Software Maintenance

Rishi Parmar

January 10, 2017

Contents

1 Lecture 1 - Introduction

First of all, I have no idea why we have an exam for this module. I don't even know what the exam is going to be on so I'm just going to go through some lectures and hope for the best. Anyway, 'software Maintenance is changing software after it has been delivered and is in use'. The majority of software maintenance work includes:

- Fixing coding errors
- Fixing design problems
- Adding additional requirements

Software maintenance can be put into **three** categories:

1. **Corrective Maintenance** $\longrightarrow$ Basically fixing bugs
2. **Adaptive Maintenance** $\longrightarrow$ Adapting the software due to environmental changes, such as laws and updated requirements from the business
3. **Perfective/Performance Maintenance** $\longrightarrow$ Improving the performance of the software, this doesn't change functionality

They mention some shit about how *maintenance* is 'preserving software in a working state' whilst *evolution* refers to improving the software. They talk about how shit code is when dealing with large software and basically a pain in the arse and WHY it is a pain in the arse. The reasons are pretty simple e.g messy and bad commenting. They then give a reminder of some Object Orientation concepts which we need to know/understand. These include:

- **Abstraction** $\longrightarrow$ When you only concentrate on the essential characteristics of the software. Basically removing the need to deal with BS
- **Inheritance** $\longrightarrow$ When one object acquires the properties of another which allows for ez object relationships
- **Encapsulation** $\longrightarrow$ Hiding internal implementation and requiring that user interaction can only be performed via an object's methods
- **Modularity** $\longrightarrow$ When source code for an object can be written/edited independently of the source code for other objects
- **Polymorphism** $\longrightarrow$ When classes can have different implementations of the same methods

For more information on Object Orientation and its concepts, check out my PG-13 notes on the topic.

They list the essentials of software maintenance in a list as follows:

- Understanding the client
- Understanding the code
- Refactoring the code
- Extending the code
- Working as a team
- Managing client expectations
- Managing maintenance process

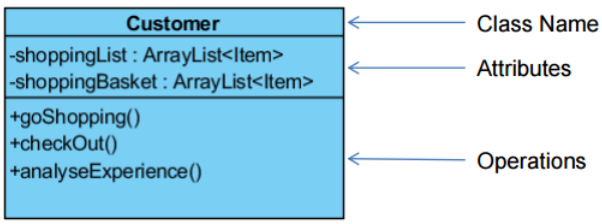
1.1 Understanding the code

They express that with large amounts of code, it is important to understand the structure of the code. Yep, you guessed it, that means class diagrams and other shitty visualisation techniques. I'll give a quick recap of all that garbage.

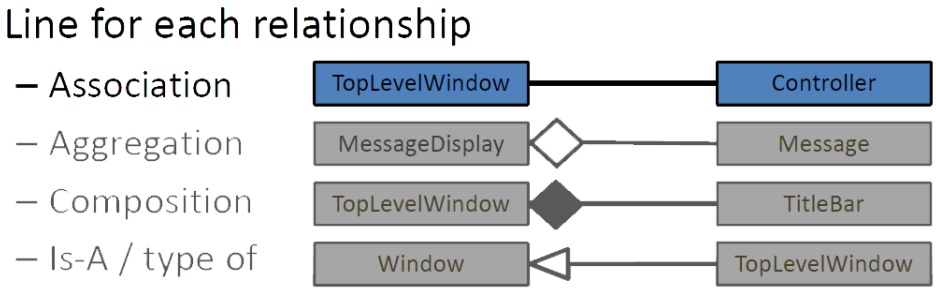
1.1.1 Class Diagrams Review

Classes are blueprints for objects in a software. They contain data and perform operations. Class diagrams represent these blueprints. They are said to address a 'static design view' of a system because they document the main structure of the software. This differs to behavioural diagrams such as use case and activity diagrams which document the dynamic aspects such as the methods and collaborations.
Class diagrams contain three rows, and arrows to represent relationships. The categories are:

1. Class Name
2. Attributes $\longrightarrow$ the variables and arrays etc.
3. Operations $\longrightarrow$ the functions/methods



Class diagrams are good because they provide a simple summary of the classes and relationships. But for larger projects, they can be a pain in the arse. Here's a reminder for the relationships (they don't give this on the slides so you can thank me later or by me a drink).



The speak a bit about how important testing is. They express the importance of testing. As for types of testing, they can be broken down into:

1.2 Testing Introduced

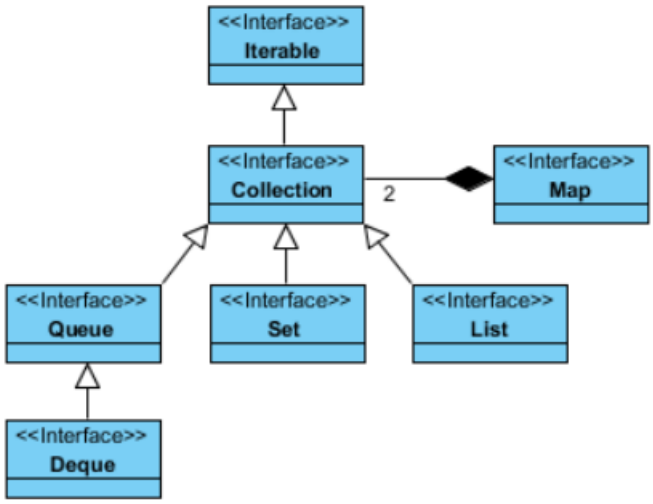
- **Regression Testing** $\longrightarrow$ This is when after you do a bug fix or change, you re-test to make sure that all of the old functionality is still there, and that the bug is fixed
- **Unit Testing** $\longrightarrow$ Automated tests to test the internal workings of the methods. The test is usually made to be small and as isolated as possible and so usually doesn't rely on other resources in the software

2 Lecture 2 - More Advanced Java Programming

In this lecture they discuss the 'Java Collections Framework' and link it to some OO concepts.

2.1 Java Collections Framework

The best way to think about JCF is as a library. It is a collection of classes and interfaces that implement commonly reusable collection data structures. This includes things like ArrayLists. The framework not only contains data structures, but also algorithms e.g searching and sorting. On the lecture they describe the framework as 'Container objects that contain objects'. The interface layout for JCF is as follows:



- **Collection** $\longrightarrow$ Something that holds a dynamic collection of classes
- **Map** $\longrightarrow$ Defines mapping between keys and objects (two collections). The important thing to remember about the Map interface is that there are unique keys for each value. This interface is used in some data structures such as HashMaps, which will be explained later
- **Iterable** $\longrightarrow$ Basically a pointer to the collections content. You an move pointer to one of the objects within the collection for use

Most of these interfaces can be found in the java.util.* package, whilst the 'Iterable' interface can be found in the java.lang.* package (probably don't need to know this). Here are some example implementations you may be familiar with:

- Classes that implement the collection interfaces typically have names in the form of <Implementation-Style><Interface>

	Implementations				
Interfaces	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

2.2 Implementations

2.2.1 ArrayList

For the next part of the lecture they show code examples of the ArrayList, TreeSet and HashMap classes. ArrayLists are probably the most important so here's an example of that in use:

```
import java.util.*;

public class ArrayListExample {
   public static void main(String args[]) {
      /*Creation of ArrayList: I'm going to add String
       *elements so I made it of string type */
         ArrayList<String> obj = new
            ArrayList<String>();

         /*This is how elements should be added to
            the array list*/
         obj.add("Javascript is shit");
         obj.add("Coffeescript is shit");
         obj.add("Low level is best level");
         obj.add("Pedro noob");

         /* Displaying array list elements */
         System.out.println("Currently the array
            list has following elements:"+obj);

         /*Add element at the given index*/
         obj.add(0, "U wot m8");
         obj.add(1, "sowwy m8");

         /*Remove elements from array list like
            this*/
         obj.remove("Pedro noob");
         obj.remove("Coffeescript is shit");

         System.out.println("Current array list
            is:"+obj);

         obj.remove(1); //Remove element from the
            given index

         System.out.println("Current array list
            is:"+obj);
   }
}
```

Note that in the ArrayList declaration you have to give the type. This is because Java is too shit to understand it if you try to use a constructor instead.

2.2.2 HashMaps

The example they used for HashMaps is decent so I'll include that for reference:

```java
import java.util.HashMap;
import java.util.Set;

public class mainClass {
    public static void main(String[] args) {
        HashMap<String, String> hm = new HashMap<String, String>();
        hm.put("first", "FIRST INSERTED");
        hm.put("second", "SECOND INSERTED");
        hm.put("second","SECOND INSERTED");
        System.out.println(hm);
        Set<String> keys = hm.keySet();
        for(String key: keys){
            System.out.println("Value of "+key+" is: "+hm.get(key));
        }
    }
}
```

```
<terminated> mainClass (1) [Java Application] C:\Program Files\Jav
{first=FIRST INSERTED, second=SECOND INSERTED}
Value of first is: FIRST INSERTED
Value of second is: SECOND INSERTED
```

An ez way of thinking of HashMaps is basically an array with a key to each element. The HashMap class implements the Map interface.

2.3 Object Orientation Important Concepts

2.3.1 Aggregation and Composition

- **Aggregation** $\longrightarrow$ The child object can exist independently of the parent. The child exists and is created *outside* of the parent.
- **Composition** $\longrightarrow$ The child object only exists *inside* of the parent object/class. If the parent were to be destroyed, it would indeed be GG for the child also.

They give examples of these concepts in the lecture notes but they're long so I wont include them.

2.3.2 Inheritance

Inheritance: Forming new classes based on existing ones. The parent class being extended is known as the **Superclass**. Whilst the child class that is inheriting this bahaviour is known as the **Subclass**. The Subclass gets a copy of every field and method from the super class (variables/functions). They are said to have an 'is-a' relationship, since each Subclass object is an object of the

superclass and can be treated as one. A subclass can call its parent's constructor or methods with super().

2.3.3 Polymorphism

Two forms of polymorphism are 'Overloading' and 'Overriding'.

- **Method Overloading** $\longrightarrow$ When there are multiple methods with the same name in a class. An example is constructor overloading. This is when you have multiple constructors which differ in their arguments. This is resolved at compile time and is known as *static binding* (more on that soon).
- **Method Overriding** $\longrightarrow$ When there are methods with the same name declared in the super and the sub class. In this case the method would literally override the other of the same name. This is resolved during runtime which is known as *dynamic binding*.

Static binding occurs when the compiler knows that there is no way that the method can be overridden. Dynamic binding occurs during overriding, when the compiler can't decide which method to call. Note that if a method is private, static or final they will always be static binding since they can not be overridden. Here is some example code for method overriding:

```
class Human{
   public void walk()
   {
        System.out.println("Human walks");
   }
}
class Boy extends Human{
   public void walk(){
        System.out.println("Boy walks");
   }
   public static void main( String args[]) {
        //Reference is of parent class
        Human myobj = new Boy();
        myobj.walk();
   }
}
```

In this case, the output would be: Boy walks.

2.3.4 Abstract classes

An abstract method is one without a body. As a result, abstract methods must come from classes that inherit other classes.If a class contains an abstract method, the class must too be declared as abstract. Abstraction is used to remove unnecessary implementation details and to focus on functionality.

2.3.5 Interfaces

An **Interface**, in its most common form, is a group of related methods with empty bodies. This makes it similar to an abstract class. An example of an interface class is:

```
interface Bicycle {

    //  wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

If you want to implement this interface, you must use the *implements* key word. Although interfaces are similar to abstract classes, they have several differences:

- An interface isn't actually a class
- Since abstract classes use inheritance, the relationships between the sub-/super classes are usually stronger than many interface possibilities
- An abstract class can contain instance variables and methods with bodies, but an interface can not
- In an interface, all methods must be public
- If you ever need to add new methods to an interface, you will have to update all classes that implement it. This is not always necessary for abstract classes

You can implement as many interfaces as you want. In general, abstract classes are better for future proofing, but interfaces are arguably more flexible since a class can implement multiple interfaces. For an excellent explanation and comparison of abstract classes/interfaces, check out this article.

2.3.6 Packages

A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and name space management. The main reason is to prevent naming conflicts, to control access, and to make searching types such as classes and interfaces a lot easier.