Workshop 1

SWEN30006 SEM 1 - 2018

Introduction and Background Knowledge

The aim of this workshop is to familiarise yourselves with the IDE we will be using through this semester and refamiliarise yourself with basic Java concepts and Object Oriented programming concepts. We will also be doing some basic UML Class Diagram modelling using Draw IO, an online diagramming tool that you will be using as part of Project Parts B and C.

It is expected that by the end of this week's workshop you will be familiar and comfortable with the development environment that we will be using throughout the semester. If you finish all tasks in this workshop early, please take the time to read up on upcoming prerequisites and ask your tutor for help where required.

Requisite Knowledge and Tools

It is expected, having completed the prerequisite subjects, that you are familiar with the following terms and concepts:

- Object Oriented Programming (Classes, Inheritance, Polymorphism, Encapsulation and Delegation)
- Basic UML Class Diagram Notation
- The Observer Pattern

The Development Environment

In SWEN30006, we will be supporting the use of the Eclipse IDE for Java development.

If you are intending to use your own laptop for this workshop please ensure you visit The Eclipse Project and download the latest version of the *Eclipse IDE for Java EE Developers* for your operating system and proceed with installation as required.

Though we are supporting the Eclipse IDE, you are free to use others if you have a preference, or no IDE at all if you prefer. However, please be aware that if you choose to use something other than Eclipse, you should not expect the tutors to be able to assist you with problems with your development environment.

Creating a Project from an Existing Program

Today we will be learning how to import a Java program to create a basic project in Eclipse. To begin, either open Eclipse from the start menu (if you are using one of the lab computers), or from where you installed it (if on your own computer). You should now be presented with a screen similar to Figure 1.

This is the default view for everything that you do in Eclipse. Depending on whether you have used Eclipse or not before, you may need to change window layout to the *Java Perspective*. To do this open the *Window* menu and select perspective, then open the *Java Perspective*. You're now ready to create your first Project. Navigate to the *File* menu and from here create a new *Java Project*. You can also use the new button on the Eclipse toolbar for this, as shown in Figure 2.

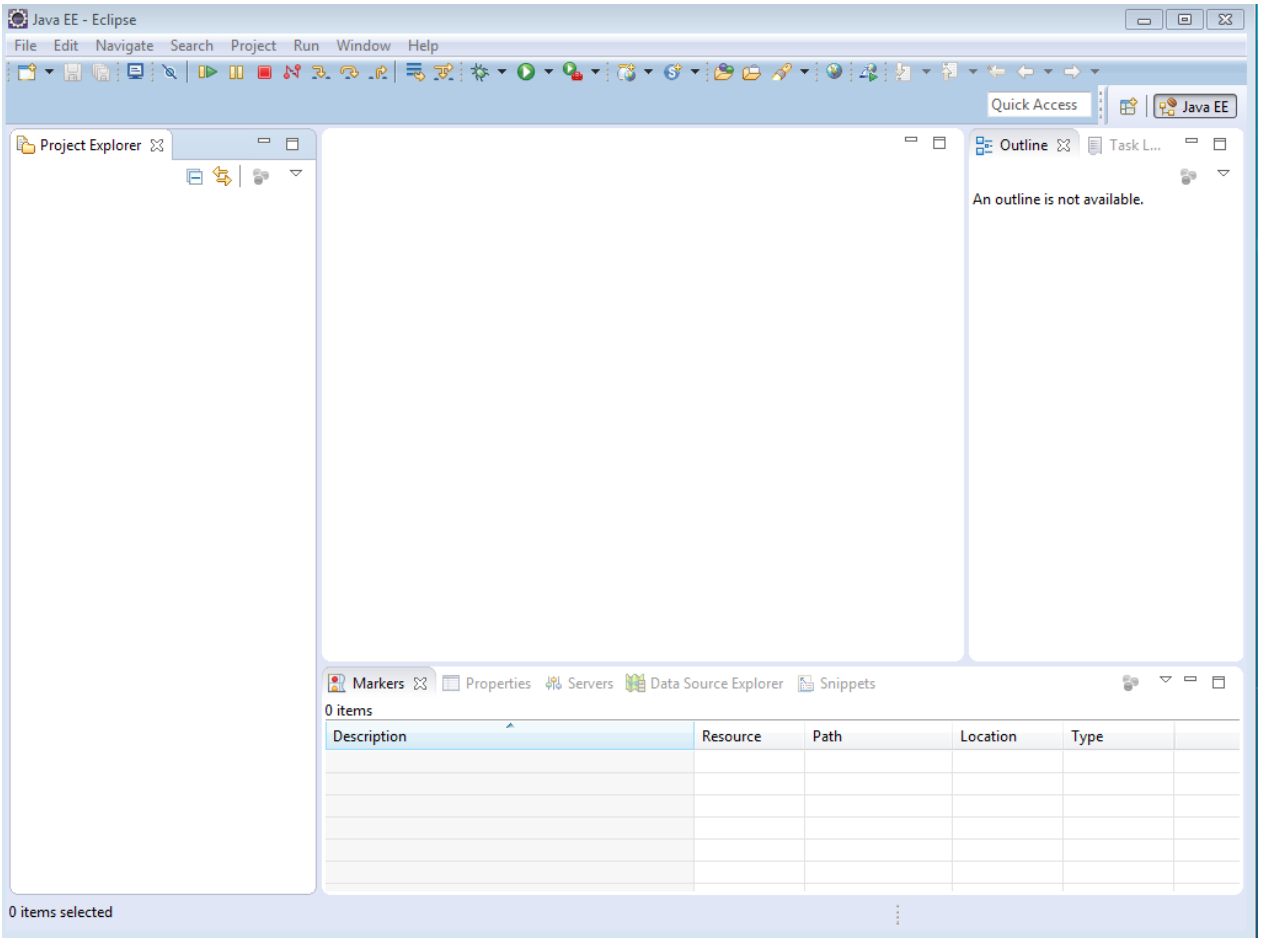

Figure 1: Eclipse Workbench

Figure 2: Eclipse Toolbar - Creating a New Project

Navigate to the LMS, click on the "Larman Case Study Code" link on the left, download the archive file "Monopoly_It1.zip".

Right click in the Project Explorer (or Package Explorer) window and select `Import...` as shown in Figure 3. Then select `Existing Projects into Workspace` and click `Next >`. Then click `Browse...` to `Select archive file` and find the archive file which you have just downloaded and click `Finish`.

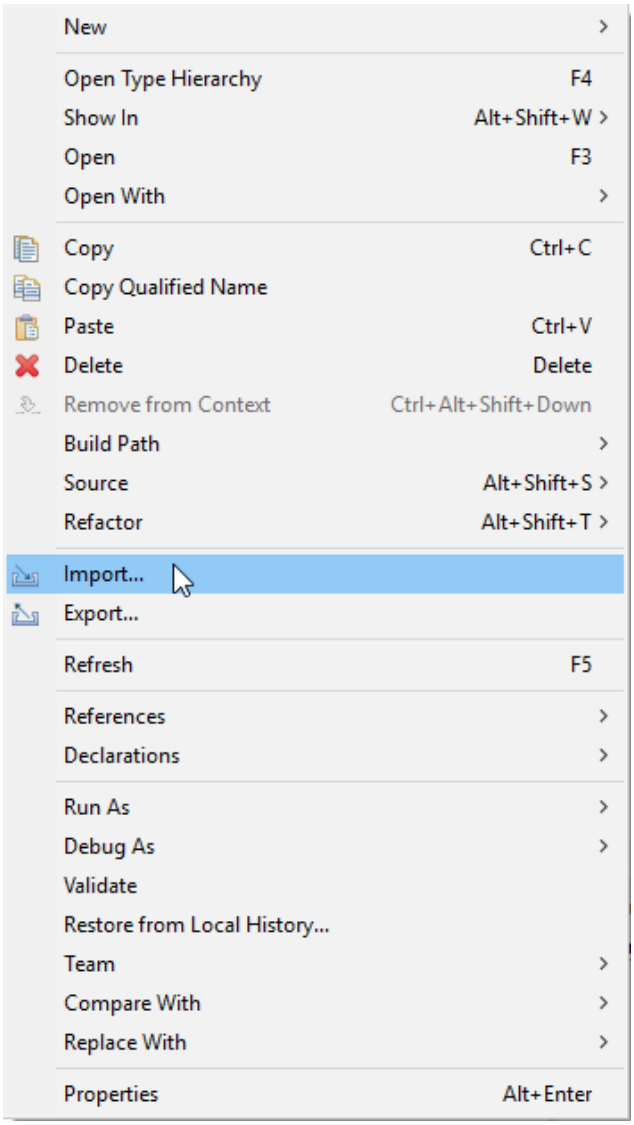

Figure 3: Eclipse Project Explorer Contect Menu - Importing Existing Java Code

You should now see a directory that is similar to the one in Figure 4.

This code you are looking at is based on the case study in the textbook Larman 3rd Edition. As such, you will see further iterations of this code over the course of the semester.

Run this program to see what it does. Following this, your next task is to perform a simple change. Look through the code and and then modify it so that at the end of the simulation it prints out the number of rounds it simulated. When completed, show this to your tutor.

You should now be familiar enough with Eclipse to continue the next part of the workshop!

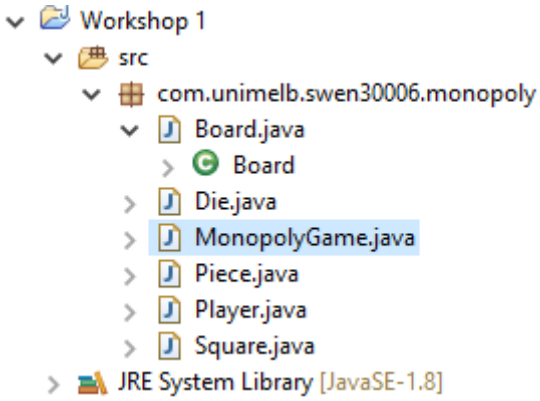

Figure 4: Monopoly Project

The Observer Pattern - A Recap

> Note: This section is considered revision, as it is expected that you have revised these concepts prior to the workshop.

The Observer Pattern is a design pattern that aims to solve problems where some set of objects are dependent on knowing the state (or a subset of the state) of other objects. They may need to know when a value changes for example, or only when a value gets below a certain value. It is a pattern that has become quite pervasive in modern software engineering, and in more advanced forms, powers a large portion of the mobile and distributed computing world.

The basic principle is that some object, an instance of Observer, wants to know about other objects, instances of Subject. The Observer provides a simple interface, `notify()`, so that the Subjects can let it know when something it is interested in has changed state. The Subjects provide two methods for registering and deregistering interest, `registerObserver(observer)` and `deregisterObserver(observer)`. These allow an observer to let a subject know it wants to receive updates. This can be represented in UML as shown in Figure 5.

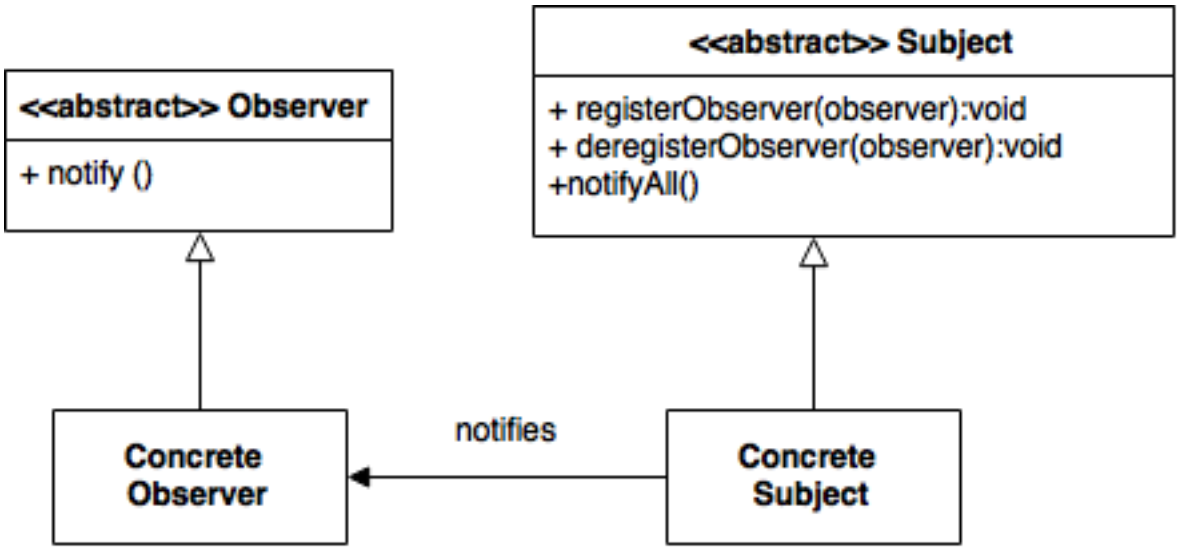

Figure 5: The Observer Pattern

1. It will allow external applications to check the state as well as know what external applications are observing (transparency)
2. Differentiating the subjects and types of subjects that are being notified of is not possible with such a simple interface? When a subject changes state, notify() is called, but the Observer doesn't know which subject called it?
3.

We will now use the Observer Pattern as a basis to explore some basic design oriented thinking. In groups of two, please work through the following questions at your own pace. We will reconvene as a class to discuss some of the answers before we move onto version control.

1. What advantages does the Observer Pattern have over simply using a while loop to check if the state has changed every x seconds?
2. The basic presentation of the Observer Pattern given in Figure 5 has a very simple interface for Observer notification. What downside does having this method cause? Consider what would happen if the Observer was observing hundreds of different types of subjects.
3. How would you go about negating the downside discussed in question 2? Using Draw IO recreate and modify the UML diagram to demonstrate this difference. You can create a UML diagram using the Software Design UML Template, or start with a blank template and select the UML Shape set in Shape pane which appears on the left. Please ask your tutor if you have difficulty with this.

Please show your tutor your solution to the first 3 questions before continuing.

4. Now that you have solved the problem of knowing which Subject notified the Observer, your task is to create a basic Java implementation of the Observer Pattern with this fix. You must create an interface for both the Subject and Observer interfaces as described in your solution to question 3.
5. You must then create two concrete classes that implement these interfaces. The class that implements the observer should simply print to console when they are notified of changes. You must the create a basic driver program to demonstrate that your Subject and Observer implementations work and are consistent with the specification.

Show your tutor when you have completed all exercises to receive your mark for this week.

Workshop Submission

You will be required to demonstrate your solution to all exercises to your tutor for review before the end of the workshop. If you do not complete all the exercises this week, you must complete the remaining exercises by next week's workshop and present these to your tutor in order to receive the mark for this workshop.
