________________ are creational patterns.

Behavioral Patterns:
Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

b. ___any of 11_________________________ and ________________________________ are behavioral patterns.

c. The goal of design is to create systems that can be easily ____modified_______.

GRASP resp. patterns
Creator
Information Expert
Low Coupling
Controller
High Cohesion

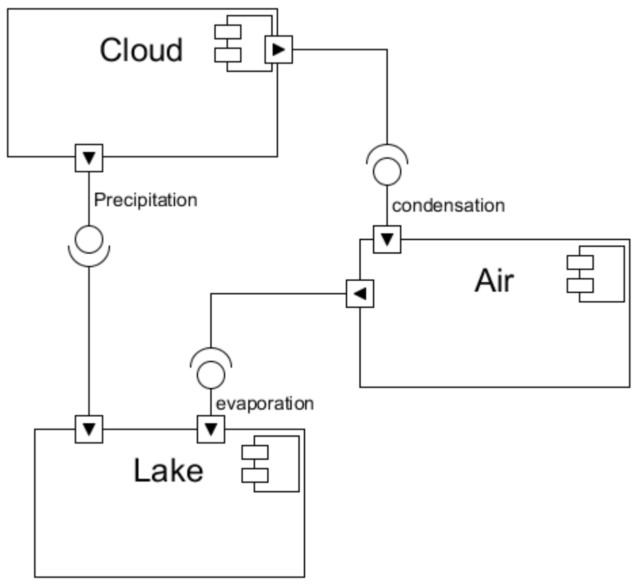
d. __any of 5____________________ and _______________________ are two responsibility assignment patterns included in GRASP.

e. The memento pattern is often used to implement ___undo____________________ behavior.

f. In a restaurant, when the wait staff writes your order down and then submits the order to the kitchen for preparation, the restaurant is employing the

____command________________________ Pattern.

3. Use the following diagram to answer the questions at right.



a) What kind of UML diagram is this? component

b) Which element is the consumer of the precipitation interface? LAKE

c) Which element is the implementer/provider of the precipitation interface? CLOUD

*drawing based on one from lucidchart.com.

4. Compare and contrast each of the following terms. You will *not* receive credit for attempts to define or describe the meaning of the terms. You *must* identify (succinctly) some meaningful commonality they share (both start with “s” doesn’t count) and some difference(s) that distinguish(es) them.

a. Proxy Pattern and Façade Pattern

Similarity: Both are structural patterns which expose an interface to an existing implementation.

Difference: Proxy exposes the same interface as the existing implementation. Proxy "intercepts" calls to the existing implementation, allowing it to control access to the underlying object. Proxies can also be used to create a "local presence" for remote resources (as in network gateways), and to delay the creation of resource intensive components (lazy instantiation). Proxy can be used for purposes such as load balancing, security.

Facade pattern is used to provide a simple or restricted interface to more complex interface(s). Facade may help abstract complex subsystem functionality by providing a uniform simple interface for the subsystem(s).

b. Refactoring code vs. rewriting code

Similarity: Both are approaches to revising an existing body of code usually to add or change functionality.

Difference: Refactoring aims to adjust design decisions without changing functionality, until the design is such that desired changes in functionality can be accomplished with a pointwise addition or subtraction of code. Refactoring focuses on keeping the change well-controlled by working in a sequence of small changes, each of which must pass the pre-existing test suite before further change is made. Refactoring attempts to achieve necessary design changes with minimal change to existing code. Refactoring is a technique for managing risk.

Rewriting often attempts to implement the full change, regardless of scope or difficulty, in a single pass. Rewriting implicitly invalidates existing tests, meaning that it is more risky and typically far more difficult to bring to a fully debugged, releasable state.

c. Observer vs. Mediator Pattern

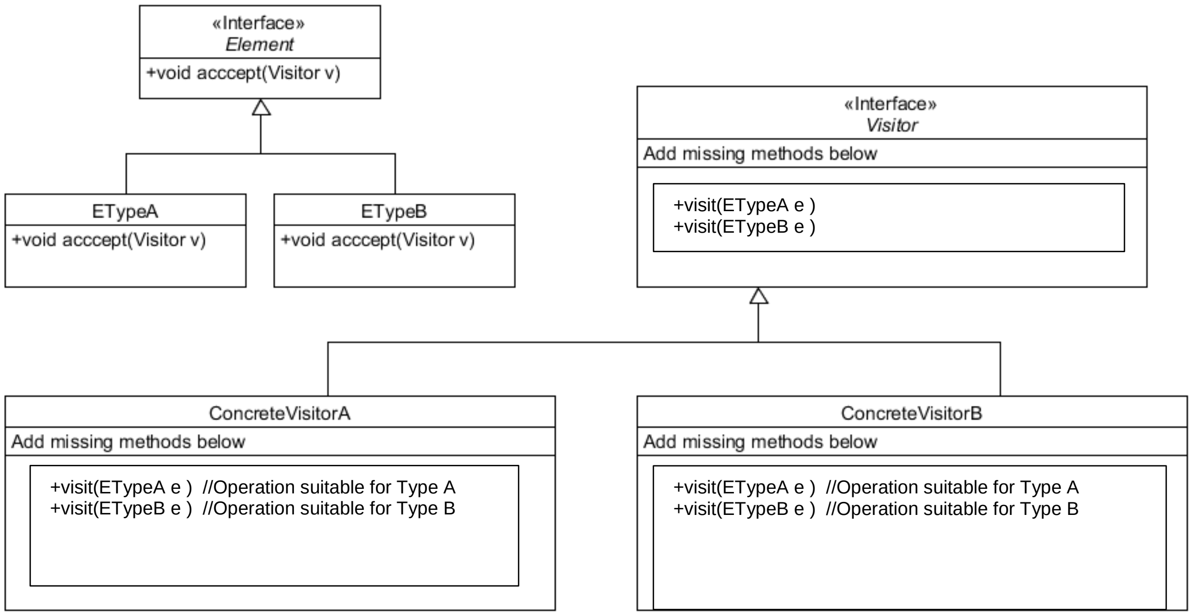
Similarity: Both are behaviour patterns which helps to derive a better design for interactions (or how they behave) among objects. Both potentially deal with many to many interactions.

Difference: Observer pattern is used to keep an arbitrary collection of client objects aware of updates to some objects which the "publisher" object maintains. Mediator provides a common point of contact and common interface through which two different classes of object can interface. Observer distributes updates, like newstand. Mediator acts more like a switch board; providing a common place where objects can exchange messages meant for specific receivers.

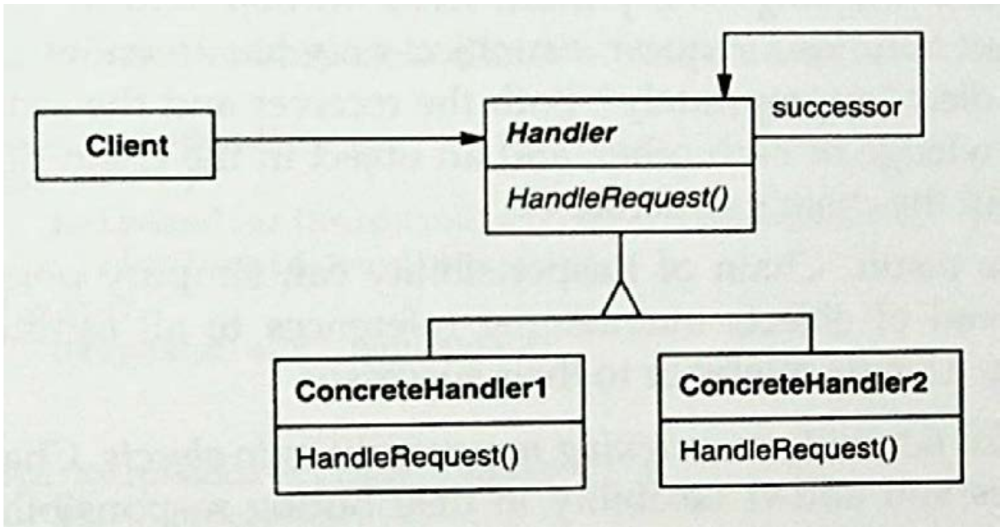
5. Write the code for a Singleton Class named MyFinal.

```
/* ignoring synchronization issues */

public class MyFinal {

private static MyFinal instance=null;

            //constructor
            private MyFinal() {
            }

            //simple singleton- not synchronised
            public static MyFinal getInstance(){
                if(instance == null){
                    Instance = new MyFinal();
                        }
                        return instance;
            }

}
```

6. The class Diagram below shows classes that are part of a visitor pattern implementation. Complete the diagram by supplying the missing methods in ConcreteVisitorA and ConcreteVisitorB. (PROBLEM CORRECTED)

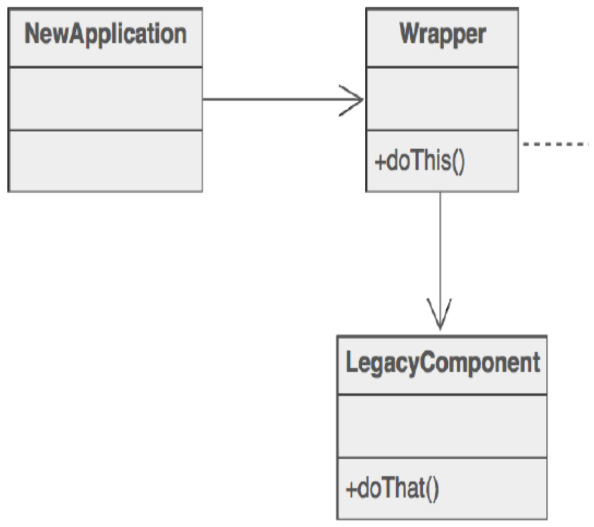

«Interface»
Element
+void acccept(Visitor v)
ETypeA
+void acccept(Visitor v)
ETypeB
+void acccept(Visitor v)
«Interface»
Visitor
Add missing methods below
+visit(ETypeA e)
+visit(ETypeB e)
ConcreteVisitorA
Add missing methods below
+visit(ETypeA e) //Operation suitable for Type A
+visit(ETypeB e) //Operation suitable for Type B
ConcreteVisitorB
Add missing methods below
+visit(ETypeA e) //Operation suitable for Type A
+visit(ETypeB e) //Operation suitable for Type B

7. What design pattern does the following diagram depict?



ANSWER: Chain of Responsibility (Clue would be to spot the successor)

8. What pattern does the following class drawing represent?



Adapter

(Clue would be to look at the legacy component: Wrapper is exposing a different interface.)

9. What Design Pattern do the following two exemplify?

```
public class Special extends Base {

        @Override
        public int getOffset() {
                return 7;
        }

}
```

```
public abstract class Base {
        public void method1(){
            //implementation omitted to save space.
        }

        public int method2(int input){
            return input+getOffset();
        }

        public abstract int getOffset();
}
```

Template method

(Clue would be to spot the abstract method -- which isn't defined in the base class -- being used by the base class, but implemented in the subclass.)

10. Consider the following classes and interface. What design pattern is PatternParticipant using?

Decorator

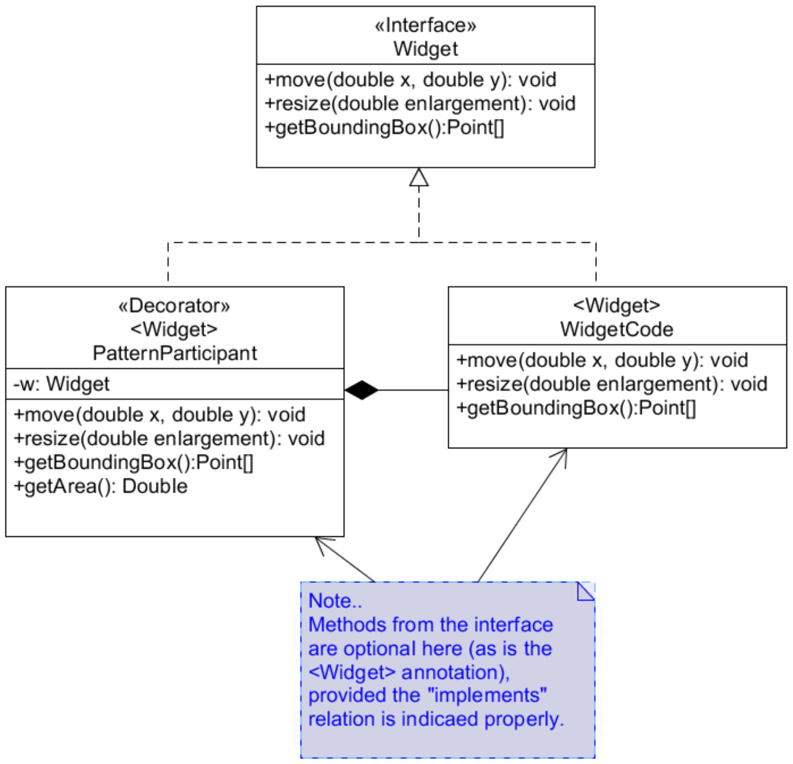
(Clue: -- notice the delegation of all but the new getArea() method. PatternParticipant creates a “fancier” version of WidgetCode -- one that has an area method – without making any change to WidgetCode.)

```
import java.awt.Point;

public interface Widget {
	public void move(double x, double y);
	public void resize(double enlargement);

	/**
	 * @return points corresponding to upper left
	 *        and lower right corners.
	 */
	public Point[] getBoundingBox();
}
```

```
import java.awt.Point;

public class WidgetCode implements Widget {
	private Point ulCorner;
	private Point lrCorner;

	public WidgetCode( Point ulCorner, Point lrCorner){
		this.ulCorner = ulCorner;
		this.lrCorner = lrCorner;
	}

	@Override
	public void move(double x, double y) {
		// ... implementation omitted for space
	}

	@Override
	public void resize(double enlargement) {
		// ... implementation omitted for space
	}
	@Override
	public Point[] getBoundingBox() {
		Point[] rval = new Point[2];
		rval[0] = ulCorner;
		rval[1] = lrCorner;
		return  rval;
	}
}
```

```
import java.awt.Point;

public class PatternParticipant implements Widget{

	private Widget w;

	public PatternParticipant( Point ulCorner, Point lrCorner){
		w = new WidgetCode(ulCorner, lrCorner);
	}

	@Override
	public void move(double x, double y) {
		w.move(x, y);
	}

	@Override
	public void resize(double enlargement) {
		w.resize(enlargement);
	}

	@Override
	public Point[] getBoundingBox() {
		return w.getBoundingBox();
	}

	public Double getArea(){
		Point[] bb = getBoundingBox();
		return Math.abs((bb[0].getX()-bb[1].getX())
			* (bb[0].getY() - bb[1].getY()));
	}
}
```

11. Give a complete class diagram for the code in the preceding problem.


«Interface»
Widget
+move(double x, double y): void
+resize(double enlargement): void
+getBoundingBox():Point[]
«Decorator»
<Widget>
PatternParticipant
-w: Widget
+move(double x, double y): void
+resize(double enlargement): void
+getBoundingBox():Point[]
+getArea(): Double
<Widget>
WidgetCode
+move(double x, double y): void
+resize(double enlargement): void
+getBoundingBox():Point[]
Note..
Methods from the interface are optional here (as is the <Widget> annotation), provided the "implements" relation is indicaed properly.

12. Which of the following two interfaces is most representative of the Builder Pattern?

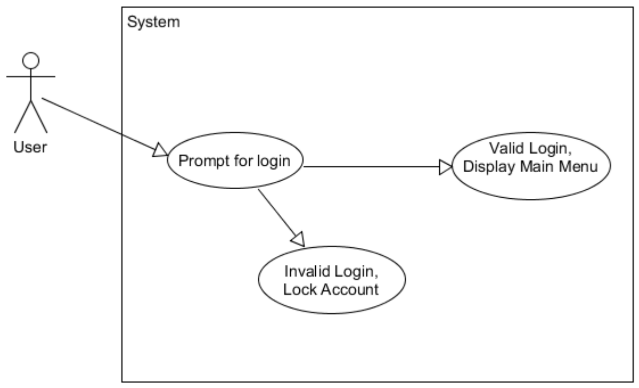
```
public interface MyMaker1 {
        public Pink newPinkInstance();
        public Orange newOrangeInstance();
        public Red newRedInstance();
        public Blue newBlueInstance();
        public Square newSquareInstance();
}
```

```
public interface MyMaker2 {
        public Document getInstance(XmlFile file);
}
```

ANSWER: MyMaker2. A Builder returns a fully assembled object hierarchy based on a specification. Document is a complex type built from an xml specification.

Clue: MyMaker1 is clearly an abstract factory: it returns simple, possibly initialized, objects, not complex object assemblies.

13. What is wrong with this Use Case Drawing?

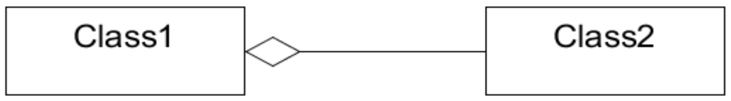


ANSWER: It is like a flow chart, which shows the steps of how user logins to the system: if the login is valid, then display main menu, else lock the account!

Use case diagrams are used to describe a set of actions (use cases) that some system or systems should or can perform in collaboration with, or more precisely "triggered by" one or more actors or external users of the system. Each use case should provide some observable and valuable result to the actors or other stakeholders of the system. Inside the system, each use case may or may not "include" or "extend" other use cases (shown by dashed arrows). But, no use case can have an arrow to other use cases without inclusion or extension.

More info: https://www.uml-diagrams.org/use-case-diagrams.html

14. What does the following association mean?



ANSWER: In class diagrams, when two classes are connected by a simple line (no diamond or arrowhead), then we have a simple association. If the diamond is left empty, it signifies it is an aggregation. This relation is stronger than a simple association. In this case a class1 aggregates class2. If the diamond is black, this means it is a composition, which is even stronger than an aggregation, because the aggregated class cannot be aggregated by other classes. Its "life" depends on the container.

More info: http://aviadezra.blogspot.be/2009/05/uml-association-aggregation-composition.html