final quiz solution

Question 1

Mohsin needs to create various user objects for his University learning platform. What is the act of creating an object called?

☐ **object invocation**

Incorrect

Incorrect. Objects are not "invoked" like methods!

☐ **class creation**

Incorrect

Incorrect. What is an instance of a class called?

☐ **object realization**

Incorrect

Incorrect. Although an instance of a class is called an object, realization is not the correct term.

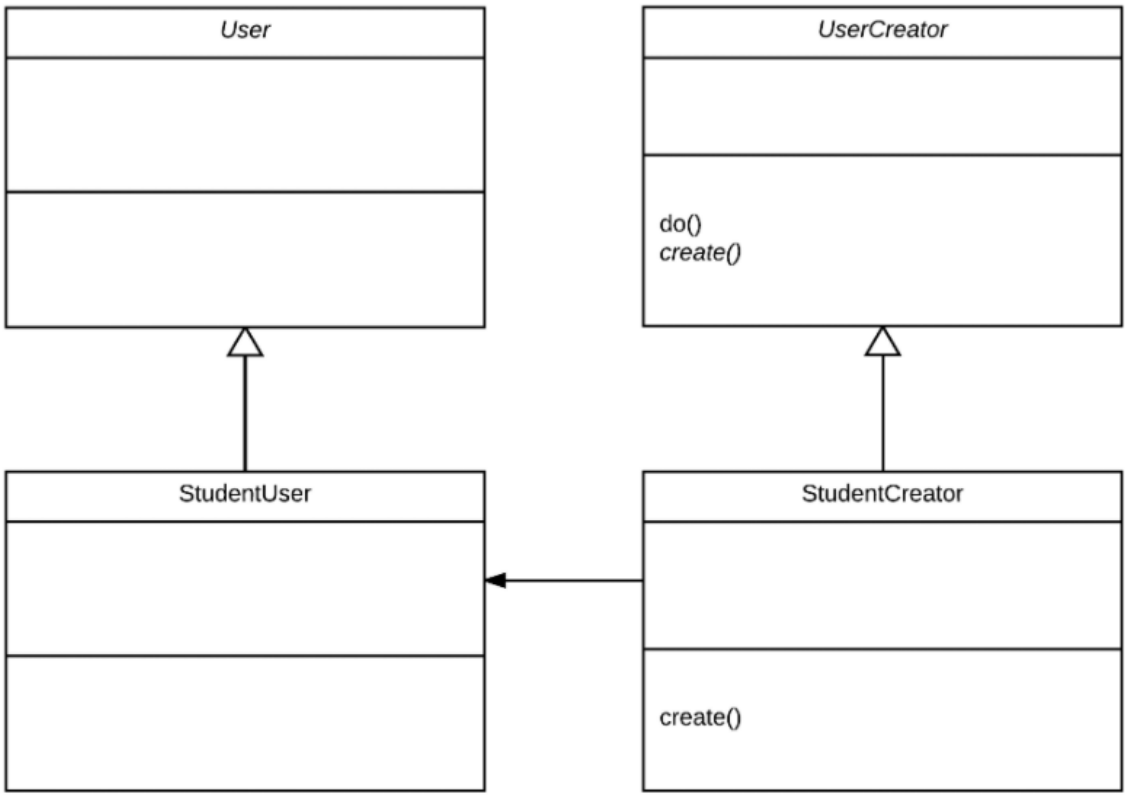
☐ **concrete instantiation**

Correct

Correct! Concrete instantiation is when an object of a class is actually created.

Question 2

Mohsin has a superclass that performs various operations on these user objects - Student, Professor, Assistant, for example. He wants the subclass to determine which object is created. This is sketched below in a UML diagram for the StudentUser class. What is this design pattern called?



☐ **Template Pattern**

Incorrect

Incorrect. A Template sets out a general recipe, and has subclasses decide how to implement some of the steps.

☐ **Factory Method Pattern**

Correct

Correct! The Creator superclass in the Factory Method pattern has operations that operate on an object, but has the actual creation of that object outsourced to an abstract method that must be defined by the subclass

☐ **Simple Factory**

Incorrect

Incorrect. A Simple Factory is simply an object that is tasked with concrete instantiation.

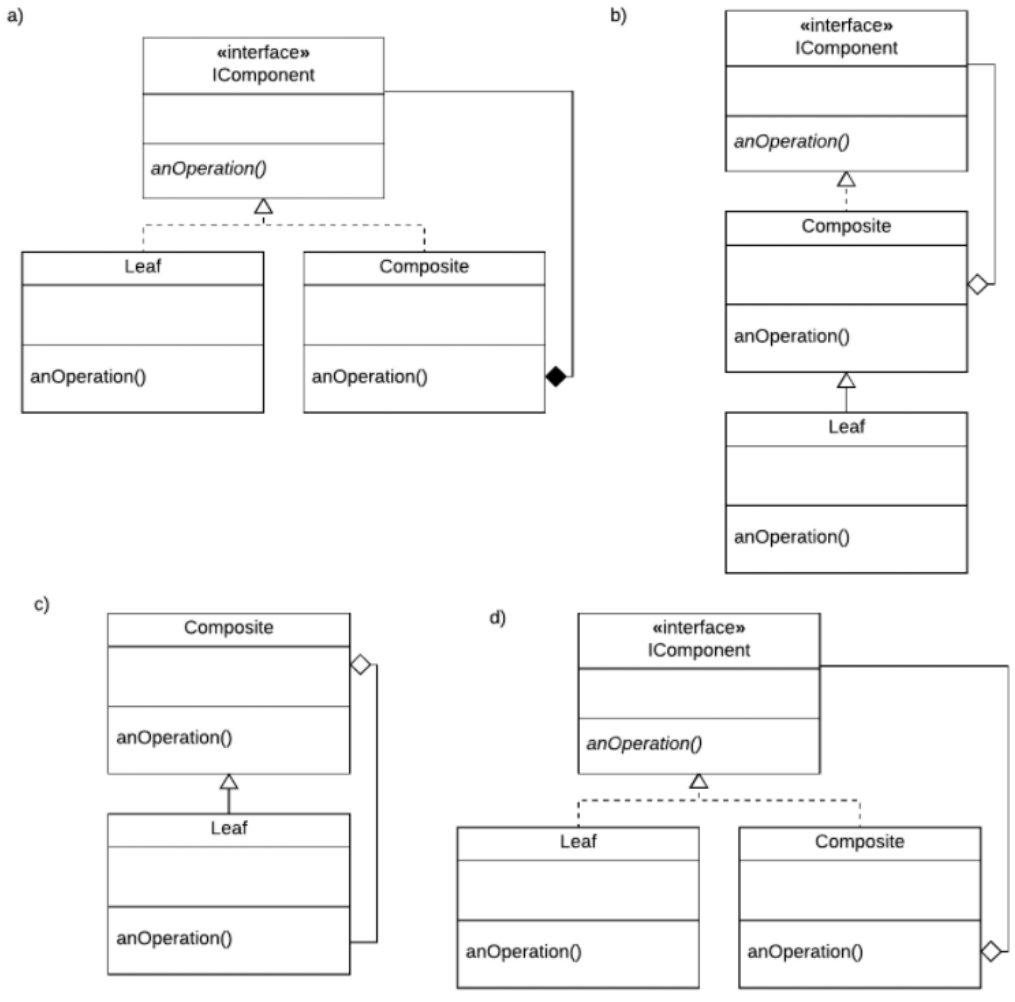
☐ **Composite Pattern**

Incorrect

Incorrect. Composite Pattern is about recursive structure objects.

Question 3

Select the correct UML class diagram representation of the Composite Pattern:



☐ **a)**

Incorrect

Incorrect. The strong "has-a" relationship depicted here (with a filled-in diamond) is not a constraint of the Composite Pattern.

☐ **b)**

Incorrect

Incorrect. This would require Leaf to inherit behaviour from the Composite class, which is not usually a desirable feature! It is also not clear from this diagram that the Composite class can contain Composite objects.

☐ **c)**

Incorrect

Incorrect. This requires Leaf to inherit behaviour from the Composite class, and it begs the question of why it is necessary to define an interface.

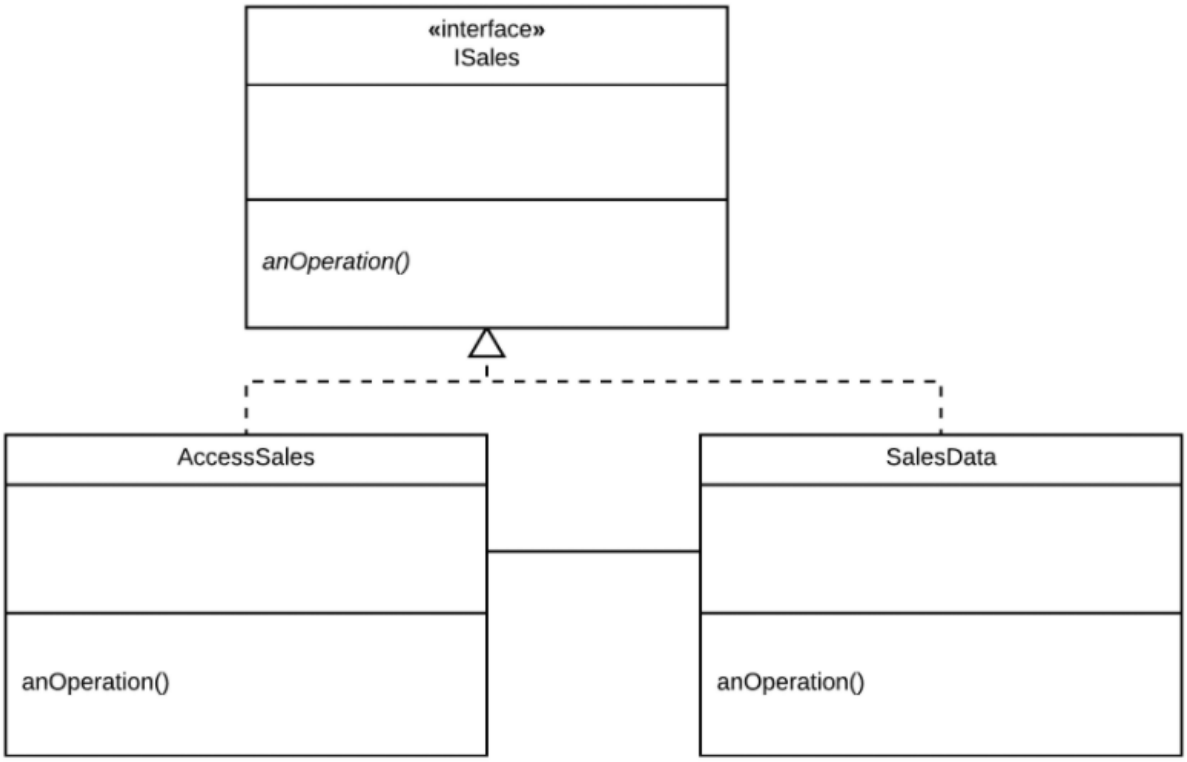
☐ **d)**

Correct

Correct! Both the component and Leaf classes implement the component interface (or they can inherit from a component superclass). The Composite class aggregates objects with this interface.

Question 4

Yola is programming for a grocery store system. She has a complex SalesData class that updates inventories and tracks sales figures, and a lightweight AccessSales class that will give select sales data to a user, depending on their credentials. AccessSales delegates to SalesData when more complex data is needed. This situation is shown below. Which Pattern is this?



☐ **Proxy Pattern**

Correct

Correct! This is a proxy. The AccessSales object acts as a lightweight version of the SalesData class.

☐ **Singleton Pattern**

Incorrect

Incorrect. Singleton is all about having only one object of a class.

☐ **Decorator Pattern**

Incorrect

Incorrect. In Decorator Pattern, functionality is aggregated by "stacking" objects.

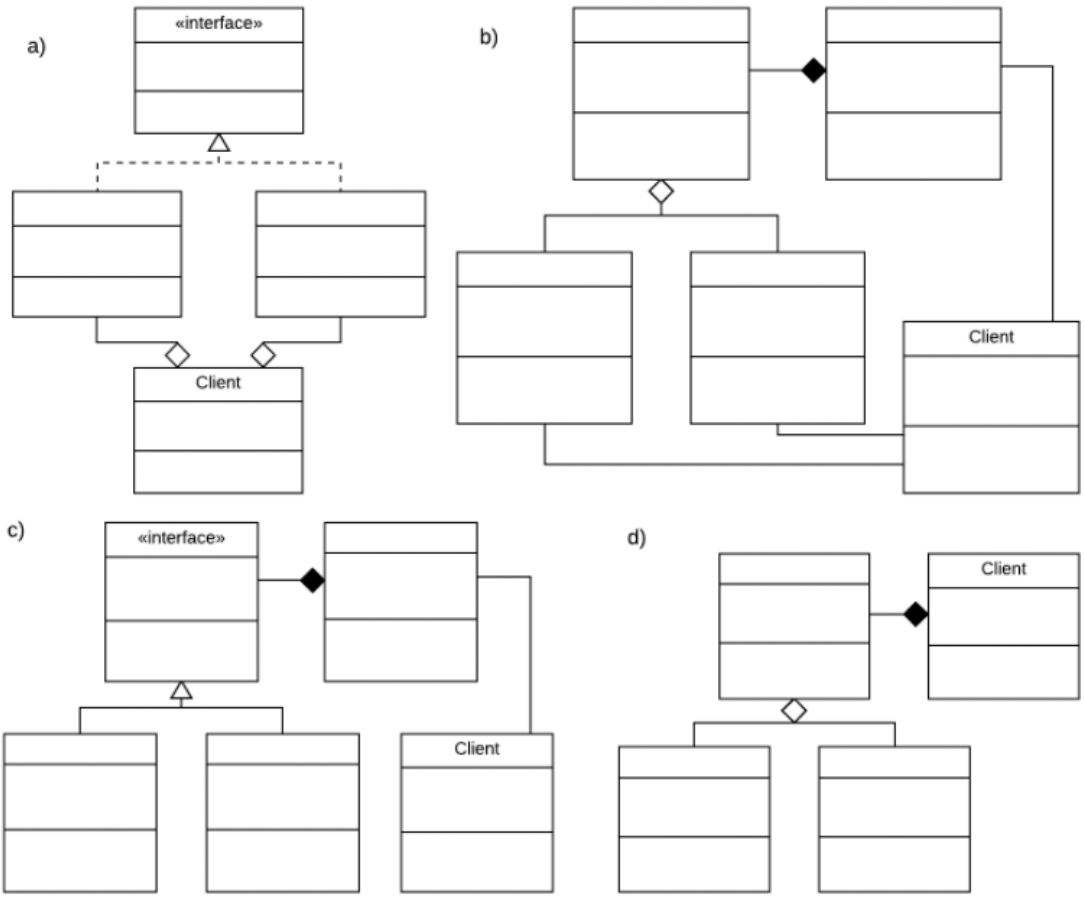
☐ **Facade Pattern**

Incorrect

Incorrect. A Facade Pattern hides the complexity of a subsystem with a simpler interface.

Question 5

Which of these UML class diagrams shows the Facade pattern?



☐ **a)**

Incorrect

Incorrect. This situation shows what the program might look like before you implement a Facade Pattern!

☐ **b)**

Incorrect

Incorrect. The point of the Facade Pattern is to have fewer interactions with your client!

☐ **c)**

Correct

Correct! The client interacts with only the Facade. The Facade then manages the subsystem.

☐ **d)**

Incorrect

Incorrect. The Facade Pattern is meant to encapsulate a complex system. This example does not have adequate separation.

Question 6

What is the difference between the Factory Method and a Simple Factory?

☐ **In the factory method pattern, the factory itself must be instantiated before it starts creating objects. This is usually done with a dedicated method.**

Incorrect

Incorrect. A Factory Method is not instantiated because it is a method, not a class.

☐ **A simple factory instantiates only one kind of object.**

Incorrect

Incorrect. Both types of factory generally instantiate more than one type of object.

☐ **In Factory Method, concrete instantiation is done in a designated method, where a Simply Factory creates objects for external clients**

Correct

Correct! This is a pretty good short definition of a factory method.

☐ **Simple factories cannot be subclassed.**

Incorrect

Incorrect. Both types of factory generally instantiate more than one type of object.

Question 7

José wants to build behaviours by stacking objects and calling their behaviours with an interface. When he makes a call on this interface, the stack of objects all perform their functions in order, and the exact combination of behaviours he needs depends what objects he stacked and in which order. Which Design Pattern best fits this need?

☐ **Decorator Pattern**

Correct

Correct! Decorator is a great pattern when you need to add behaviours with aggregation.

☐ **Factory Method Pattern**

Incorrect

Incorrect. Factories are usually used in cases where you need to instantiate different kinds of objects.

☐ **Singleton Pattern**

Incorrect

Incorrect. Singleton is all about one single object!

☐ **Composite Pattern**

Incorrect

Incorrect. Though Composite pattern could potentially be used to build behaviours in a similar way, another one of these patterns is much more focused on behaviours as opposed to the relationship between objects.

Question 8

You need to connect to a third-party library, but you think it might change later, so you want to keep the connection loosely coupled by having your object call a

consistent interface. Which Design Pattern do you need?

☐ **Proxy**

Incorrect

Incorrect, although the pattern you are looking for is similar!

☐ **Facade**

Incorrect

Incorrect. Although a Facade is often used to mask a complex subsystem, its design goal is usually to provide a simple interface rather than to provide a specific interface.

☐ **Decorator**

Incorrect

Incorrect. The decorator object allows you to build functionality with aggregation! It does not address issues of connecting different interfaces

☐ **Adapter**

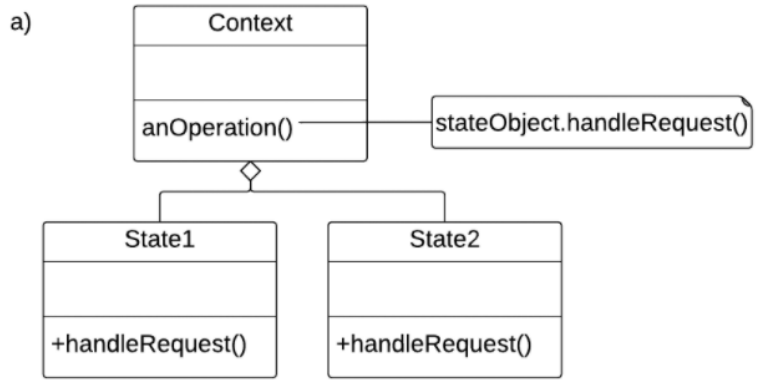
Correct

Correct! The adapter pattern keeps loose coupling between the client and the interface in question. If either changes, only the adaptor needs to be changed.

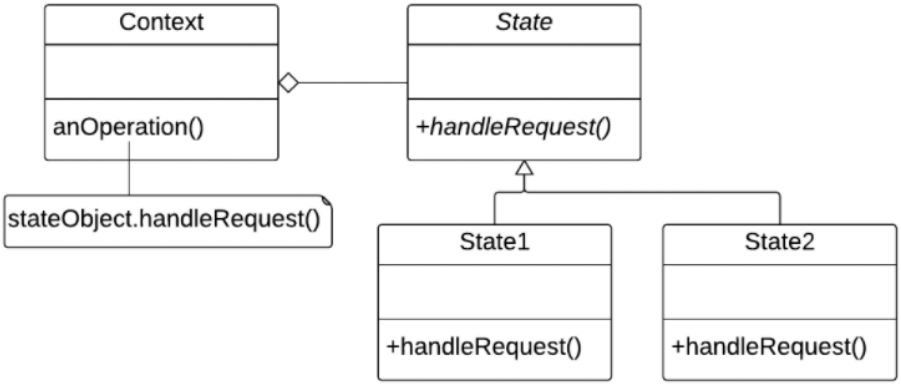
Question 9

Which of these diagrams shows the State pattern?

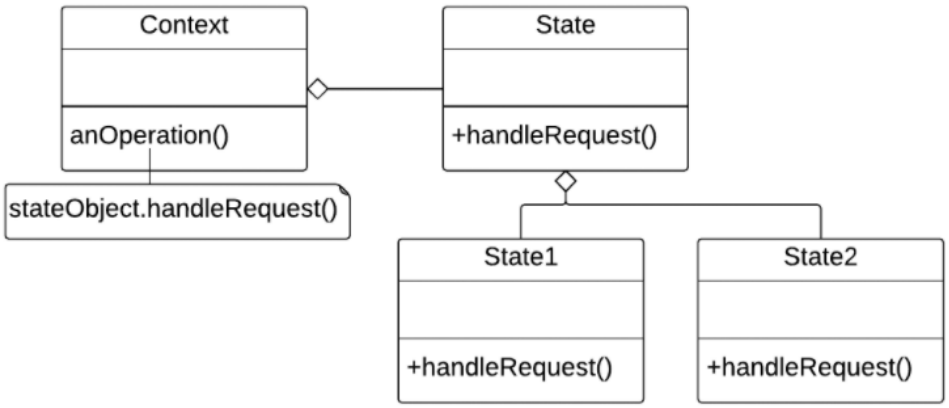
a)


Context
anOperation()
stateObject.handleRequest()
State1
+handleRequest()
State2
+handleRequest()

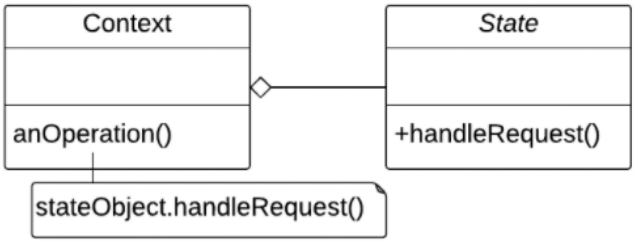
b)


Context
anOperation()
stateObject.handleRequest()
State
+handleRequest()
State1
+handleRequest()
State2
+handleRequest()

c)



d)



☐ **a)**

Incorrect

Incorrect. Without the polymorphism of the state object, you will need to use conditionals in your context to decide how to handle a request, which is not easy to extend.

☐ **b)**

Correct

Correct! The context "has a" state object to determine its state. How requests are handled is determined by the current State object.

☐ **c)**

Incorrect

Incorrect. This diagram does not have the proper relationships between the state classes (State 1 and State 2 should be subclasses of State) and the general State class should be abstract.

☐ **d)**

Incorrect

Incorrect. This doesn't work at all! The State of your context cannot be set using an object.

Question 10

Which of these design principles best describes the Proxy pattern?

☐ **decomposition, because the Proxy object has different concerns than the subject**

Incorrect

Incorrect. Decomposition is when you split a class into two different classes. The Proxy is a stand in for its subject, implementing the same concerns, but maybe in more lightweight ways.

☐ **generalization, because a proxy is a general version of the real subject**

Incorrect

Incorrect. Generalization is not the main drive of the Proxy pattern.

☐ **separation of concerns, because the Proxy object has different concerns from the subject**

Incorrect

Incorrect. The proxy and its subject implement some of the same concerns, but in different ways!

☐ **encapsulation, because the Proxy hides some of the detail of the subject**

Correct

Correct! The Proxy encapsulates some behaviour of the subject in a simpler way, and delegates to the subject when needed.

Question 11

Ashley has a method in her class that needs to makes a request. This request could be handled by one of several handlers. Which design pattern does she need?

☐ **Facade**

Incorrect

Incorrect. This is not the usual use of the Facade pattern.

☐ **Decorator**

Incorrect

Incorrect. The Decorator pattern is usually for adding functionality, so it might be used if it were desirable to handle the request in several ways at once.

☐ **Chain of Responsibility**

Correct

Correct! The Chain of Responsibility is a pattern for passing a request down a line until one of the handlers can handle it.

☐ **Template**

Question 12

Colin is designing a class for managing transactions in software for a banking machine software. Each transaction has many of the same steps, like reading the card, getting a PIN, and returning the card. Other steps are particular to the type of transaction. Which pattern does he need?

☐ **MVC**

Incorrect

Incorrect. A Model-View-Controller is a good candidate for being used somewhere in a banking machine, but it does not fit the specific design issue.

☐ **Template**

Correct

Correct! The Template method is used for situations in which the same general set of steps are followed, but some steps are different in their specifics.

☐ **State**

Incorrect

Incorrect. This is not the usual use of the State pattern.

☐ **Mediator**

Incorrect

Incorrect. The Mediator pattern is used when there are many objects to coordinate.

Question 13

Which of these is **NOT** a good use of the Command pattern?

☐ **Building macros, for example in an image manipulation program**

Incorrect

Incorrect. Sequences of commands are called macros, and the Command pattern is great for supporting such functionality!

☐ **Supporting undo/redo queues of commands**

Incorrect

Incorrect. This is a great use of the command pattern!

☐ **Sending a command to a third-party service or library**

Correct

Correct! This better describes the Facade or Adapter pattern.

☐ **Building a user-interface that can be used to perform operations**

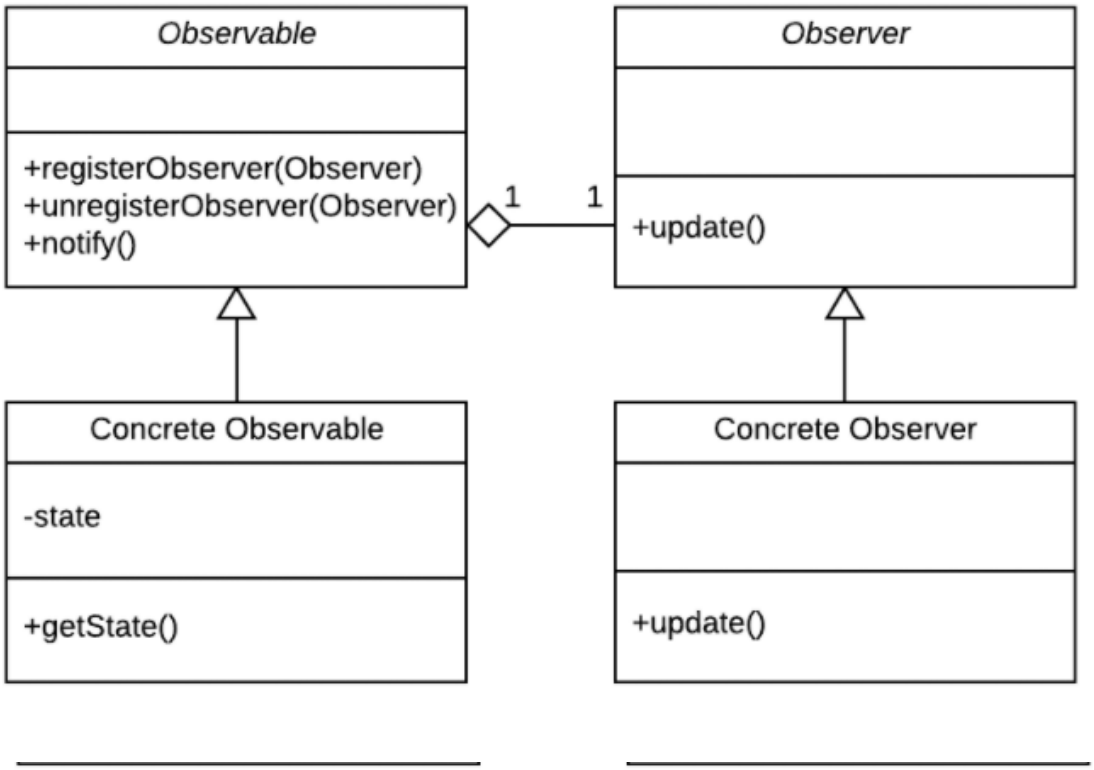
Incorrect

Incorrect. This is a good use of the Command pattern, because it decouples user-interface concerns from the operations themselves.

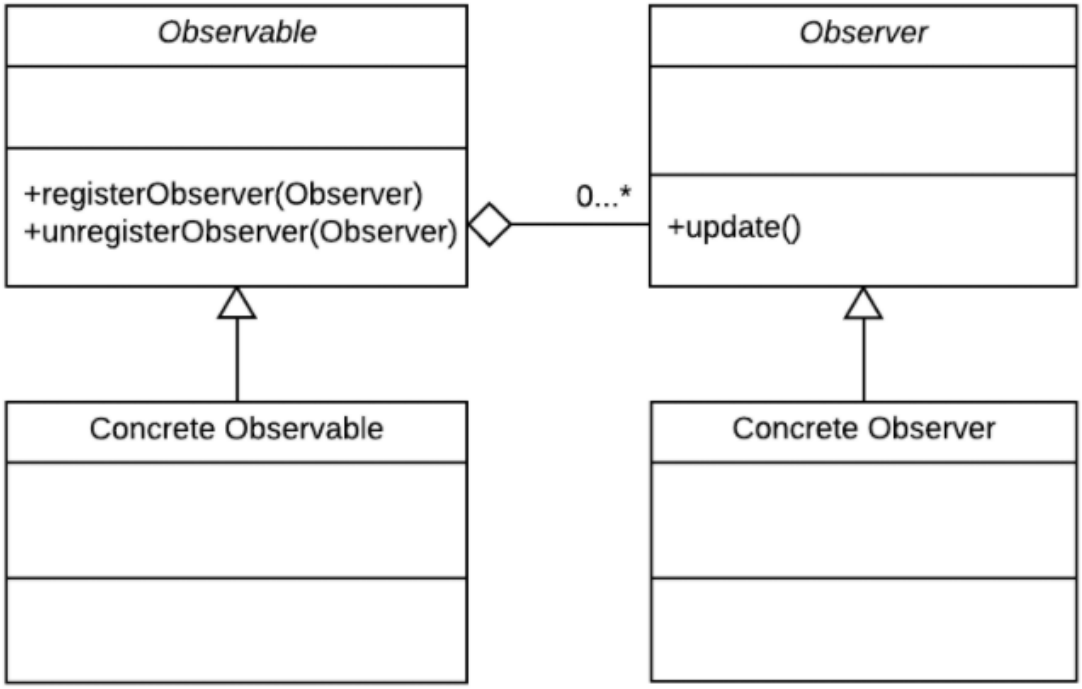
Question 14

Choose the correct UML class diagram representation of the Observer pattern:

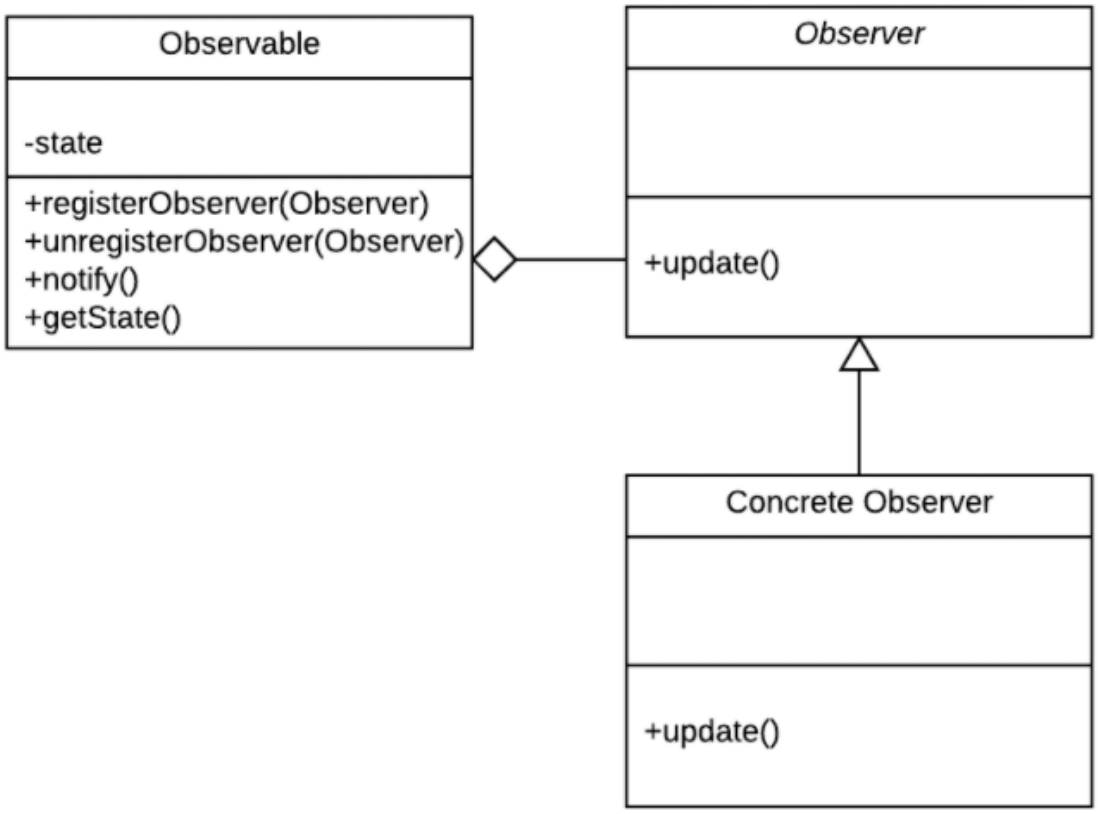
a)


Observable
+registerObserver(Observer)
+unregisterObserver(Observer)
+notify()
1
1
Observer
+update()
Concrete Observable
-state
+getState()
Concrete Observer
+update()

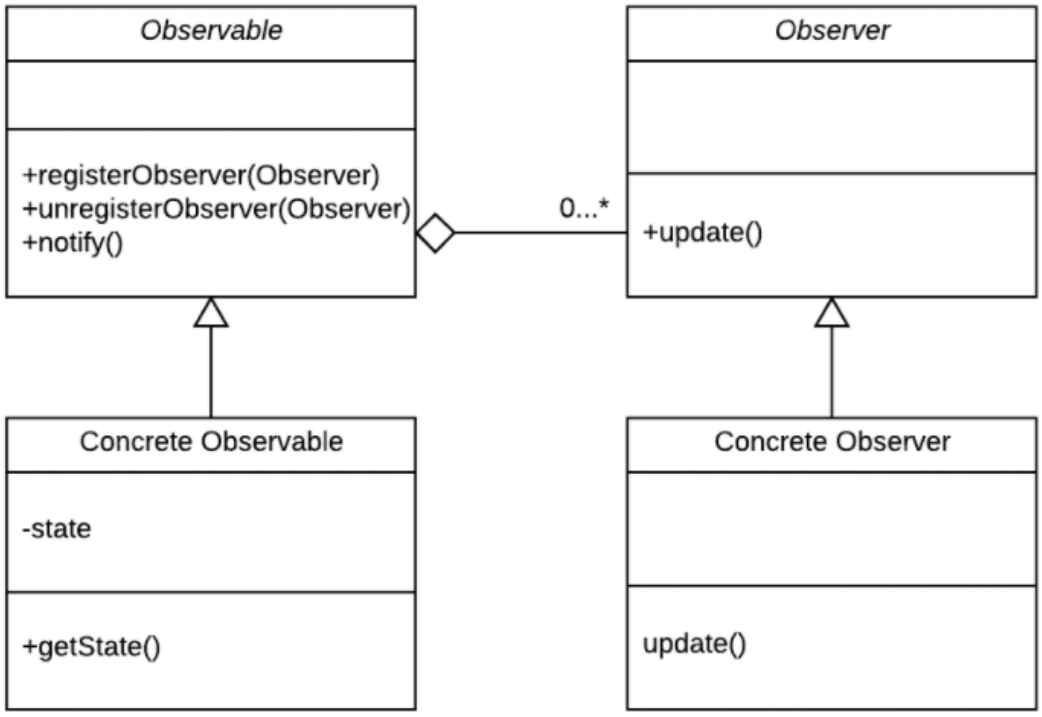
b)


Observable
+registerObserver(Observer)
+unregisterObserver(Observer)
0...*
Observer
+update()
Concrete Observable
Concrete Observer

c)


Observable
-state
+registerObserver(Observer)
+unregisterObserver(Observer)
+notify()
+getState()
Observer
+update()
Concrete Observer
+update()

d)


Observable
+registerObserver(Observer)
+unregisterObserver(Observer)
+notify()
0...*
Observer
+update()
Concrete Observable
-state
+getState()
Concrete Observer
update()

☐ **a)**

Incorrect

Incorrect. There could be many Observers associated with one Observable.

☐ **b)**

Incorrect

Incorrect. The Observable class is missing a method to notify the Observers. How will they know when a change is made?

☐ **c)**

Incorrect

Incorrect. Typical implementation includes subclasses of the Subject or Observable class.

☐ **d)**

Correct

Correct! This diagram has all the correct elements of an Observer pattern.

Question 15

Which code smell may become a problem with the Mediator design pattern?

☐ **Refused Bequest**

Incorrect

Incorrect. The Mediator pattern does not necessarily use inheritance, so Refused Bequest is not a common code smell.

☐ **Speculative Generality**

Incorrect

Incorrect. There's nothing in particular about Mediator that would lead to Speculative Generality.

☐ **Inappropriate Intimacy**

Incorrect

Incorrect. The Mediator object is a control object, and its purpose is to coordinate other objects in order to reduce pairwise interactions.

☐ **Large Class**

Correct

Correct! The Mediator class can quickly become very large, which means it might have this or related code smells, like Divergent Change or Long Method.

Question 16

Hyun-Ji is developing a program. She wants to create a Student class that behaves differently based on if the student has not registered for classes, is partially registered, fully registered, or fully registered and paid. Which design pattern does she need?

☐ **Proxy**

Incorrect

Incorrect. Proxy would involve having a lightweight object to stand in for the student. Not the case here!

☐ **Template Method**

Incorrect

Incorrect. Template method involves a step by step process that is partially implemented by the subclasses.

☐ **State**

Correct

Correct! The State of the student will determine its responses to various requests. Exactly what she needs.

☐ **Mediator**

Incorrect

Incorrect. There is only one entity class being talked about here; a mediator is for coordinating many others!

Question 17

Which of these methods is found in a typical Observer class?

- [ ] **addObserver()**

Incorrect

Incorrect. The Observable keeps track of the Observers, perhaps in a ListArray object.

- [ ] **notify()**

Incorrect

Incorrect. The Observable class notifies its Observers that a change has occurred with this method.

- [ ] **getState()**

Incorrect

Incorrect. Only the Observable knows its state. The Observer does use this method (or one like it) to get the Observable's state.

- [ ] **update()**

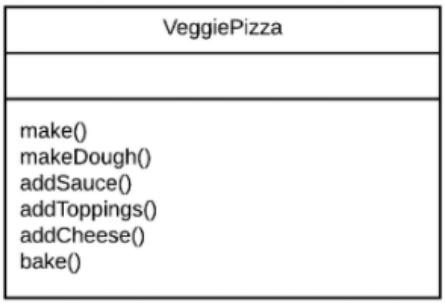
Correct

Correct! The Observer class needs to update itself.

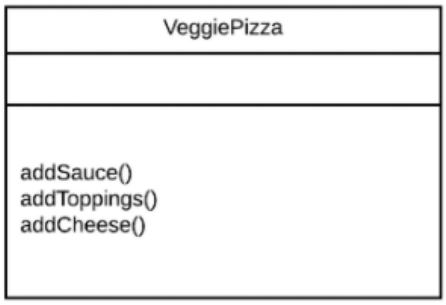
Question 18

Fernando is making pizza objects with the Template Method pattern. The make() function is the whole process of making the pizza. Some steps are the same for every pizza - makeDough(), and bake(). The other steps - addSauce(), addToppings() and addCheese() - vary by the pizza. Which of these subclasses shows the proper way to use a template method?

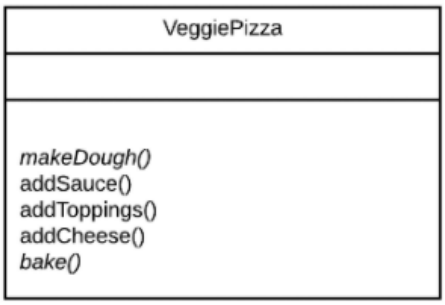
a)



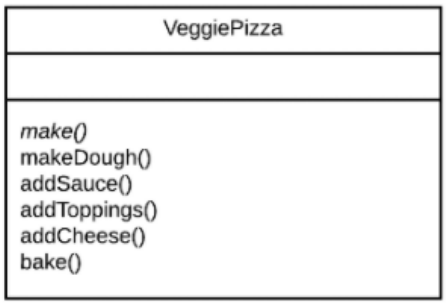
b)



c)



d)



☐ **a)**

Incorrect

Incorrect. The make() process and the common steps should be implemented in the Pizza superclass!

☐ **b)**

Correct

Correct! Only some of the steps are implemented in the subclass.

☐ **c)**

Incorrect

Incorrect. You're getting there, but it doesn't make sense to have abstract methods in your subclass, unless you're planning another subclass

☐ **d)**

Incorrect

Incorrect. You're getting there, but it doesn't make sense to have abstract methods in your subclass, unless you're planning another subclass.

Question 19

In the Mediator Pattern, which pattern is often used to make sure the Mediator always receives the information it needs from its collaborators?

☐ **Template Method**

Incorrect

Incorrect. The Template method is used in many places, but not specifically for this one.

☐ **Chain of Responsibility**

Incorrect

Incorrect. Incorrect. The Chain of Responsibility pattern does not address the issue of informing the Mediator of changes.

☐ **Command**

Incorrect

Incorrect. The Command pattern does not address the issue of informing the Mediator that something has changed.

☐ **Observer**

Correct

Correct! The Mediator can be made an Observer of all of its Collaborators.

Question 20

In the MVC Pattern, which of these is usually made into an Observer?

☐ **Model**

Incorrect

Incorrect. The Model corresponds to the Observable!

☐ **View**

Correct

Correct! Views are usually subscribed to the model so that when changes are made, the views are updated.

☐ **Controller**

Incorrect

Incorrect. Although you may have reason to make the Controller a subscriber, it is not a common feature of this pattern.

☐ **Back-End**

Incorrect

Incorrect. The back-end is the Model!

Question 21

Which of these answers does **NOT** accurately complete the following sentence? "A class is considered closed to modification when..."

☐ **...all the attributes and behaviours are encapsulated**

Incorrect

Incorrect. This is an aspect of being closed to modification. It prevents unintended access.

☐ **...its collaborators are fixed**

Correct

Correct! This is NOT part of being closed to modification. New collaborators may be created that call on this object. Of course, it cannot call on any new collaborators without being modified.

☐ **...it is tested to be functioning properly**

Incorrect

Incorrect. This is an aspect of being closed to change. It reduces the chance that the class will have to be modified to fix a bug.

☐ **...it is proven to be stable within your system**

Incorrect

Incorrect. This is an aspect of being closed to modification. It reduces the chance that the class will have to be changed to fix bugs.

Question 22

How does the Dependency Inversion Principle improve your software systems?

- [ ] **Client classes become dependant on low-level concrete classes, rather than dependant on high-level generalizations**

Incorrect

Incorrect. This is the typical situation, but we would like to improve on it!

- [ ] **Client classes use an adapter to facilitate communication between itself and the rest of the system**

Incorrect

Incorrect. This is not the Dependency Inversion Principle!

- [ ] **Dependency becomes inverted by having the system depend on the client classes**

Incorrect

Incorrect. Think of what depends on what in a typical system, and how it can be improved.

- [ ] **Client classes become dependent on high level generalizations rather than dependant on low level concrete classes**

Correct

Correct! Being dependent on a generalization allows your system to be more flexible.

Question 23

Allison has a search algorithm, and she would like to try a different implementation of it in her software. She tries replacing it everywhere it is used

and this is a huge task! Which design principle could Allison have used to avoid this situation?

☐ **Dependency Inversion**

Correct

Correct! Allison should have made every client of this search algorithm call an interface or an abstract class instead of the concrete search algorithm. That way, when she changed the implementation, the clients would be unaffected.

☐ **Don't Repeat Yourself**

Incorrect

Incorrect. Every client of the search algorithm needs to search for something! How could Allison make this more flexible?

☐ **Composing Objects Principle**

Incorrect

Incorrect. It is not clear how this would have helped her in this case.

☐ **Principle of Least Knowledge**

Incorrect

Incorrect. Allison may have hid away all the implementation details of the search algorithm, and still have the problem here!

Question 24

Which of the code smells is shown in this code example of a method declaration?

```
private void anOperation(String colour, int x, int y, int z, int speed)
```

☐ **Primitive Obsession**

Incorrect

Incorrect. It's impossible to determine from only one method declaration that primitive data types are being overused!

☐ **Message Chains**

Incorrect

Incorrect. Remember that message chains are often nested method calls: A.getB().getC().anOperation()

☐ **Long Method**

Incorrect

Incorrect. This smell has to do with the method itself, not just the signature!

☐ **Large Parameter List**

Correct

Correct! A long parameter list like this is often an indication that you should define an abstract data type to contain this bundle of information.

Question 25

Which object-oriented design principle do Long Message Chains, a code smell, usually violate?

☐ **Cohesion**

Incorrect

Incorrect. Long Message Chains do not necessarily affect cohesion.

☐ **Separation of Concerns**

Incorrect

Incorrect. Long Message Chains do not directly take away from separation of concerns.

☐ **Open/Closed Principle**

Incorrect

Incorrect. Long Message Chains do not directly affect the degree to which your classes are open to extension but closed to change.

☐ **Principle of Least Knowledge / Law of Demeter**

Correct

Correct! A class should only know about a few other classes. Long message chains will make your code rigid and difficult to change.

Question 26

Which code smell can you detect here?

```
public class Person {
  int age;
  int height;
  String hairColour;
  public int getAge() { return age; }
  ...
}
```

☐ **Feature Envy**

Incorrect

Incorrect. This class does not seem to interact with another one.

☐ **Primitive Obsession**

Incorrect

Incorrect. It's impossible to see from one small class if primitives are being overused generally in the software!

☐ **Data Class**

Correct

Correct! This class seems to only contain data and a getter (with presumably more getters and setters). Maybe there are some operations you could move into this class.

☐ **Data Clump**

Incorrect

Incorrect. There is no data clump here: a data clump is a set of data that is often seen together (for example in constructors or method calls) and may be better structured as an abstract class.

Question 27

What are the components of the MVC pattern?

- [ ] **Member, Vision, Controller**
- [ ] **Model, View, Command**
- [ ] **Model, Vision, Command**
- [x] ~~**Model, View, Controller**~~

Question 28

The interface segregation principle encourages you to use which of these object-oriented design principles? Choose the **2 correct** answers.

- [ ] **decomposition**

Correct

Correct! Instead of using inheritance, the Interface Segregation principle encourages you to separate functionality into different interfaces, then combine it to get the behaviour you want.

- [ ] **generalization**

This should not be selected

Incorrect. Interface segregation encourages you to make more, specific interfaces (as opposed to general superclasses)

- [ ] **abstraction**

Correct

Correct! The principle encourages you to select good abstractions for your entity.

- [ ] **encapsulation**

This should not be selected

Incorrect. This principle is all about interfaces, which have no variables or methods to encapsulate.

Question 29

Interface Segregation is a good way to avoid which code smell? **Choose the best possible answer**.

☐ **Switch Statements**

Incorrect

Incorrect. Interface segregation probably won't help you reduce switch statements.

☐ **Long Method**

Incorrect

Incorrect. Since you do not populate a method in the interface, interfaces will not help with this problem.

☐ **Refused Bequest**

Correct

Correct! By composing with interfaces instead of inheriting, you can avoid your classes inheriting behaviour that they will not use.

☐ **Divergent Change**

Incorrect

Incorrect. Divergent change is an issue of the coherence of a class. Segregating interfaces will not help with that!

Question 30

Which of these statements about the Decorator pattern are true?

1. The decorator classes inherit from the basic object which is being decorated
2. Decorator objects can be stacked in different order

☐ **The first statement is true**

Incorrect

Incorrect. This would mean that each decorator inherits the behaviour of the basic object. This is not good separation of concerns!

☐ **The second statement is true**

Correct

Correct! This allows you to build behaviour in different ways. It's also why you must use an interface to build this pattern instead of inheritance, because you do not want to fix the order of objects with inheritance.

☐ **Neither statement is true**

Incorrect

Incorrect. At least one of these statements is true!

☐ **Both statements are true**

Incorrect

Incorrect. At least one of these statements is false!