Coursework: Software Engineering Coursework 2 (Final Report)

Module: CS2800 (Software Engineering)

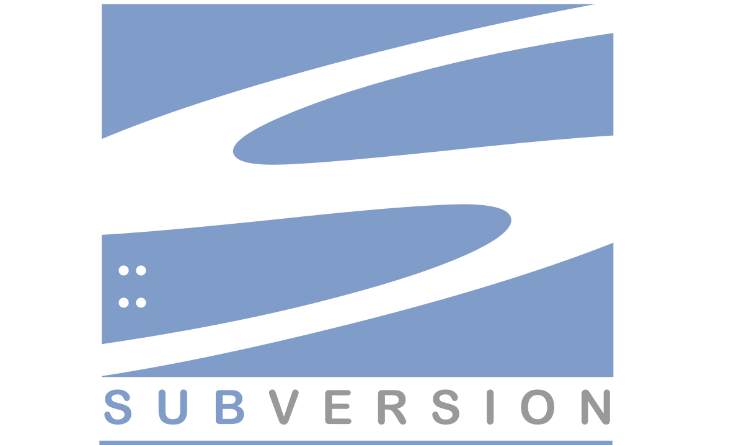
Full Name: Mock King Lun Kelvin

Moodle name: ZHAC195 (King Mock)

Student number: 100903435

Email address: [king.mock.2019@live.rhul.ac.uk](mailto:king.mock.2019@live.rhul.ac.uk)

SVN repository: <https://svn.cs.rhul.ac.uk/personal/zhac195/>

****

**UML and Design Patterns** (Diagram shown on next page):

|  |  |  |
| --- | --- | --- |
| Pattern type | Pattern used | Improvement to UML |
| Architectural | MVC  (Model-  View-  Controller) | * CalcController provides aggregation and composition with ViewInterface and both views (AsciiView and GuiView) respectively. * It gets information of user input from the GUI interface (or Ascii interface) and eventually applies the corresponding view interface so that an appropriate evaluate() can be called (Reverse Polish or Infix). |
| Behavioral | Observer | * An observer has been added as an interface here with the method tell(). * It is for handling events such as recognizing whether the “calculate” button is clicked. If so, evaluate() is called and a result corresponding to the expression that a user enters is printed on screen. |
| Creational | Singleton | * CalcModel and CalcController classes have been amended their structures, each of them forming a singleton pattern. * They restrict themselves to contain only 1 instance since it is impossible to have more than 1 controllers/models managing and running the interface at the same time. * The CalcController class creates an instance of its own type. * Since CalcController is a controller that processes information from the view (user side) to the model (back-end system side), it restricts itself to have multiple instances so that there is only one controller processing the user input (without any conflicts or multiple outputs). |
| Structural | Facade | * CalcController acts as a facade for both AsciiView and GuiView User Interfaces (UI). * Since the mechanism of expression evaluation behind is complicated, both views hide details to reduce complexity. |
| Adapter | * Since the controller does not specify how it works to handle user events, AsciiView and GuiView (as views of user interfaces) both adapt its behaviors. |

Other changes made:

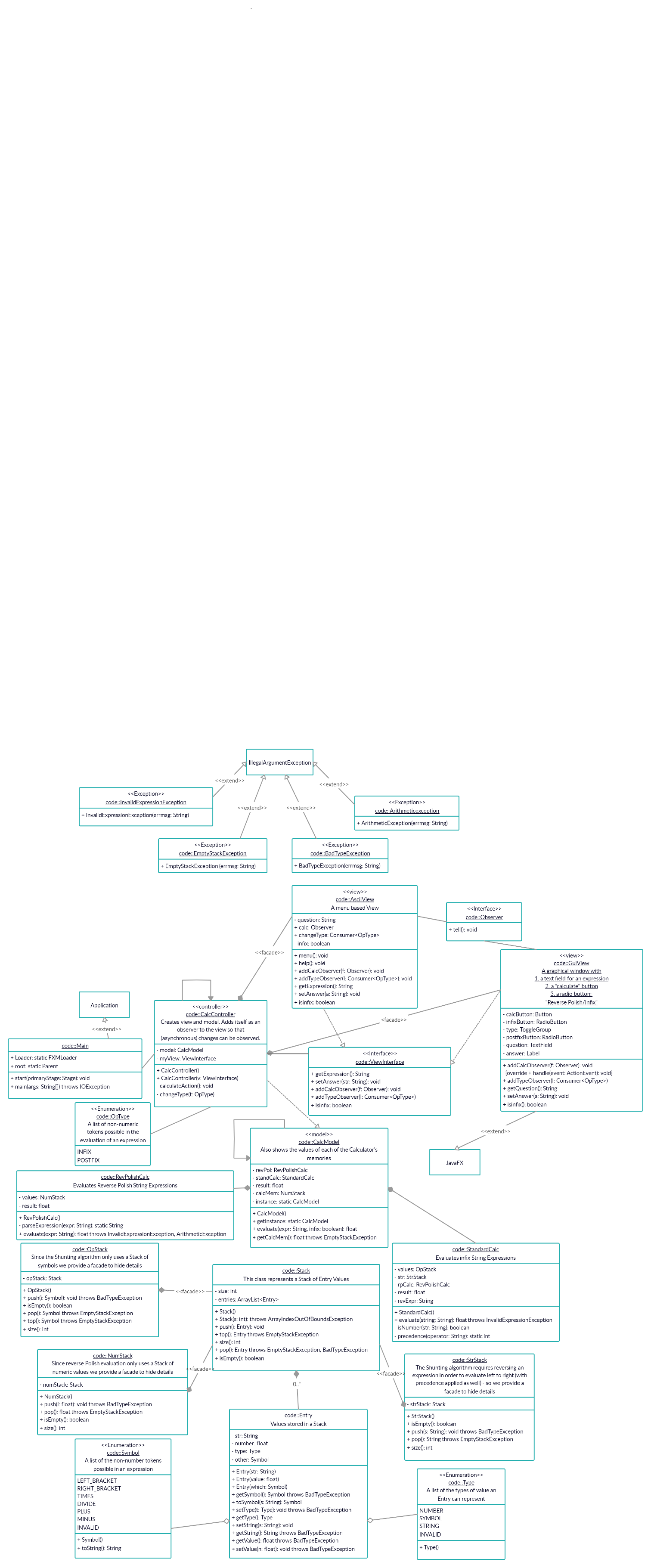
* OpType

It is an enumeration added to distinguish the type (infix or reverse polish) of the expression that the user enters in the User Interface.

* Main

It is a class that contains a main method inside as the driver of the whole program.

* The top() has been added to OpStack so that the previous add-in item is visible for manipulating an infix expression, without deleting the item itself.
* The isinfix() has been implemented onto both view interfaces so that they can identify the expression type by themselves without relying on an external class (to make them less coupled).
* The calculator interface (having dependencies on RevPolishCalc and StandardCalc) has been removed from the overall structure of classes in order to de-couple classes.



**Exam Questions**

1

1. Good code adds business value in a timely fashion.
2. Since the most recent revision on the trunk is supposed to be stored intact, all other revisions on the trunk are stored as negative deltas. A negative delta describes how to go backward in the development history (depending on how much work has been done and committed to the trunk) by producing the desired revision if applied to the successor of that revision. Implementing negative deltas has the advantage that extraction of the latest revision is a simple and fast copy operation.
3. (i) First, an SVN repository has to be created with a new project (with meaningful and an identifiable name) inside it. The first layer of the repository should be in single project layout. Then, setting up the trunk, branch and tag is necessary for the SVN archive before coding. The trunk should contain all committed codes. The branch should contain different folders inside that indicates different small parts of working codes. The tag is mainly for storing snapshots of releases made to the client. Afterwards, *svn copy* can be applied to create different branch folders for storing different parts of working codes. After finishing working on the branch, the local copy should be updated to the current head revision (also using *svn update*) and synchronized to the repository. Finally, all significant changes made at the moment can be committed to the repository for the new branch created using the command *svn commit* so major changes of codes are updated to the repository and stored in trunk. Coding starts.

(ii) Subversion is a server-based system which allows team of multiple people working on a software project together.

In terms of setting up the archive, we should ensure that the main developing codes are all on the trunk since this is the only directory that allows team-working codes. We should also create our own branch using *svn copy* and *svn update* for updating the revision from the current branch in order to obtain the latest revision from the repository especially when other collaborators are updating something else.

In terms of daily procedures, a collaborator should update his working copy with command: *svn update* to ensure that there is no conflicts between the local copy of codes and the repository’s. Commands like *svn add*, *svn delete*, *svn copy* and *svn move* are used so that some files can be modified (added, removed or refactored). If conflicts do occur, *svn revert* is needed to undo and get rid of the unnecessary changes. Since SVN will not allow commits containing a conflicted file until a certain number of temporary conflict files have been deleted, *svn resolved* indicates the completion of resolving the conflicts, before committing codes. There are some commands (*svn status* and *svn diff*) can be used to examine the changes of the working copy.

In terms of finalizing the work into code base, the commands svn checkout and svn co are for checking the repository out so that it indicates the completion of your current edit. Next time getting back to the repository for a new edit, we can use *svn checkin* to check our codes in the repository. After a commit by either collaborator, others need to use *svn merge* so that changes have been applied to all members.

1. By exploring the releases on the SVN archive, 6 main criteria can determine the coding expertise:
2. **Number of faults**

It is an indirect measure of code which determines code reliability. The number of faults reflects on how a programmer manipulate errors within each commit, which is a technique that varies upon different programmers.

1. **Manipulation and removal of code smells**

It is an indirect measure of code. Code smells include the violation of checkstyle, not readable codes (such as lengthy and redundant if-else clauses) and any other bugs in particular programs. Having code smells is a symbol of inadequate programming skills, which can be seen through commits of the programmer on the SVN log/repository.

1. **Documentation Quality**

It is also an indirect measure of code that investigates the code quality of Javadocs and meaningful comments for each commit, class and method.

1. **Testing techniques**

It is a direct measure that looks into the use of TDD test cases. Those test cases can split huge tasks into a single unit that serves only 1 main purpose for a commit. Good programmers usually have small subtasks with more detailed test cases. Thus it measures code quality.

1. **SVN log/repository (number of commits regularly)**

Obviously, the number of commits done on the SVN repository can be a direct measure of code. It determines the productivity of a developer or a team by analyzing how active or regular a project is updated and committed.

1. **Code Length**

Lines of Code (LOC) is a direct measure of code length. Numbers of both commented and non-commented lines can also be measured to conclude whether the code has been performing its tasks right. This can be compared with the complexity of code.

1. (i) As Final ensures the instance to be constant after creation, no one can change the instance after the constructor. This gives advantages such as easier serialization, and insurance against unauthorized persons changing the object by writing defensive codes.

(ii) Serialization can cause breakage of singleton property of singleton classes. It is used to convert an object of byte stream and either save it in a file or send it over a network. After serializing, it will create a new instance once it is de-serialized. Once this happens, the singleton pattern is broken. Since only 1 instance is allowed for this design pattern, it is hard to create (and replace with a new one) with serialization, and so do making a singleton Serializable.

(iii) Basically, a class of singleton pattern creates an instance for itself. However, it restricts multiple objects and sub-classes. Alternatively, factory is a design pattern that allows object creation (or called instantiation) to implement the same interface. It not only separates multiple object creations, but also allows the creation of complex objects which share some initialization logic, without changing the classes. So this pattern allows multiple instances to be serialized, while the singleton pattern does not, where both of them implement objects on the same interface.

A singleton object provides a global point of access to themselves and thus, it is tightly coupled to the class itself. A factory object allows the class hierarchy to be decoupled, which solves the restriction occurred in a singleton object.

Also, since the factory pattern involves an extra class which has methods for building and returning objects, it solves the problem of sub-classing.