CSE 3311 Project Report

Submitted by:

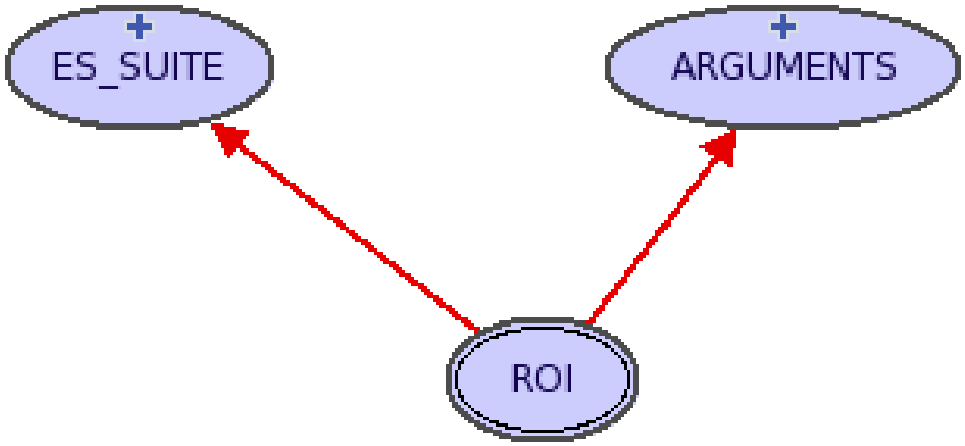
Team members	Name	Prism Login
Member 1:	Weili shao	CSE23097
Member 2:	Manik gupta	Cse13025
Member 3:	Xiaoyu zhang	Cse13026
Member 4:	Dmytro Shebanov	cse21045
Member 5:	Abasifreke James	CSE22010
*Submitted under Prism account:		

* Submit only under **one** Prism account (do **not** submit twice)

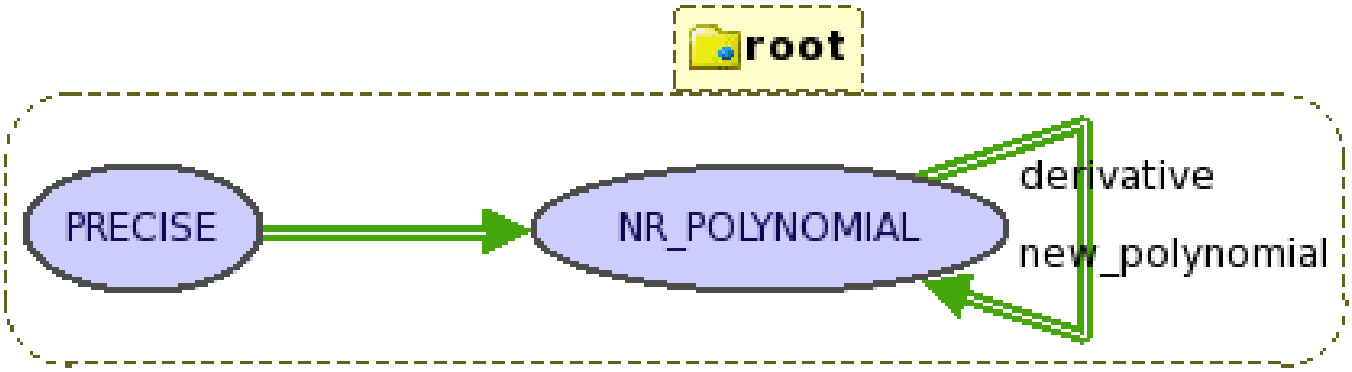
Contents

1. BON Class Diagram

ROI BON CLASS



PRECISE BON CLASS DIAGRAM



2. Cluster and Class Descriptions

We have three clusters, with the following contents:

* parsing cluster:

Contains the provided csv classes, along with PARSER class developed by us,

Which is capable of returning an ARRAY of TUPLE, each representing a data row, as well as a TUPLE that represents the data in the header file. It's also possible to get a detailed error message, if an error occurs.

* root cluster:

Contains the 'main' class, ROI, which is responsible for connecting everything together and producing output. It also contains the provided polynomial class, as well as two utility classes that provide the financial calculations.

* test cluster:

Contains some tests.

3. Design Decisions

The command pattern (while useful for the provided example test cases) seemed overkill for our parser, as we only needed to perform the commands once. Also, there is no need to be able to dynamically insert (or remove) commands from a queue. Consequently, it was abandoned in favor of a method that performs these commands in sequence.

The requirement that our program should never crash forced us to resort to defensive programming. All possible inputs that could cause failure (with the exception of diverging polynomials) are trapped at the parser stage, which makes most (at least pre-condition) contracts a little redundant.

We decided to parse the entire csv spreadsheet into an array before returning it ("batch-processing"), rather than returning one line of data at a time ("streaming"), the latter being a lot easier for the person writing the parser. However, the former approach was significantly more convenient for the people implementing the financial calculation classes, so in the end the convenience of the many outweighted the convenience of the few.

Error handling is done by setting an Integer error flag (as opposed to Boolean), since we have a third type of error (which is really a warning). There are several codes, but it really boils down to just 3 possibilities (success, really bad error, warning), which map to 0, >0, <0, respectively. The actual error message can be fetched from the parser object, so this strategy cannot really be criticized for being tightly-coupled (eg: if the printer was responsible for mapping error codes to error messages, but this is not the case).

4. Design Patterns

As previously mentioned, we decided not to use the Command pattern for our parser. At the early stages, it still uses the Iterator pattern to traverse the rows (as implemented in the provided classes), and we haven't changed this implementation.

At the conceptual level, our strategy is essentially the Pipe-and-Filter architecture; the task is easily split into several computational steps, with some code to ensure that the data between the filters is transferred in the correct format. More specifically, it's the Batch Sequential style, since generally, each filter will finish working on the entire dataset before producing output (although, the earlier stages of the parser do streaming).

5. Testing

Provide descriptions for the unit and the acceptance test cases that you have developed for ensuring the quality of the system.

We have implemented several test cases to ensure we obtain precise outputs for TWR and PRECISE ROI calculation. First we implemented test case to ensure TWR and PRECISE calculate with small data. Then we implemented test cases to ensure TWR and PRECISE can take argument in the function and outputs the result.

6. Contracts

Describe **two** of the most important contacts that you have composed in this system. Do not list trivial contracts like “require input /= Void”.

require else

end_date.is_greater (start_date) = true

voidcheck: start_date/= void; end_date /= void; data /= void

emptycheck: not data.is_empty

We used this require statement to make sure that the client provides an end date that is greater than the start date. And that the start date and end date fields are not void and that the data field is not empty.