COIS3040 Final. Winter 2021. 3 hours.

**Part A) Multiple choice questions (2 points):**

1. **The Decorator pattern is useful for:**
   1. **Attaching additional responsibilities to an object at runtime.**
   2. **Bridging two different classes in a system.**
   3. **Separating the construction of a complex object from its representation.**
   4. **Representing the part-whole relationships in a hierarchy of objects.**
   5. **None of the above.**

Answer : a

1. **Which architectural pattern allows a set of equal distributed entities be connected to each other via a common protocol?**
   1. **Client-Server pattern.**
   2. **Multi-tier pattern.**
   3. **Broker pattern.**
   4. **Peer-to-Peer pattern.**
   5. **None of the above.**

Answer : d

**3) In Continuous Integration:**

**a) Modules are tested using Top Down integration.**

**b) Modules are tested using Modified Sandwich integration.**

**c) Each team member integrates at least daily, leading to multiple integrations per day.**

**d) Each module is unit tested only.**

**e) None of the above.**

Answer : c

**4) The Façade pattern:**

1. **Is a behavioural pattern that connects incompatible components.**
2. **Is a creational pattern that allows product families to be created using factories.**
3. **Is a structural pattern that connects incompatible components.**
4. **Is a structural pattern that provides a common interface for a group of related modules.**
5. **None of the above.**

Answer : d

**Part B) Short answers:**

1. **Discuss one difference between the Abstract Factory pattern and the Builder pattern. (2 points)**

Builder Pattern is used for separating complex objects constructions by focusing on creating these complex objects step by step. Builder pattern ultimately has one complex and single product.

Abstract Factory Pattern does no separation but emphasizes a family of product objects which may either be simple or complex. The pattern does not have one builder for just one project because it has many different types of factories to build many products.

**2) Briefly describe what does increase cohesion and reduce coupling mean? (2 points)**

Cohesion is a measure of how strongly the responsibilities of a module are related and it measures the probability that a change will only affect small number of modules. Increase cohesion means if responsibilities of two modules are not same then they should be placed in different modules. This way only components sharing responsibilities are in a same module – the parts of codes related to each other are in the same place.

Coupling is a measure of how strongly the responsibilities of different modules are related and the probability that change to one module will transmit to other modules as well. Reduce coupling means separating unrelated parts of code as much as possible. This way change to one module will not heavily affect other modules.

**3) Think about a weakness of the Client Server solution? Describe how you mitigate it using architectural tactics. (2 points)**

One of the weaknesses of the Client-Server architectural pattern is that there may be performance bottlenecks. That means due to having all information into a few servers and in a few locations, there can be computer or server slow down to crawl. This can happen either due to overload network or inability of one of the components of system to keep up with the pace of the rest of the system.

This is a performance side effect which can be managed by using tactics like increasing resources that will help performance. By increasing resources, we have better efficiency and decrease in latency.

**Part C) Scenarios and code examples:**

**public class InternetCharge{**

**double bandwith;**

**boolean isLimited;**

**double monthlyRate;**

**public InternetCharge(double bandwith, boolean limited, double monthlyRate) {**

**this.bandwith = bandwith;**

**this.isLimited = limited;**

**this.monthlyRate = monthlyRate;**

**}**

**public void setMonthlyRate(int rate) throws Exception {**

**if (rate < 0)**

**throw new Exception("Rate cannot be negative");**

**if (isLimited) {**

**this.monthlyRate = rate \* 1;**

**} else if (!isLimited) {**

**this.monthlyRate = rate \* 1.5;**

**}**

**}**

**public double getMonthlyCharge() {**

**return monthlyRate \* bandwith \* 0.5;**

**}**

**}**

**1) Write two test methods for InternetCharge. The first test method is to test whether the getMonthlyCharge operation is correct. The second test method is to test whether an exception is thrown by the setMonthlyRate method. (2 points)**

**import static org.junit.Assert.\*;**

**import org.junit.Test;**

**public class InternetChargeTest {**

@Test

//testing getMonthlyCharge

//successful test is if method works correctly

public void TestGetMonthlyCharge() throes Exception

{

InternetCharge charge = new InternetCharge(1.0, true, 100);

charge.setMonthlyRate(10);

charge.getMonthlyPayment():

}

//testing setMonthlyPayment

//a successful test is when the method throws an exception

@Test(expected = Exception.class)

//we test if the class throws an exception or not

public void TestSetMonthlyRate(int rate) throws Exception

{

//using default constructor

InternetCharge charge = new InternetCharge(1.0, true, 100);

charge.setMonthlyRate(-20);

}

**}**

**2) Apply the Singleton pattern to InternetCharge using eager instantiation. Discuss the difference between lazy and eager instantiation in Singleton. (2 points)**

public final class InternetChargeSingleton {

private static final InternetChargeSingleton charge = new InternetChargeSingleton ();

private InternetChargeSingleton (){}

public static Singleton getInstance(){

return charge;

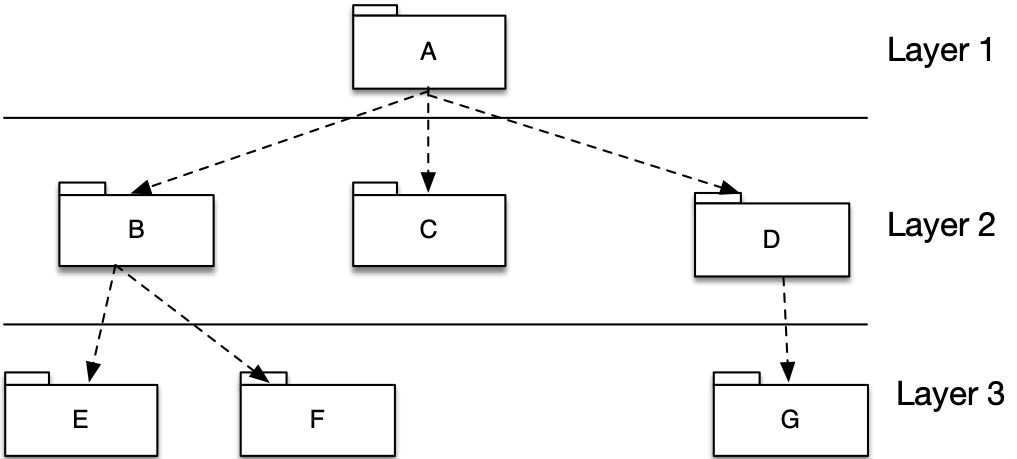
}

}

In eager instantiation, the instance of Singleton Class is created when the class is loading. This means that the instance is created even when the client application may not be using this class.

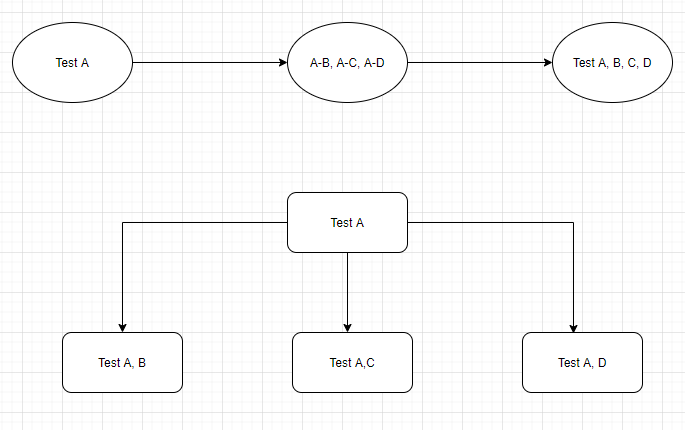
In Lazy instantiation on the other hand, instance is not created until the getInstance method is called. The instance is directly created within the method. It also checks if an instance has already been created, then it does not create an instance and just returns the existing instance.

**D) In the layer architecture below, show how you would do integration testing for layer 1 and 2 using the top-down approach. Then, show how you would do integration testing for layer 2 and 3 using the bottom-up approach. Explain where you should use driver(s) and where you should use stub(s). (2 points)**

****

*Top-Down Testing for Layer 1 and 2*

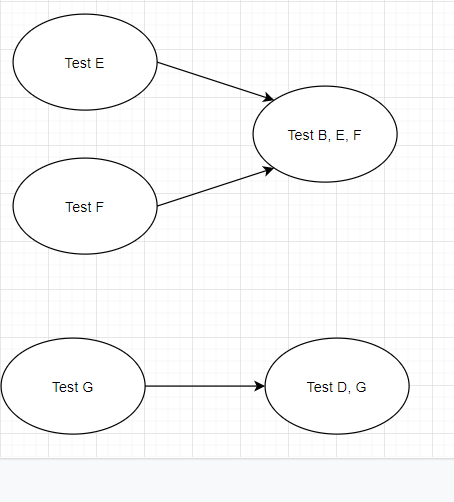
After unit testing subsystem A, the integration test proceeds with double tests A-B, A-C, A-D, followed by the quad test A-B-C-D

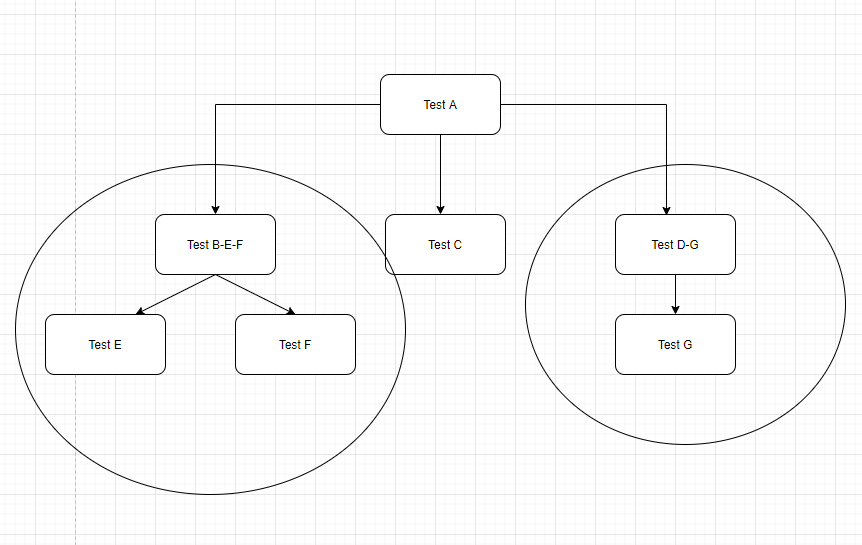


For top-down testing we do not need to use any Drivers. But we need to use Stubs that will allow us to test for all conditions and we may need a large number of stubs. A stub is to be used for partial implementation and returning fake values. Since we are not testing the final bottom layer (Layer 2), we need stubs which will act as temporary replacement and return fake but same output as original system.

*Bottom-Up approach for Layer 2 and 3*

After unit testing subsystems E, F, G, the bottom-up integration test processed with the triple test B-E-F and the double test D-G



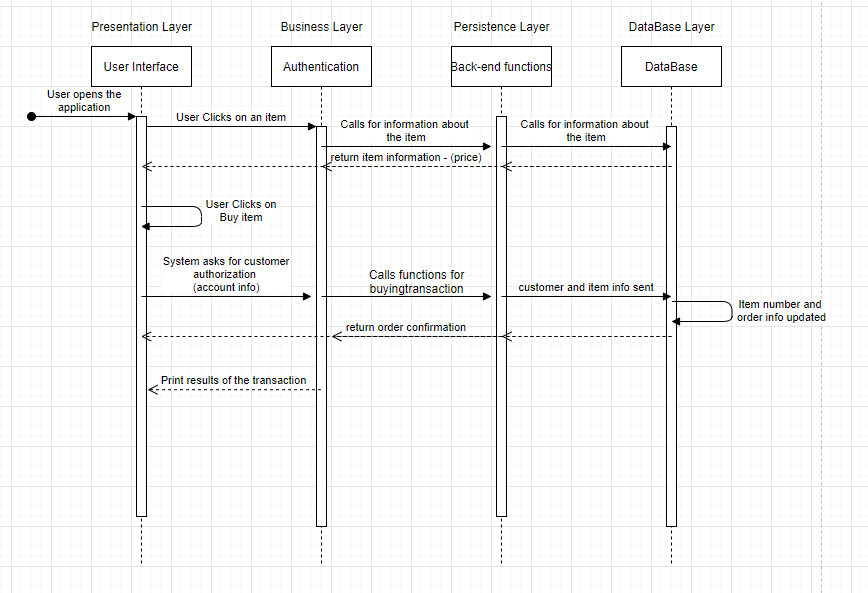


For the bottom-up testing we do not need to use Stubs. But We need to use Drivers. Drivers are needed to control the test cases. Starting from the bottom layer, we need drivers to start these test cases and not stubs. These drivers pass appropriate data and values to the lower levels to get them running for testing.

**E) In an online retail store, the user should be able to manipulate the items’ data by updating their prices, and by buying and selling items. The user should be able to view the prices and the results of buy/sell transactions on the screen.**

**Which architectural pattern suits this application the best? Draw a sequence diagram illustrating a buying transaction. (3 points)**

*Layered Architectural Pattern* will suit the application the best.

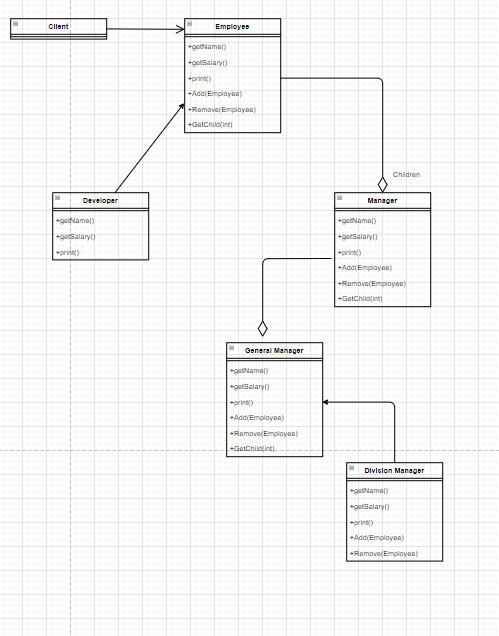


**F) In a software company, there are two types of employees: managers and developers. There is a hierarchy of managers, for example, a division manager works under a general manager.**

**Which design pattern can be used to model this? Illustrate your answer by drawing a class diagram. (3 points)**

When we want to represent part-whole hierarchies of a system, the most suitable design pattern to use is the *Composite Design Pattern*.

(The arrows with black head are actually open arrows, I could not find the one in the software I was using)



**G) One of architectural tactics for Interoperability is to manage interfaces, which include orchestration and tailoring interfaces. What do they mean? Which design pattern can help with orchestration and which one can help with tailoring interfaces? (2 points)**

Interoperability defines the degree to which two or more systems can exchange useful information between them. One of the tactics is managing interface which includes Orchestrate and Tailor Interface.

Orchestrate uses a control mechanism to coordinate, manage and sequence the invocation of services. This method is used when the systems have to interact in a complex fashion for completing a complex task. For example, workflow engines – they need orchestrate to manage smooth functioning and flow of data.

The design pattern that can be used for Orchestrate method is Mediator Pattern.

Tailor Interface on the other hand is the method to add or remove capabilities to an interface. This includes process like translation, buffering or data-smoothing, where data can be added. Example where capabilities can be removed is when we want to hide certain functions for security reasons.

The design pattern used for Tailor interface is decorator pattern.

**H) Suppose that there are 4 classes in a system: A, B, C, D. Changing class A requires changing class B. Changing class C requires changing class D. (3 points)**

1. **Is there any problem with this design setup? If yes, what is the problem? How would you solve it using an architectural tactic?**

Yes, there is no problem with this design setup. Changing Class A first requires us to change Class B and similarly changing Class C requires changing Class D. This is high coupling. Unless changing one class has small impact on the other class i.e. low coupling, this is a problem. Because every change in A and C will require us to change B and D as well.

One point to note is that change in one module is less costly and easier, but if ever both A and C are changed, then two modules are affected and hence I consider this as a problem.

Tactic for fixing this is Encapsulation. It will introduce a new interface to the module and the probability that a change to one module propagates other modules will decrease.

1. **Which design pattern can be used to create a common interface for A, B, C, D?**

Façade Design Pattern can be used to crease a common unified interface to a set of classes in a subsystem, in this case – A, B, C, D. This will be the higher level interface which will make the subsystem easier to use.