UML diagrams for the design patterns mentioned in the exam are provided at the end of the exam. For some questions the details of the diagram structure will be useful. For other questions, the details of the diagram structure are not especially relevant. Diagrams are provided in either case.

*True/False, if false explain why in 1-2 sentences (**2 points each)***

1. A *Remote Proxy* helps to hide all the low-level details of exception handling from the proxy *Client*.

   False, not all of these details can be resolved by the proxy itself. The remote proxy will throw exceptions to the Client who can determine how they should be handled.

2. The *Waterfall* process model was invented to address risks not addressed by the *Spiral* process model.

   False, the Spiral model was invented to address risks not addressed by the Waterfall model.

3. The *Template Method* is used to instantiate an object by choosing between a set of potential concrete classes (i.e. sub-classes of some common base-class).

   False, the Template Method provides a skeleton algorithm with steps to be determined by concrete sub-classes.

4. The *Sprint Review* meeting allows developers to show a demo of their work, so anyone in the company can see what was accomplished.

   True

5. Managers should not attend *Daily Scrum* meetings.

   False, everyone is invited to attend Daily Scrum meetings.

6. *Sprint Backlog* items describe tasks that developers should complete.

   True

Short Answer, 1-3 sentences ***(4 points each)***

7. In terms of their *intent* (not their implementation), what is a difference between the *Builder Pattern* and the *Abstract Factory*?

   The Builder hides the details of composing complex objects from the Client whereas the Abstract Factory provides an interface for creating instances from a family of related classes.

8. Describe one benefit of *Agile* methods which make them more suitable for some projects than the more traditional software processes (e.g. *Waterfall*)?

   Agile methods are more able to respond to unexpected changes in customer requirements.

9. Describe a situation or software requirement that would motivate someone to use the *Command* pattern?

   In the case that there is a requirement to suspend, record, or queue actions for invocation.

10. What is the purpose of the *Product Owner* in the *Scrum* methodology?

    The Product Owner helps to determine and prioritize product features.

11. What is one problem that is addressed by *Continuous Integration* in *Extreme Programming*?

    Merge conflicts.

12. (parts a and b).

Consider the following *Component* interface `ConsoleIO` (below), which is used to represent an object that can `read` and `write` text to the console.

```
public interface ConsoleIO {
	public void write(String s);
	public String read();
}
```

12.a. Complete this implementation for an abstract *Decorator* for the `ConsoleIO` interface which simply delegates the implementation of its methods to another `ConsoleIO` component instance. New code may need to be added at any place, the location of code to be added is not explicitly marked. **(8 points)**

```
abstract class ConsoleIODecorator implements ConsoleIO
{
	ConsoleIO delegate;

	ConsoleIODecorator(ConsoleIO delegate) {
		this.delegate = delegate;
	}

	public void write(String s) {
		delegate.write(s);
	}

	public String read() {
		return delegate.read();
	}
}
```

12.b. Complete this implementation for a concrete *Decorator*, `NoNullDecorator`, for the `ConsoleIO` interface. This Decorator adds the following behavior to the component it wraps (i.e. that it delegates to). In the case that the wrapped component returns a `null` String, the decorator will instead return the empty `String` "". In the case that a `null String` is written to the decorator, an empty `String` "" is passed to the wrapped component. New code may need to be added at any place, the location of code to be added is not explicitly marked. **(8 points)**

```
class NoNullConsoleDecorator extends ConsoleIODecorator
{
     NoNullConsoleDecorator(ConsoleIO console) {
          super(console);
     }

     public void write(String s) {
          if(s == null) {
               s = "";
          }
          super.write(s);
     }

     public String read() {
          String input = super.read();
          if(input == null) {
               input = "";
          }
          return input;
     }
}
```

13. An object `o1` “matches” an object `o2` if it implements `Matchable` (see below) and `o1.match(o2)` is true.

The “first match” for a given array, `arr`, and object, `obj`, is defined as the object in `arr` at the smallest index which “matches” `obj`.

Complete this implementation of a *generic method*, `replaceFirst`. Its behavior is to replace the “first match”, for a given array and given object, with the given object. The method must guarantee the replacement to be type compatible with the array.

New code may need to be added at any place, the location of code to be added is not explicitly marked.

(12 points)

```
interface Matchable<T> {
      public boolean match(T other);
}

//Add code below
```

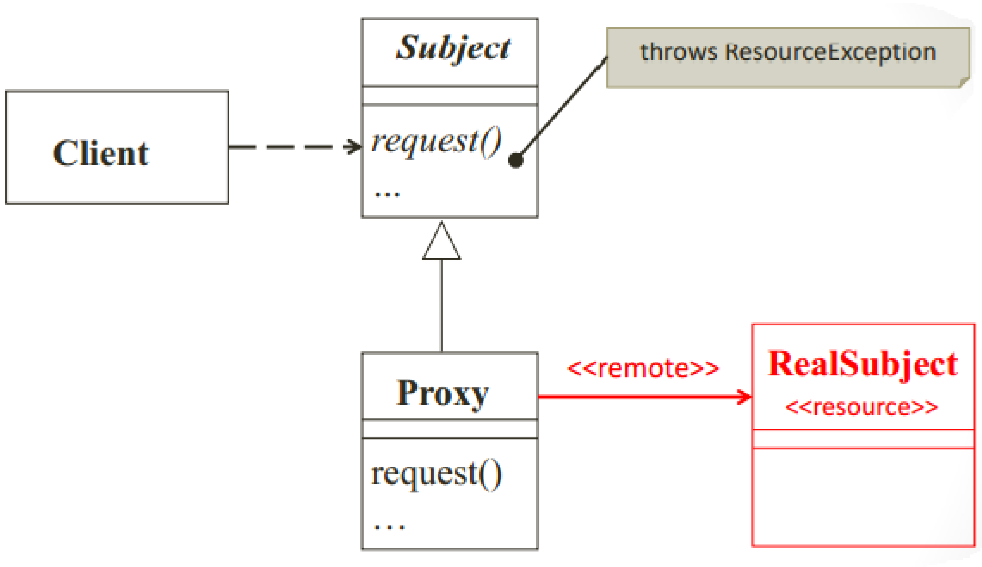
```
static <T extends Matchable<T>>
void replaceFirst(T obj, T[] arr) {

     for(int i=0; i < arr.length; i++) {
          if(arr[i].match(obj)) {
               arr[i] = obj;
               return;
          }
     }

}
```

14. Assuming `toString()` is defined in the class `Object`: will the following method compile? If not, why? **(5 points)**

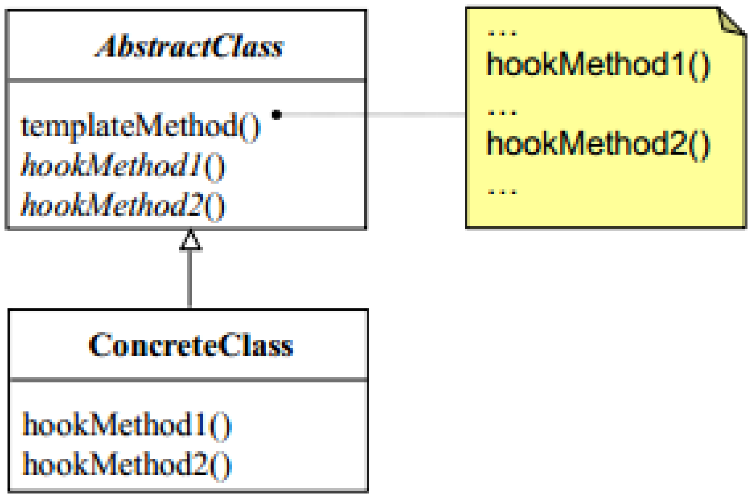
```
static void print(List<? extends Number> list) {
    for (Integer i : list) {
           System.out.print(i.toString() + " ");
         System.out.println();
     }
}
```

No, because Integer is the same as Number and it also is not a super-type (i.e. super-class or interface) of Number. Since the list could contain any kind of Number, it is not safe to assume it contains Integer.

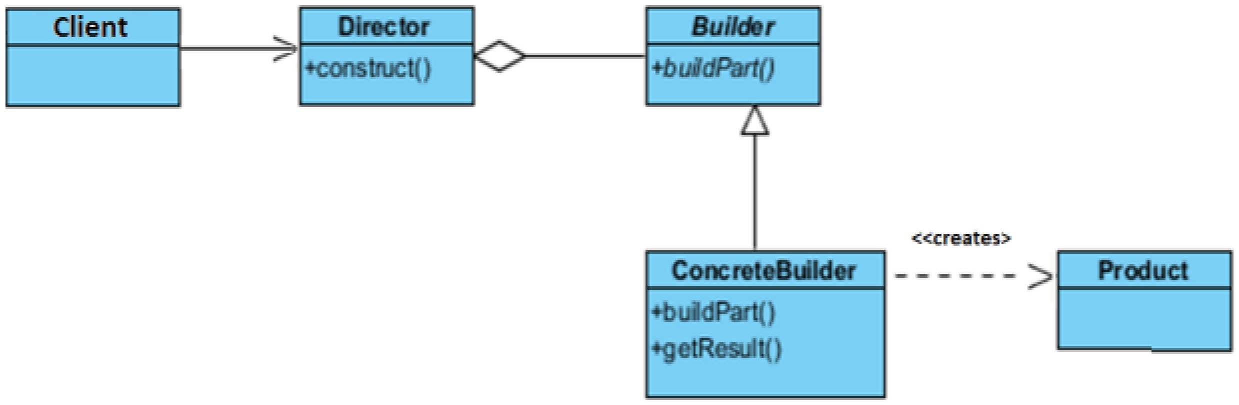
REMOTE PROXY



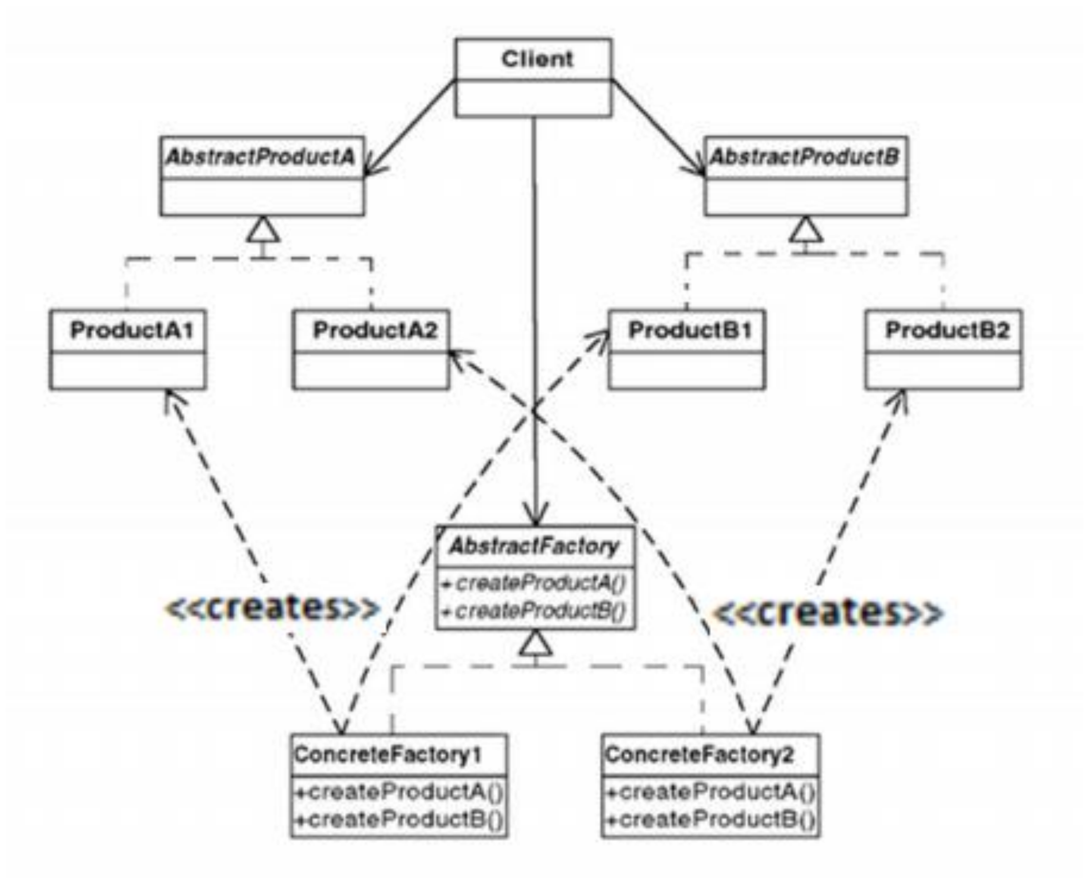
TEMPLATE METHOD



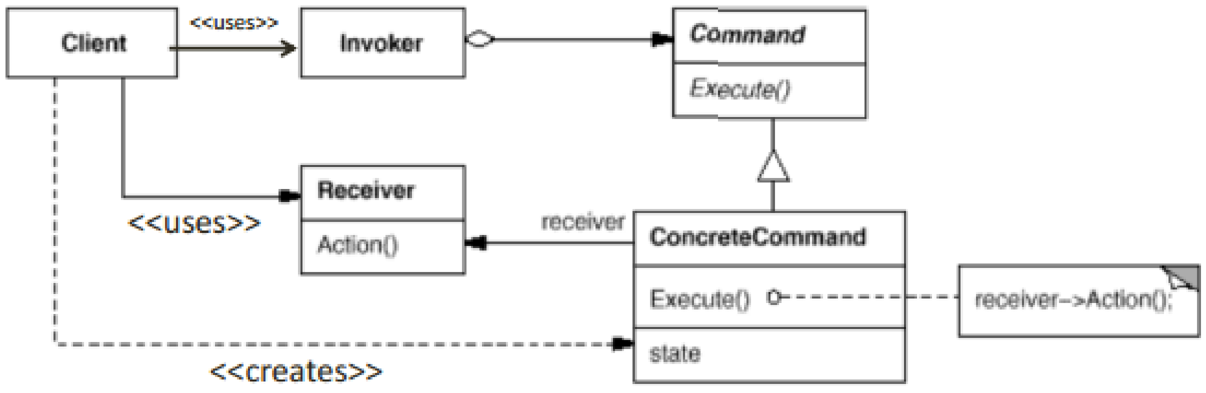
BUILDER



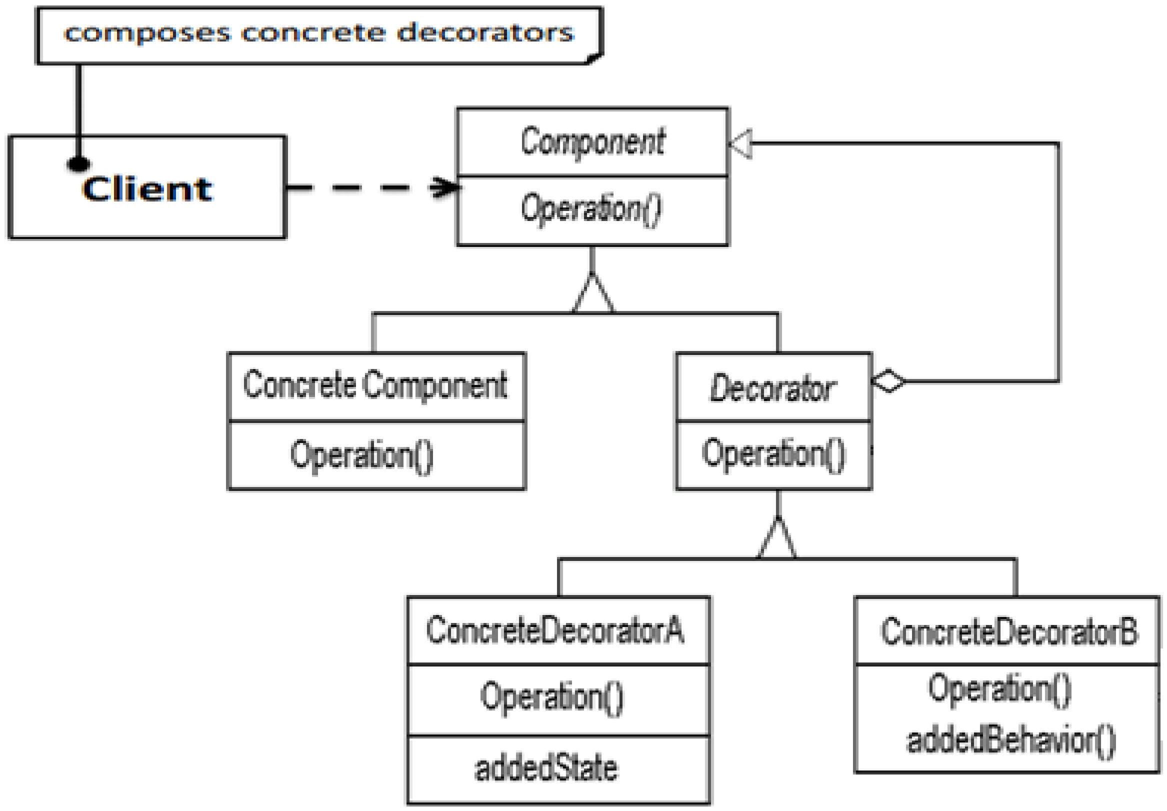
ABSTRACT FACTORY



COMMAND



DECORATOR



composes concrete decorators
Client
Component
Operation()
Concrete Component
Operation()
Decorator
Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
addedBehavior()