Software Engineering

Unit 1 Questions

Team 29

Sudipta Halder (2021202011)

Anjaneyulu Bairi (2021202008)

Sowmya Lahari Vajrala (2021202010)

Josh Elias Joy (2021204009)

Q1. Does pair programming reduce the need for refactoring? Support your argument with appropriate evidence.

Ans:

Yes pair programming would reduce the need for refactoring given that each member is able to work with each other in an effective and collaborative manner. Pair programming involves two programmers:

-**A driver** who codes and focuses on the immediate problem at hand.

-**A navigator** who observes the code of the driver, looks out for errors and has a look at the overall design.

Each person focuses on a different perspective, which is not possible by a single person. Having two minds working on a project can foster a challenging scenario where each person is forced to keep up with the other and this could bring out the best in each other to come up with well-crafted solutions.

The navigator will continuously review code which means that more errors are found while coding and since the navigator has an opportunity to see the bigger picture rather than a single line of code, insights and suggestions could be proposed to improve the overall structure of the code or a better way to implement the solution.

Each person will have different experiences, varied knowledge and have faced different challenges. This inherently means that both people could converge to a single solution that would be better than one proposed by a single programmer.

A single programmer is generally focused on solving the immediate problem as fast as possible, this might lead to a solution that would be similar to a quick fix or a hack. It would require the same person to switch context and analyze the whole code to come with a solution that is well designed, this would be a cognitively demanding task. Thus having a Navigator to observe the overall design and keeps checks on the drivers actions will be a boon.

Working as a team and solving problems together can foster friendships and long term bonds, this can make mundane work more enjoyable and thus lead to better code.

Q2. In "No Silver Bullet -- Essence and Accidents of Software Engineering" by Fred Brooks, the author claims "there is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity". List arguments to support/contradict his claim.

Ans:

Presently I would partially agree with the statement "there is no silver bullet to improve productivity, reliability and simplicity". But in the near future this might change as there have been advancements in AI technology

Fredwork talks about accidental and essential complexity. Accidental complexity arises from the programmers and their implementations, these can be solved by incorporating the correct design choices. Fredwork has suggested methods to tackle accidental complexity.

Essential complexity arises from the fact that certain software problems are complex and cannot be simplified by using different strategies. Fredwork has suggested potential silver bullets to tackle essential complexity (including artificial intelligence) but ultimately dismissed them as they do not solve the complex problem, rather they provide incremental improvements to facilitate solving these problems in an easier way.

"No Silver Bullet—Essence and Accident in Software Engineering" was published back in 1986 and a lot of advancements have taken place. I do feel in the near future AI technology would be better equipped to handle essential complexity. Here are some examples which are currently used to improve productivity, reliability and simplicity

-IDE's have become much more advanced and have features to complete and generate pieces of code unlike before. For example, PyCharm is an intelligent Python IDE. It provides features such as automated code refactoring, code inspections, code completion etc.

-DeepCode is an AI software platform which learns from open-source programmers and uses the acquired knowledge to make suggestions on how the code can be improved. Developers can use deepcode to review code and inform them of vulnerabilities.

Q3. The C++ implementation of class Adapter specifies the use of multiple inheritance. Using a diagram show how this could be implemented in Java? Be specific about the Java features that you are using.

Ans: C++ implementation of class Adapter uses multiple inheritance. An example is given below:

```cpp
#include<iostream>
#include<string>

using namespace std;

class IPlayer //Client wants to use this.
{
public:
    virtual void Play() = 0;    //This is function client wants to use
};

class OldPlayer //What we have
{
    string _song;
public:
    OldPlayer(string song) :_song(song){}

    void OldPlay(int volume)    //This is function we got
    {
        cout << "\nPlaying song : " << _song << " at volume :" << volume << "\n";
    }
};

class Adapter : public IPlayer, private OldPlayer
{
    int _volume;
public:
    Adapter(string song, int volume) :_volume(volume), OldPlayer(song){}

    void Play()     //Comes from public inheritance
    {
        OldPlay(_volume);   //Comes from private inheritance
    }
};

int main()
{
    IPlayer* newStuffDad = new Adapter("Good Ole Boy Like Me", 50);
    newStuffDad->Play();    //The call to the interface function is routed by the adapter to the LegacyCode.
    cout << "\n";
        delete newStuffDad;
    return 1;
}
```

- **Line 07:** The wicked new interface everyone wants to use.
- **Line 10:** The Play method does not need anything.
- **Line 13:** The good old class which was fine in times before.

- **Line 17:** The OldPlayer class cTor wants the song name.
- **Line 19:** The OldPlay method which actually wants what volume to play the song at.
- **Line 25:** The adapter class doing the unthinkable. Multiple inheritance.
- **Line 29:** The cTor of this class takes in the volume as well as the song to be played. Passes the song name to the cTor of the OldPlayer class. And stores the volume at which the song will be played.
- **Line 31:** The Play method. See the cheeky thing happening in that method. The OldPlay method gets called with the volume parameter.
- **Line 39:** The hot new player for new generation. See that the song name as well as the volume at which it has to be played is passed on.
- **Line 40:** And the Play method which is not taking anything. And the new generation is happy. Whatever man.

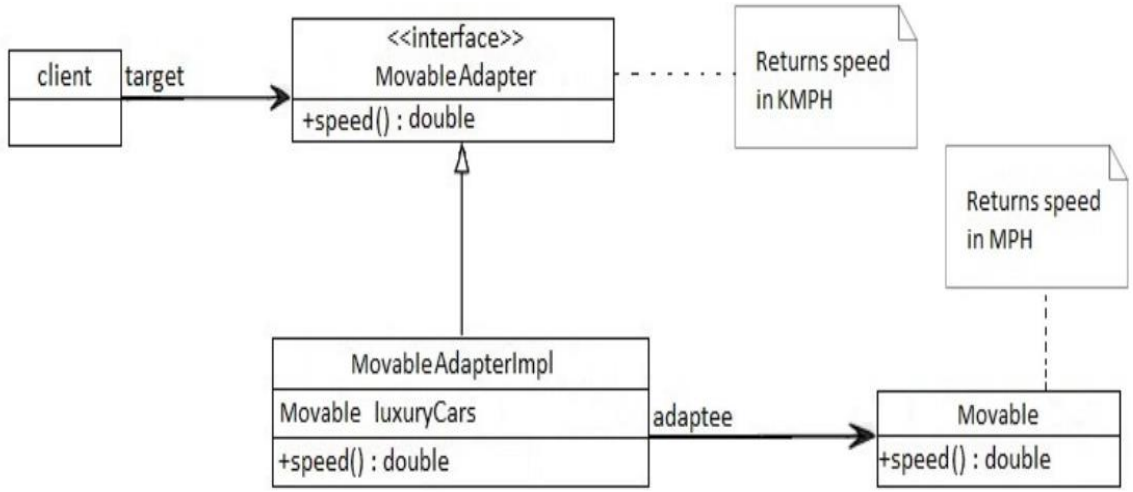
But, in Java multiple inheritance is not allowed. So, we need to make a different approach to implement the adapter pattern. The following features are used for the implementation of Adapter:

- Abstraction: By using interfaces.
- Inheritance: The adapter class inherits from the interface that the existing codebase conforms to.
- Composition: The adapter class composes of the existing service class that it tries to adapt to the expected interface.

Example in Java:

Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH). Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).

To deal with this problem, we'll create an adapter which will convert the values and give us the desired results:



First, we'll create the original interface *Movable* which is supposed to return the speed of some luxury cars in miles per hour:

```
public interface Movable {
    // returns speed in MPH
    double getSpeed();
}
```

We'll now create one concrete implementation of this interface:

```
public class BugattiVeyron implements Movable {

    @Override
    public double getSpeed() {
        return 268;
    }
}
```

Now we'll create an adapter interface MovableAdapter that will be based on the same Movable class. It may be slightly modified to yield different results in different scenarios:

```
public interface MovableAdapter {
    // returns speed in KM/H
    double getSpeed();
}
```

The implementation of this interface will consist of private method *convertMPHtoKMPH()* that will be used for the conversion:

```java
public class MovableAdapterImpl implements MovableAdapter
{
    private Movable luxuryCars;

    // standard constructors

    public MovableAdapterImpl(BugattiVeyron carObj)
    {
        luxuryCars = carObj;
    }

    @Override
    public double getSpeed()
    {
        return convertMPHtoKMPH(luxuryCars.getSpeed());
    }

    private double convertMPHtoKMPH(double mph)
    {
        return mph * 1.60934;
    }
}
```

Now we'll only use the methods defined in our Adapter, and we'll get the converted speeds.

```java
public class Main
{
    public static void main(String[] args)
    {
        BugattiVeyron bugatiCar = new BugattiVeyron();
        System.out.println("Speed of car in MPH: " + bugatiCar.getSpeed());
        System.out.println("Now, we would pass the car obj inside the adapter to get the speed in KPH");
        MovableAdapterImpl adapterObj = new MovableAdapterImpl(bugatiCar);
        System.out.println("Speed of car in KPH: " + adapterObj.getSpeed());
    }

}
```

The output is as follows:

```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe  (05-Feb-2022, 1:29:31 am – 1:29:31 am)
Speed of car in MPH: 268.0
Now, we would pass the car obj inside the adapter to get the speed in KPH
Speed of car in KPH: 431.30312
```

Q4. One method of classifying iterators does so along two dimensions. The first indicates the location of control of the iteration (internal to the iterator or external client control), and the second which indicates the location of the definition of the iteration logic (embedded as part of the collection objects or in objects separate from the collections). Considering each dimension separately, what are the positive and/or negative aspects of iterators of each type?

Ans: Iterator is a behavioural pattern or an interface in collection framework used to traverse elements one by one. The underlying collection object representation will not be revealed while using iterator. Collection object can be tree, hash, graph, complex data structures, array list etc. There are three types of iterators in java collections which are Enumerations, Iterator and List Iterator.

Enumeration: It is an interface that was the first iterator in Java, and it can only accept collections like vectors or hash tables, both of which are inheritance collections. Enumerator can only be used for forward navigation, and it cannot be used to delete elements.
Iterator: This is a more complex version of enumeration that is intended to accommodate all sorts of collections; however, it only allows forward traversal.

List Iterator: This type can be used with list types like array lists, double linked lists, linked lists, and so on. It is capable of bidirectional processing of components. Iterators are divided into two categories: iteration control location and iteration location.

- Location of Iteration control:
    - Control is external to iterator – Iteration client has control.
        - Each object should be processed by the client, who should drive the iterator. List comparisons, for example, have more flexibility than internal iterators.
    - Control is internal to iterator – Iterator has control.
        - Iterator is given an operation to apply each element. It's simple to use because logic is specified solely by the internal iterator.
- Location of Iterator:
    - Iterator is embedded in collection object.
        - Advantage: Its intact encapsulation of collection object.
    - Iterator is separate class from collection class.
        - Advantages:
            - It is feasible to decouple collection structure and traversal without fear.
            - Only one of the traversals such as forward, backward, and bidirectional can be employed.
            - At the same time, we can do numerous traversals.

Q5. What language constructs would you use to give iterators privileged access in Java and in C++? How will your answer depend on the classifications for iterators given above?

Ans:

Iterator pattern is used to separate data structures and algorithms that are applied on them. This iterates different collections in a generic manner without exposing the internal details of a particular collection. In C++, this is done using pointers and in Java Iterator interface is used.

C++:

In C++ privileged access to iterators can be provided using friend functions. This will allow the iterator to have access to the collection. But the problem arises while defining new traversals. A new friend function needs to be added for every such new traversal.

This can be avoided by making the iterator classes extend the collection classes using protected access specifier. This will allow the iterator to access the data members that are nor publicly available.

In case of Internal iterators, the above methods can be used but the external iterators cannot have privileged access over the collections. The other dimension of classification is the location of iteration logic. If the logic is embedded as a part of the collection objects, then the iterator will by default has access to all the data of the collection. If the logic is placed in a separate object, then the class to which the object belongs to should extend the Collection class.

Java:

In Java, to give privileged access, the iterator object should be a derived class of the Collection object. Also, if the iterator is defined in the same module as the collection class, then the iterator will have privileged access. In case of Internal iterators, the iterator should be a child class of the container class, but the external iterators cannot have privileged access over the collections.

If the logic of the iterator is embedded as a part of the collection objects, then the iterator will by default has access to all the data of the collection. If the logic is placed in a separate object, then the class to which the object belongs to should extend the Collection class.

Q6. Draw sequence diagrams for the registration and update cycles of an Observer pattern implemented using appropriate Java classes. Be sure to label the object with the Java class and its role in the Observer pattern using Stereotypes.

Ans:

Observer pattern:

Observer design pattern is used when different objects need to be notified of the occurrence of a specific event with respect to an object that they are observing.

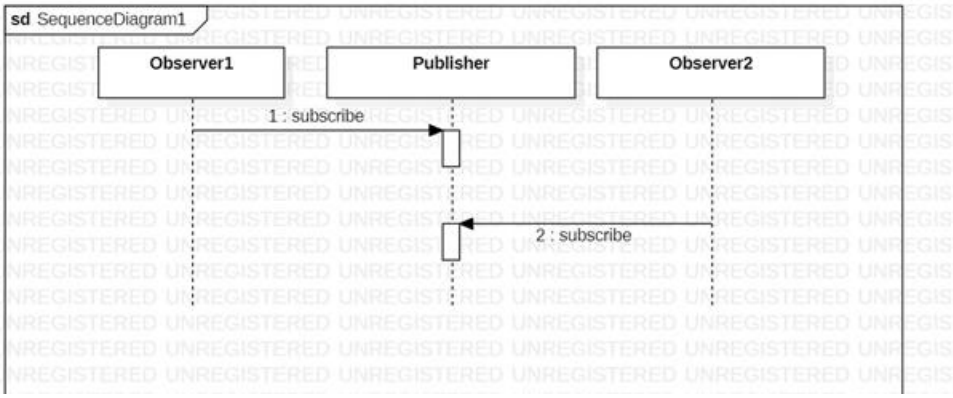
One common approach to deal with such cases is to make the object the object that is being observed to directly notify all the objects that are observing it. The problem arises in the case when a new observer must be added. This requires a change in the code of the object that is being observed. This approach also unnecessarily exposes the private data of the observers.

Observer pattern resolves those issues following a subscriber mechanism. In this pattern the object that is being observed is called subject or publisher and the objects that are observing the subject are called subscribers. The publisher class provides a mechanism for the subscribers to subscribe to the state change of the observer. The subscribers can unsubscribe if required. All the subscribers implement a common subscriber interface which has an update method.

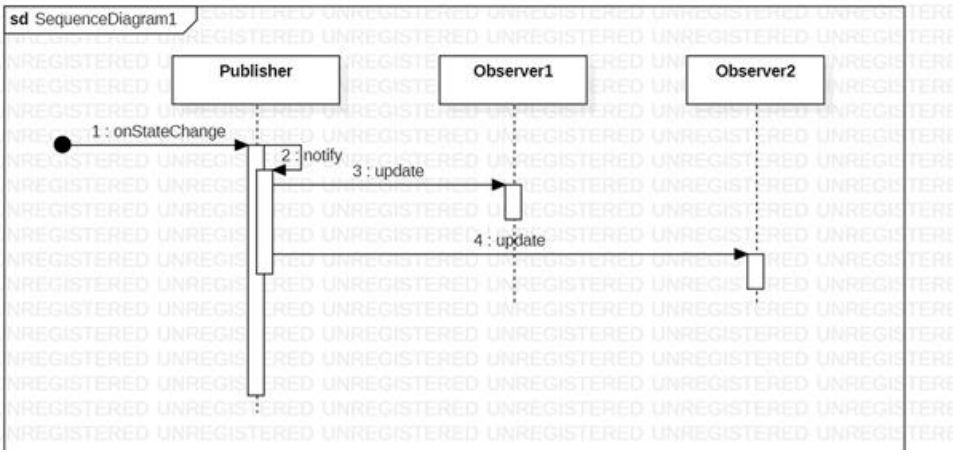
The publisher implements this as follows:

- The publisher maintains a list of subscribers.
- The publisher provides methods to subscribe/unsubscribe.
- Whenever an object subscribes to it, that object is added to the list of subscribers.
- Whenever an object unsubscribes to it, that object is removed from the list of subscribers.
- Whenever a state change/event occurs, the publisher notifies the same to all the subscribers present in the subscribers list.

Registration Cycle:



Update Cycle:



The corresponding Java class for Publisher is java.util.Observable.

The corresponding Java Interface for Observer is java.util.Observer.

Q7. You have seen that it is quite common to use the Observer pattern within GUI frameworks, such as Java's Swing framework. Why do you need to be concerned about how long it takes to process a notification by a GUI element? Should this be the application designer's concern or be dealt with by the framework itself? How can the concern be eliminated? why do we need to be concerned about processing time of a notification?

Ans:

Why do we need to be concerned about how long it takes to process a notification by a GUI element?

When a state change happened for any GUI element, all entities which are referring to the changed GUI element must get information about this change and their state should be updated according to the change that happened.

This process should be time bounded so we can get consistency between GUI elements.

Who should take care of this?

Framework should take care of this process because the changes made in one GUI element should be processed and notified to all other relatable entities.

How to eliminate this concern?

Here we can use Observer pattern.

Observer is a behavioural design pattern. It specifies communication between objects: observable and observers. An observable is an object which notifies observers about the changes in its state. This pattern sends a broadcast communication and entities which are subscribed to that observable will receive notification about changes automatically. By this pattern we can avoid sending individual notifications to entities.

Q8. Argue both for and against including the functions to handle the Composite's children in the Component interface. What are the implications for implementation of the pattern and on the intention of the pattern?

Ans:

Let us say we have 3 functions in our interface. Operation(), Add(), Remove().

Add() and Remove() are handlers here.

Including handlers in interface:

It provides transparency by allowing all components to be treated equally. Clients can try to do useless things to leaf components during run-time, therefore it is not safe.

Intentionally it is easy to generalize one common interface for Leaf and composite classes. But while implementing we need to take care about Add() and Remove() in Leaf, this will lead to error checking in Leaf because there is no implementation of those functions.

Including handlers in class:

When we include handlers in class instead of interface, there is no need of error checking.

Implementation wise it is good because any attempt to perform a child operation at compilation time on a leaf component is caught, ensuring safety. However, because the leaf and composite components now have different interfaces, we lose transparency.