Software Maintenance

Rishi Parmar

January 9, 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

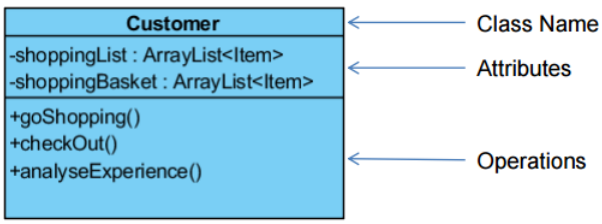
1.1 Understanding the code

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.

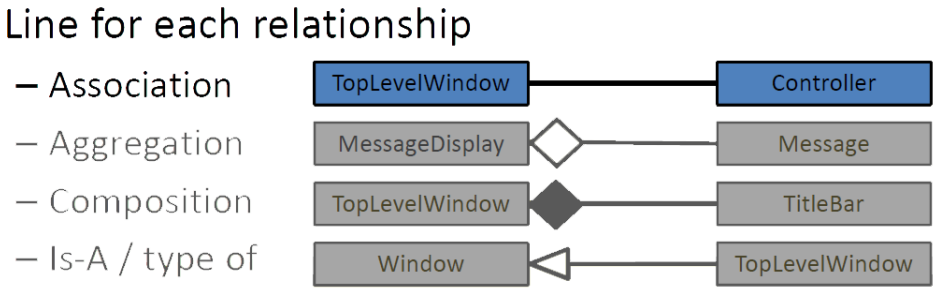
1.1.1 Class Diagrams Review

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

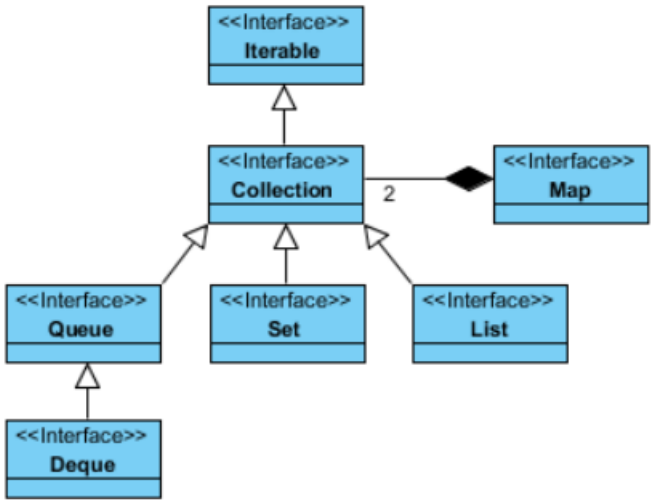
- **Regression Testing** $\longrightarrow$ This is when after you do a bug fix or change, you re-test to make sure that all of the old functionality is still there, and that the bug is fixed
- **Unit Testing** $\longrightarrow$ Automated tests to test the internal workings of the methods. The test is 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

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("Haskell 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("Haskell is shit");

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

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

         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.