CS2800: Software Engineering Report

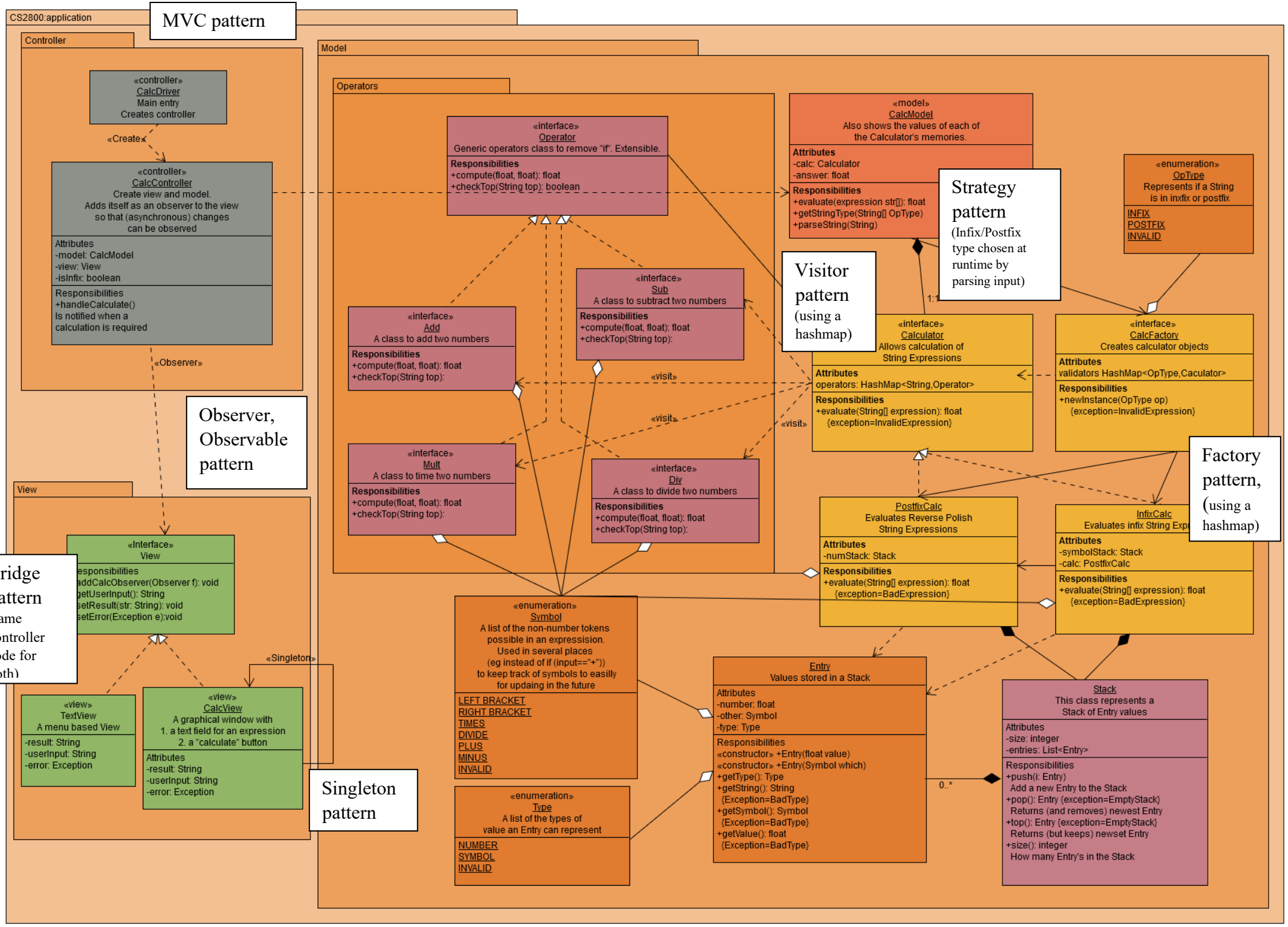
Henry Russell

Moodle: ZFAC023

Student Number: 100892912

Russell, Henry (2018) <ZFAC023@live.rhul.ac.uk>;

https://svn.cs.rhul.ac.uk/personal/zfac023/CS2800/calculator/


CS2800:application
MVC pattern
Controller
Model
«controller»
CalcDriver
Main entry
Creates controller
«Create»
«controller»
CalcController
Create view and model.
Adds itself as an observer to the view
so that (asynchronous) changes
can be observed
Attributes
-model: CalcModel
-view: View
-isInfix: boolean
Responsibilities
+handleCalculate()
Is notified when a
calculation is required
«Observer»
Observer, Observable pattern
Operators
«interface»
Operator
Generic operators class to remove "if". Extensible.
Responsibilities
+compute(float, float): float
+checkTop(String top): boolean
«model»
CalcModel
Also shows the values of each of
the Calculator's memories.
Attributes
-calc: Calculator
-answer: float
Responsibilities
+evaluate(expression str[]): float
+getStringType(String[] OpType)
+parseString(String)
Strategy pattern
(Infix/Postfix type chosen at runtime by parsing input)
«enumeration»
OpType
Represents if a String
is in inxfix or postfix
INFIX
POSTFIX
INVALID
Visitor pattern
(using a hashmap)
«interface»
Add
A class to add two numbers
Responsibilities
+compute(float, float): float
+checkTop(String top):
«interface»
Sub
A class to subtract two numbers
Responsibilities
+compute(float, float): float
+checkTop(String top):
«visit»
«visit»
«visit»
«interface»
Calculator
Allows calculation of
String Expressions
Attributes
operators: HashMap<String,Operator>
Responsibilities
+evaluate(String[] expression): float
{exception=InvalidExpression}
«interface»
CalcFactory
Creates calculator objects
Attributes
validators HashMap<OpType,Caculator>
Responsibilities
+newInstance(OpType op)
{exception=InvalidExpression}
«interface»
Mult
A class to time two numbers
Responsibilities
+compute(float, float): float
+checkTop(String top):
«interface»
Div
A class to divide two numbers
Responsibilities
+compute(float, float): float
+checkTop(String top):
Factory pattern, (using a hashmap)
PostfixCalc
Evaluates Reverse Polish
String Expressions
Attributes
-numStack: Stack
Responsibilities
+evaluate(String[] expression): float
{exception=BadExpression}
InfixCalc
Evaluates infix String Exp
Attributes
-symbolStack: Stack
-calc: PostfixCalc
Responsibilities
+evaluate(String[] expression): float
{exception=BadExpression}
View
Bridge pattern
(same controller code for both)
«Interface»
View
Responsibilities
addCalcObserver(Observer f): void
getUserInput(): String
setResult(str: String): void
setError(Exception e):void
«Singleton»
«view»
TextView
A menu based View
-result: String
-userInput: String
-error: Exception
«view»
CalcView
A graphical window with
1. a text field for an expression
2. a "calculate" button
Attributes
-result: String
-userInput: String
-error: Exception
Singleton pattern
«enumeration»
Symbol
A list of the non-number tokens
possible in an expression.
Used in several places
(eg instead of if (input=="+"))
to keep track of symbols to easilly
for updaing in the future
LEFT BRACKET
RIGHT BRACKET
TIMES
DIVIDE
PLUS
MINUS
INVALID
«enumeration»
Type
A list of the types of
value an Entry can represent
NUMBER
SYMBOL
INVALID
Entry
Values stored in a Stack
Attributes
-number: float
-other: Symbol
-type: Type
Responsibilities
«constructor» +Entry(float value)
«constructor» +Entry(Symbol which)
+getType(): Type
+getString(): String
{Exception=BadType}
+getSymbol(): Symbol
{Exception=BadType}
+getValue(): float
{Exception=BadType}
0..*
Stack
This class represents a
Stack of Entry values
Attributes
-size: integer
-entries: List<Entry>
Responsibilities
+push(i: Entry)
Add a new Entry to the Stack
+pop(): Entry {exception=EmptyStack}
Returns (and removes) newest Entry
+top(): Entry {exception=EmptyStack}
Returns (but keeps) newset Entry
+size(): integer
How many Entry's in the Stack

Part 1: UML

I have updated the UML for abstraction, code reuse and avoiding antipatterns with the aim of modularisation, simplifying complex parts, readability, understandability and preparing for future updates and maintenance. To do this, I went through several stages:

The most noticeable change in the diagram was in code organisation, I decided to break things down into multiple subpackets to make the project easier to navigate and to group logical concepts. I also renamed some of the classes, attributes and fields (such as "String what") to make the code more readable.

Another such modification involved removing details from the stack which I wouldn't use. This means less code and less code means less bugs and is faster to create. I removed the String entry type, replacing it with symbol only, this allows code to be maintained better as new symbols can be added and removed and are stored in a central enum. I also delete the three façades as they contained repeated code which is the once and only once smell in my opinion. They didn't solve anything that couldn't already be solved with java generics. Now all the stack code is in a single class which reduced coupling. It also simplifies the UML.

Modularising the calculate methods using visitor pattern was another modification. This allowed me to remove long switch statements by mapping each symbol to a visit function. This is an advantage is because of how short and readable the infix and postfix calculate methods became as it makes them easy to debug (less tracing through ifs), maintain (calculate method is short), and understand as the (abstracts away unnecessary details). It also makes the program very extensible and easy to add new operators in the future, all I need to do is make one new class to define its behaviour (implementing the compute method) and add the new operator to the symbol enum because I did not hard code any strings in the program, mentioned above.

Another seemingly small but very significant modification was removing the radio button from the javafx view, this simplifies things from a user perspective as the calculator becomes cleaner and there are fewer things to worry about. From a UI/UX designer perspective, simple UI's are easier to maintain and faster to create. Removing the radio button also did two huge things to my program. Firstly, making it trivial to bridge the implementations between TextView and CalcView (as there is one less state to worry about), which in turn also simplified the controller as it now only has a couple of shared view methods. It also allowed me to select which calculator to use (infix/postfix) at runtime (by checking if last character is a symbol) which is strategy pattern. To implement this, I used factory creation pattern which builds the correct type of calculator with a map and used an enum to represent the different operations types. This allowed me to further break down the code and remove branching if statements which lowered overall code as less checks where needed.

Everything else was kept consistent with the original UML. The MVC, and singleton patterns were kept (Dave's code) as well as well as the observer listeners which were very useful. The result was a flattened UML diagram with reduced coupling to the model, view and controller classes where the code in each class is short and each class does one thing well.

Part 2: Exam Questions

a) Good code must work and be readable.

b) A delta is a file modification or change in a version control system (VCS). Negative deltas are the changes from the current version of the software to the previous version. Negative deltas are preferred and used by SVN because they allow developers to cycle back to a previous revision, this is useful if a newer version has been exposed to have flaws such as bugs or having reliability or security vulnerabilities.

c) 1) Before coding the following steps must be taken:

i New feature branches should be created on the repository:
$svn\ copy\ http://svn.server.com/svn/buildings/trunk\ \backslash\ http://svn.server.com/svn/buildings/branches/internalisation\ -m\ "created\ branch\ for\ internalisation\ code"$
$svn\ copy\ http://svn.server.com/svn/buildings/trunk\ \backslash\ http://svn.server.com/svn/buildings/branches/screen\_input\ -m\ "created\ branch\ for\ new\ input\ screen\ "$

ii Master branch (trunk) should be cloned to local machine: $svn\ checkout\ http://svn.server.com/svn/buildings/trunk$

iii Developer should then switch to assigned branch: $svn\ sw\ http://svn.server.com/svn/buildings/branches/screen\_input$

2) Follow these steps:

i The first thing the college should do is steps ii) and iii) from 1).

ii The college next develops (and test if applicable) a small slice of a feature (e.g. TDD cycle), a bug fix or general refactoring quality of life improvement that is different to what current developers are working on (to avoid potential conflicts). $svn\ update$ should be run at regular intervals to ensure no conflicts have occurred.

iii Files to add to add or remove from the repository should be staged
$svn\ add\ input\_view.js$ \ $svn\ rm\ view.js$

iv If a conflict has occurred on $svn\ update$, the bad file will be listed (e.g. input_view.js), it must be resolved. This can be done by:

a. Resolve by accepting your changes:
$$svn\ resolve\ --accept\ mine-full\ input\_view.js$$

b. Resolve by accepting their changes:
$$svn\ resolve\ --accept\ theirs-full\ input\_view.js$$

c. Resolve manually: (e.g. with text editor such as atom or sublime, vim or dedicated software BEFORE the command):
$$svn\ resolve\ --working-full\ input\_view.js$$

v Note: in some circumstances, the other developers involved should be contacted to see if it is okay to overwrite their changes.

vi When ready to push changes to branch, $svn\ update$ should be run again, any conflicts should be resolved then $svn\ commit -m\ "MESSAGE"$ where the message follows the company template.

vii Once approved by the lead, code review and test team, the branch should be merged to the trunk.

$svn\ update$ // check that current version is up to date

$svn\ status$ // check that the current version has no local changes

$svn\ sw\ http://svn.server.com/svn/buildings/branches/trunk$

$svn\ merge\ --reintegrate\ \backslash$

$http://svn.example.com/repos/calc/branches/screen\_input$.

This should then be built, tested and committed.

$svn\ commit -m\ "Merged\ screen\_input\ into\ trunk"$

viii A sync merge should be done at this point to make sure the current branch is up to date with the trunk. This is done the same as above but from the branch instead.

ix Repeat above until branch is complete. When it is no longer being worked by anyone it should be deleted.

$svn\ rm\ http://svn.server.com/svn/buildings/branches/screen\_input\ \backslash$

$-m\ "Deleted\ branch\ screen\_input"$

d) A developers coding expertise can be defined as the ability to consistently deliver "good code" as defined in "a)" in a timely fashion. One way of measuring this is to use SVN to assess the performance of the engineer by analysing their history on:

a Frequency of commits;

This is not always fair as number commits do not necessarily transfer to their quality. Therefore, all you can determine from this is "activity".

b Number of Lines committed;

This is also not always fair does not consider complexity of task and quality of the code. Encourages long comments, boilerplate and repeated code. If comments are not assessed, it punishes their usage.

c Frequency of productive commits messages;

This is an improvement on d)a) and d)b) as it shows how often value is added. However, developers may then be interested in producing working code over good and efficient code.

d Reviews for good code, smells and adherence to conventions;

This is the fairest, as it encourages effective problem solving and clean solutions. Code can be assed based on its time complexity, functionality, readability, reusability, modularity, tests and maintainability. However, it still has some issues. It can only be used on the final version as older versions may contain code that is in yet to be refactored. Also, code reviews can be time consuming so not always possible, finally it does not consider the time that went into the code.

e) Singletons involve only one class which is responsible for instantiating itself. Final will guarantee that the instance following creation cannot be modified, it is therefore good practice to use final on a singleton as it shows the instance is immutable. A private constructor already prevents an object of type singleton being created by anything other than the singleton itself, however if setters are included the instance could change. Final protects this from happening and offers assurance against someone else changing the code later.

f) Serialization takes an object's state from memory and writes it to a file/sends it to another computer over the network. If, for example you serialize a singleton twice, two singletons now exist. The intention of a singleton is to only ever have one. It is very hard to serialize in such a way that prevents this from happening.

g) The factory design pattern provides a way to create objects without directly calling their constructor. Instead the request for an object is sent to a factory which creates and returns a new object of the specified type. A factory can be used to replace singleton if it returns only one object once, through an internal state. This can later be extended to return multiple objects if needed.

The problem of deserialization is solved by sending a request to the factory for a deserialized object, the factory then creates and returns the deserialized object. If then later the factory is asked for another deserialized object, the factory will return null as there should only be one of the “singleton style” class.

The problem of sub-classing is also solved as the class file of the object is only available to the factory and not the calling class. This prevents a subclass from being created which is good as a “singleton style” class should not be sub classed.