13 November, 2014



Team Number: 4 Consultation Day/Hour: Tuesday, 1pm

Team Name: Team 4

Team Members Information:

Group PKB

Saloni Kaur A0084053L [a0084053@nus.edu.sg](mailto:a0084053@nus.edu.sg)

M I Azima A0085594N [a0085594@nus.edu.sg](mailto:a0085594@nus.edu.sg)

Group PQL

Saima Mahmood A0084176Y [a0084176@nus.edu.sg](mailto:a0084176@nus.edu.sg)

Nguyen Trong Son A0088441 [A0088441@nus.edu.sg](mailto:A0088441@nus.edu.sg)

Vu Phuc Tho A0090585X [A0090585@nus.edu.sg](mailto:A0090585@nus.edu.sg)

CS3202 Software Engineering Project

Final Report

# Project Story

CS3202 has been an eventful project based module. In the continuation from CS3201, we have implemented the full scope of the Static Program Analyzer (SPA), alongside some extensions. We ensured that we improved on the weaknesses from our previous project and strengthened the base to build our SPA upon. We also had the addition of a new member with, alongside one member of our group opting to change groups. This switch of members actually improved the dynamics of our group, allowing us to work better with one another. The remainder of this report shall discuss in detail how we went about implementing the various components of the SPA. This discussion shall include a summary of our main achievements (Section 1), project plans (Section 2), UML diagrams (Section 3), design decisions (section 4), coding standards and experiences (Section 5), query processing (Section 6), testing (Section 7) and finally end off with a concluding discussion (Section 8).

# Summary of Main Achievements

## Basic SPA Functionality

For the purposes of the CS3202 development of the SPA, we have implemented the full SPA as described in the Project Handbook. This includes the implementation of the components:

* Parser
* Design Extractor
* Program Knowledge Base (PKB)
* Query Processor (QP).

The PKB stores the design abstractions implemented:

* Abstract Syntax Tree (AST)
* Follows/Follows\*
* Parent/Parent\*
* Modifies
* Uses
* Calls/Calls\*
* Next/Next\*
* Affects/Affects\*

The QP handles the processing of queries involving the aforementioned design abstractions alongside a combination of “with”, “such that” and “pattern” clauses. It has been implemented to also return tuple results. The QP also includes components that handle the optimization of query evaluation.

The description above just highlights the main functionality implemented. Overall, all of the required functions from iteration 1-3 have been implemented. The details of their implementation shall be discussed in the later sections of this report.

## Highlights of System

In the implementation of the functionality, defined by the handbook, we have ensured that aspects of our software standout from the norm. This is in the way that we have implemented some functions and also the addition of certain components. The main highlights of our project includes:

* Polymorphism of data structure for PKB (MapTable & ListTable)
* Next\* Implementation
* Addition of Query Representator (QR) and Query Optimizer (QO) in the QP
* Query Evaluator (QE) in the QP [really?]

These highlights will be further elaborated on in Section 4 and Section 6.

## Extension for Bonus Points

We have implemented the first extension for the extended code pattern. This includes the relationships Contains, Contains\* and Siblings. Details entailing the design decisions for this extension shall be discussed in Section 4. The test cases for this extension has been included alongside the other test cases for the SPA.

# Project Plans

## Project Schedule

The tables in this section show how we distributed the work into various tasks throughout the 4 main iterations.

### Iteration 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Team Member** | **Testing** | **Writing Test Cases** | **Revamp of PKB Tables** | **Refractoring QP** | **Working on QP** | **Extending Parser Functionality** |
| Azima | \* | \* | \* |  |  |  |
| Saima | \* |  |  |  | \* |  |
| Saloni | \* |  | \* |  |  |  |
| Sean | \* |  |  |  |  | \* |
| Tho | \* |  |  | \* | \* |  |

Table 2.1: Iteration 1 Work Distribution

Since this was the starting iteration, we had mainly focused on revamping the design of the whole system, from the Revision of the Prototype iteration. As a result, we were unable to finish implementing all of the required functionality defined in iteration 1.

### Iteration 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Team Member** | **Testing** | **Writing Test Cases** | **Revamp of TNODE** | **Next for PKB** | **Working on QP** | **Fixing issues with Parser** | **Report** |
| Azima | \* | \* |  |  |  |  |  |
| Saima | \* | \* |  |  | \* |  | \* |
| Saloni | \* | \* |  | \* |  |  |  |
| Sean | \* |  |  |  |  | \* |  |
| Tho | \* |  | \* |  | \* |  | \* |

Table 2.2: Iteration 2 Work Distribution

This iteration saw us complete all of the requirements from iteration 1 and iteration 2. The main setback in this iteration was in the design of the TNode data structure. This class is used to build various trees used for data storage. Previously we had implemented the TNode without the use of pointers. This means that every time we would pass data, the TNode would create a new copy instead of passing the original data from one function to another. This would slow down the SPA process, since each time we would add a new node, the existing tree would be copied another time to attach to the new node. This was seen to be a waste of processing time and storage. It slowed our system down considerably.

We overcame the aforementioned problem by creating a new TNode and assigning a pointer to its address. Basically, creating a pointer of type TNode.

Old way:

TNode node();

TNode \* pointer = &node;

New way:

TNode \* pointer = new TNode();

### Iteration 3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Team Member** | **Testing** | **Writing Test Cases** | **Revamp of Relationships Data Structure** | **Affects** | **Working on QP** | **Fixing Issues with QE** | **Report** |
| Azima | \* | \* |  |  |  |  | \* |
| Saima | \* |  |  |  | \* |  | \* |
| Saloni | \* | \* |  |  |  |  | \* |
| Sean | \* |  | \* | \* |  |  |  |
| Tho | \* |  |  |  | \* | \* |  |

Table 2.3: Iteration 3 Work Distribution

Iteration 3 saw us having a number of issues with the implementation of the Next \* and Affects relationships. Also we had to improve on the managing of temporary results in the Query Evaluator, for multiple clauses. What these issues were and how we fixed them would be discussed in Section 4 and 6 respectively. This slowed us down considerably in this iteration and we could carry out aggressive testing, as we had initially planned to.

### Iteration 4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Team Member** | **Testing** | **Writing Test Cases** | **Extension Relationships** | **Fixing Issues with System** | **Report** |
| Azima | \* | \* |  | \* | \* |
| Saima | \* | \* | \* | \* | \* |
| Saloni | \* | \* | \* | \* | \* |
| Sean | \* |  |  | \* | \* |
| Tho | \* |  |  | \* | \* |

Table 2.4: Iteration 4 Work Distribution

The emphasis placed on this iteration was to implement the extension relations, Siblings and Contains, and also on vigorous testing on the system.

## Organization of Meetings

As this project was implemented in a total of 4 iterations, we measured our progress based on the state of our SPA in the previous iteration and the requirements for the current iteration. To make sure that we stayed consistent, we met, on average, at least 3 times a week, including every Tuesday which was our consultation time slot. In the first meeting of the week we would set an agenda for the things to be achieved in that specific meeting, and for that week. The subsequent meetings would be organized around our tutor’s feedback and on improving our system based on the critique we would have received. Fig 2.1 below shows an excerpt from one of our meeting’s agenda.

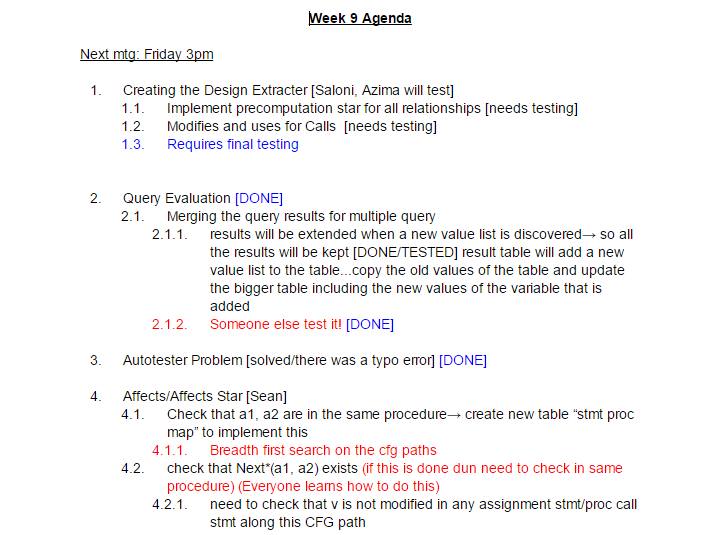


Figure 2.1: Agenda Excerpt

# UML Diagrams

The UML sequence diagrams presented in this section display how the SPA program flow works between the Parser, PKB and QP. These diagrams allowed us to visualize the various component interaction of the SPA and thus aided in the project planning.

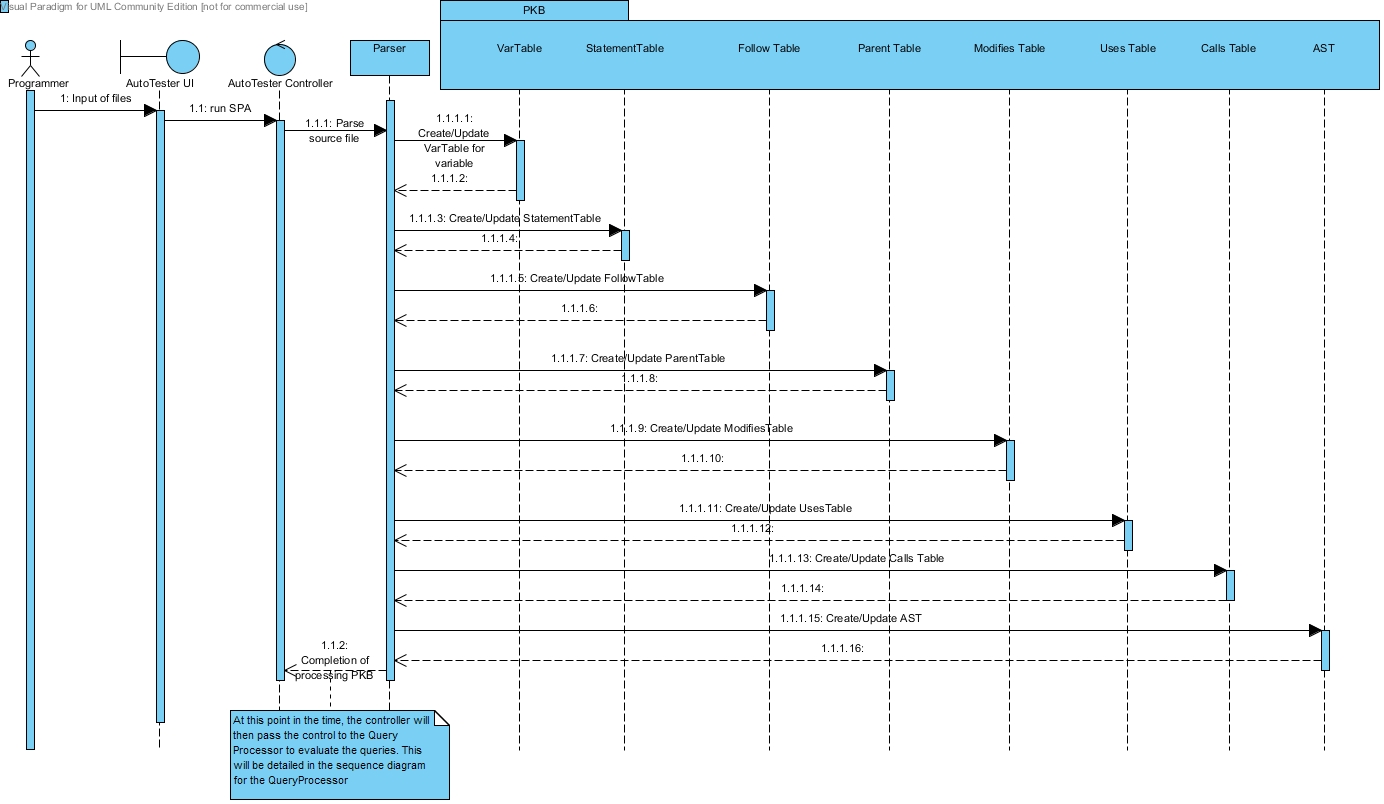


Figure 3.1: Sequence Diagram for Processing PKB

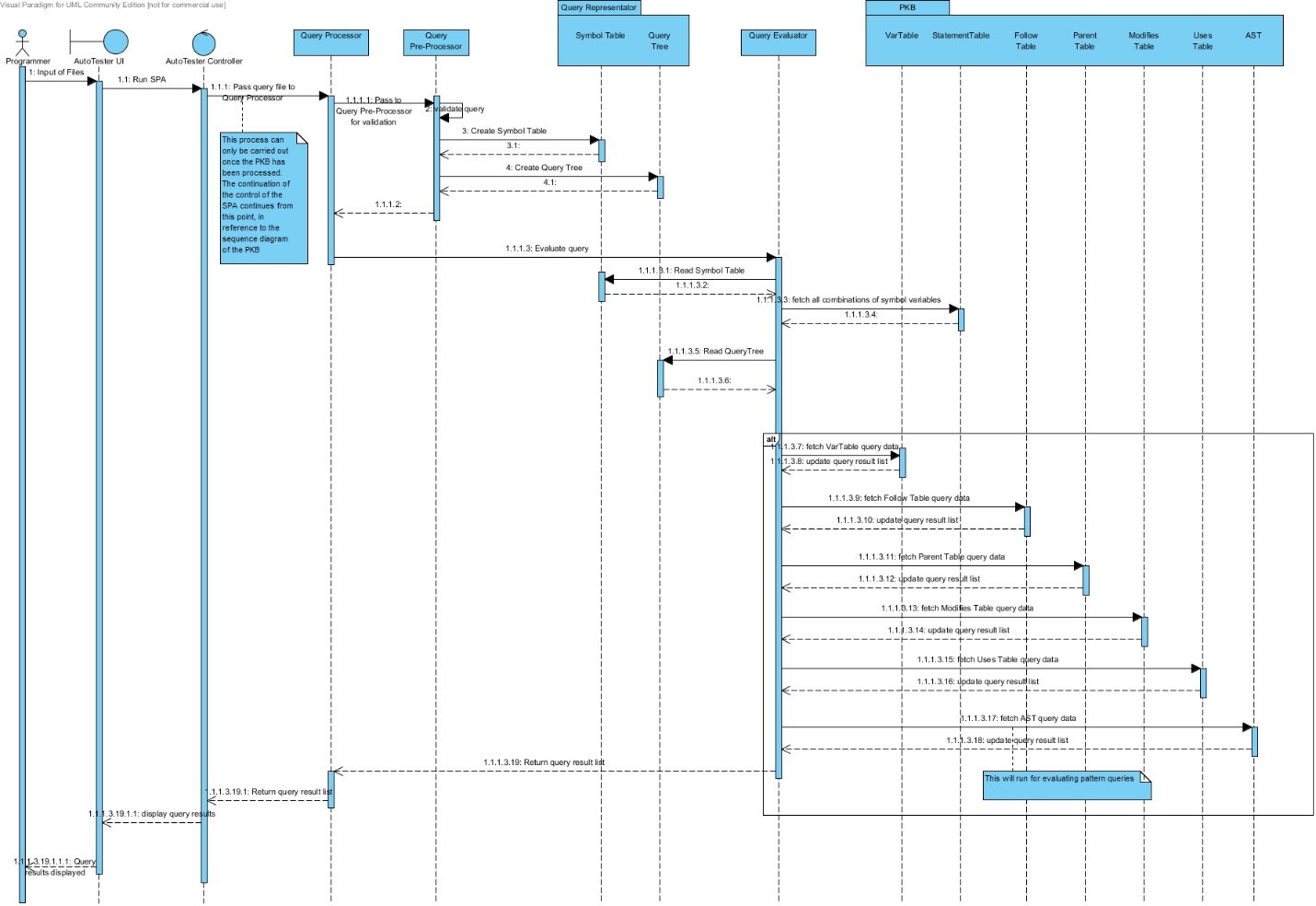


Figure 3.2: Sequence Diagram for Query Processor Flow

These 2 diagrams shown in Fig 3.1 and Fig 3.2, allowed us to understand the system properly before we went to code it out. In making these UML diagrams, we are able to separate the various components and abstract out the most important things. This enabled us to look at each decomposed aspect of the system and come up with an in-depth system architecture. This in turn allowed us to see what the basic requirements for each component were.

# Design Decisions

////talk abt separation of concerns and virtues of simplicity and standardization

//system decomposition with information hiding {section 10.1}

//show software architecture (emphasis on the differences in the QP)

## PKB Data Structure Representation

- discuss the design problem

-identify goals to be met by relevant design solutions

-discuss alternatives, previous implementations

-analyze strengths and weaknesses [include big o notation analysis]

-discuss implemented design based on the weakness of the previous design decisions

-analyze using big o notation

-give example for insertion, searching and getting for one relationship

## PKB API

## SPA Relationships

### AST

### Modifies/Uses [for procedure calls]

### Parent/Follows

### Calls

### Next/Next Star

### Affects/Affects Star

### Contains/Contains Star

### Siblings

## Design Patterns

# Coding Standards & Experiences

In terms of the coding standards, our group has decided to adopt the following naming conventions described in this section.

## Naming Conventions

### General Rules

* Do not use underscore, hyphen or any other non-alphabet characters.
* Any name should has all the first letters of internal words capitalized, e.g. getProcName()
* Avoid using abbreviations. Some words are acceptable in short forms, including: *Var*, *Proc*, *Stmt*, *AST*. Other words such as *Children*, *Number* should be fully spelled out.

### Specific Rules

* API Name:
  + API names should be nouns, in mixed case with the first letter of each internal word capitalized.
* Method:
  + Method names should be in the form of a verb. With method names containing more than one word, use mixed case with the first letter of each internal word capitalized.
  + Name of some specific methods:

1. Methods to insert new records to the database should have the form insertXXX().
2. Methods with return value type BOOLEAN should have the form: isXXX() e.g. isExist(), isMatchVar().
3. Methods with return types of other values should have the form getXXX() e.g. getVarName()
4. Methods that return the number of records inside a table/ list should have the form getSize().
5. Methods that change the values or status of an object should have the form: setXXX()
6. Methods that return values from star queries, such as Calls\* and Next\*, should have the form getXXXStar().

In all of the above examples, the “XXX” is used in place of the specific name which the method will adopt.

To keep the abstract and concrete PKB API in sync, we created a variable table, statement table and procedure tables. These tables are vectors mapping variable names to indexes of the vectors. So that a API method like BOOLEAN isModifies(STMT\_NUM s1, INDEX varIndex)understands that INDEX is the mapped value of a certain variable name, where INDEX is just an integer in C++ type.

# Query Processing

## Query Validation

Table-driven approach to query validation??

## Query Evaluation

### Data Representation (QR)

### Basic Query Evaluation

#### Manage the temporary results

### Optimization

### Design Decisions

# Testing

## Testing Experience

The testing of the SPA was carried out in three stages. Firstly, for each component of the PKB and PQL, we carried out unit testing. Unit testing allowed us to test the internal functions of each component. After unit testing was completed, we carried out integration testing of the different components. This ensured that the different components work properly together, for instance, the Parser and the Query Evaluator. Once we tested specific components, we then tested the system as a whole in validation testing. Keeping in mind that his project was implemented in iterations, we carried out this 3 stage testing procedure each time a new functionality was implemented. Even if it was for a new function in a preexisting component.

The most vigorous type of testing that was carried out was validation testing. We tested the system with hundreds of test cases and many complex source codes. The queries that we used to test covered a range of possibilities. From the most basic, to boundary cases, to where the thing to be returned could not be found in any of the tables.

## Examples of Test Cases

### Unit Testing

The aim of unit testing, would be to discover any logical errors present in the code. This allowed us to pinpoint specific errors, saving us the hassle of running into such errors during further stages of testing. Unit testing was done by manually inserting values and asserting that the function outputs were as expected.

The following example illustrates the unit testing of the ListTable component.

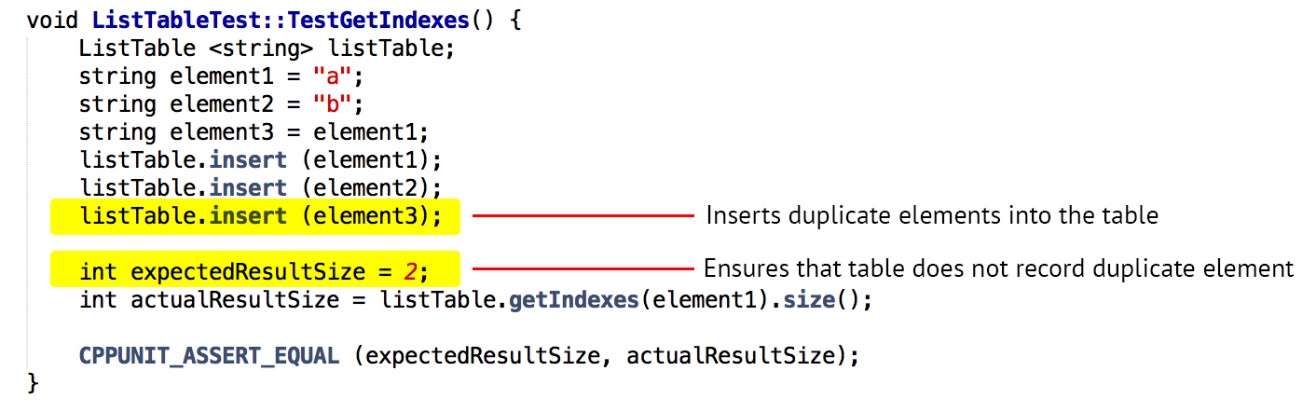


Figure 7.1: Unit Test Example

### Integration Testing

Many of the components work well when tested individually. However when they are integrated with other components, unforeseen errors may appear. This is the aim of integration testing, to identify such errors between the interactions of components. Such errors, in our case, were attributed to the fact thtt the PKB, Parser and PQL were written by different members. Hence the components had slightly different methods of implementation and different expected inputs and outputs. During integration testing, these flaws between the components became apparent to us. The Parser parses the simple source code, which is provided as the input file, and the PKB is built from it. In the following example, the integration between these components, including the Design Extractor, is illustrated when testing the Calls relationship.

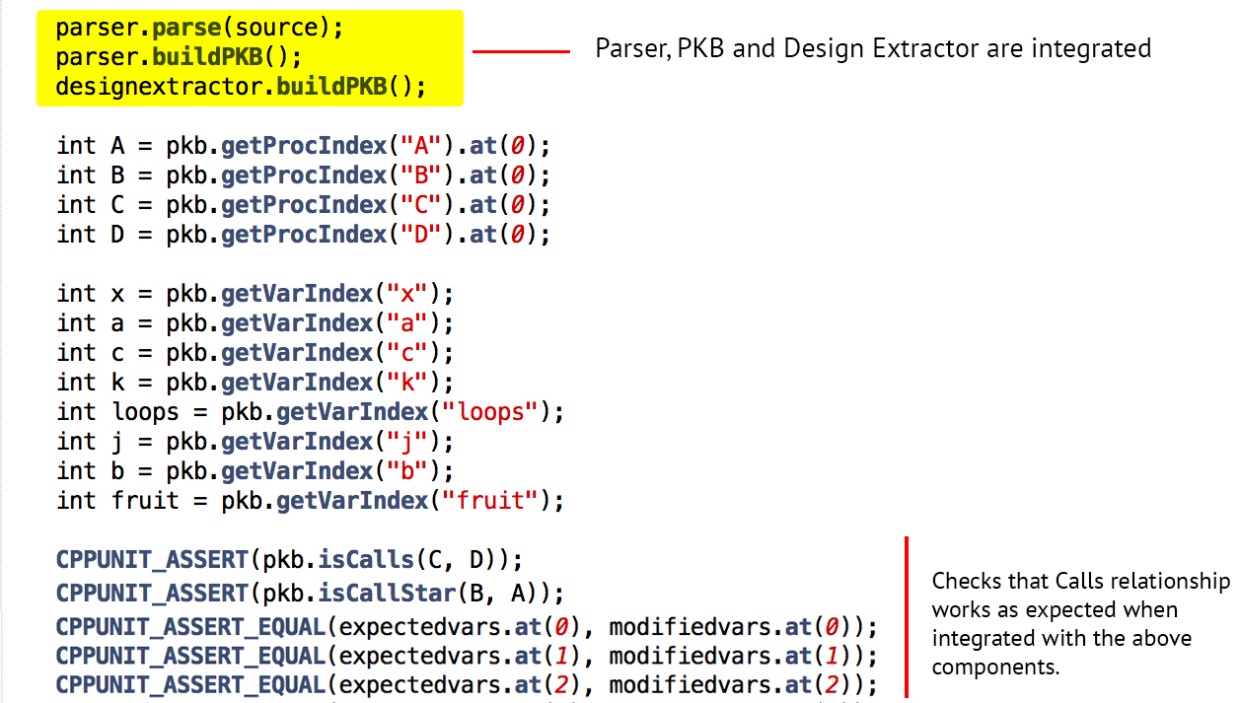


Figure 7.2: Integration Test Example

### Validation Testing

Validation testing, also known as System testing, is crucial to test the SPA system as a whole. There are many logical loopholes that might have been missed during unit and system testing, which can be spotted during extensive system testing. Thus, it is important to have a variety of test cases and source codes. The aim, in our case, was to ensure that the system can handle queries of multiple complexities and parse hundreds of lines of source code, including those haphazardly formatted.

The following Simple source code aims to test complex pattern queries by using multiple expressions and variables, the heavy use of brackets is also to be noted.

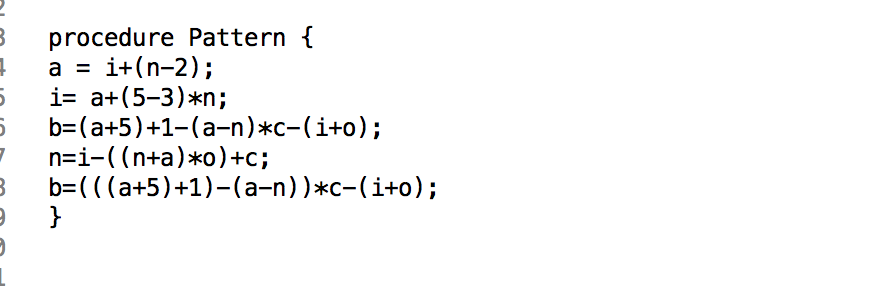


Figure 7.3: Simple Source for Validation Testing

The figure below shows two query examples. The first query tests the combination of multiple relationship clauses. This tests that the evaluated results for the different clauses are merged correctly to output only the correct answers that satisfy the whole query. This is essential for evaluation of any query that has more than one clause. The second query focuses on selecting variables by their attributes, such as ‘stmt#’ and ‘varName’. This is a commonly used selection method in queries, hence it’s essential to ensure that all such possible attribute selections have been addressed by the system.

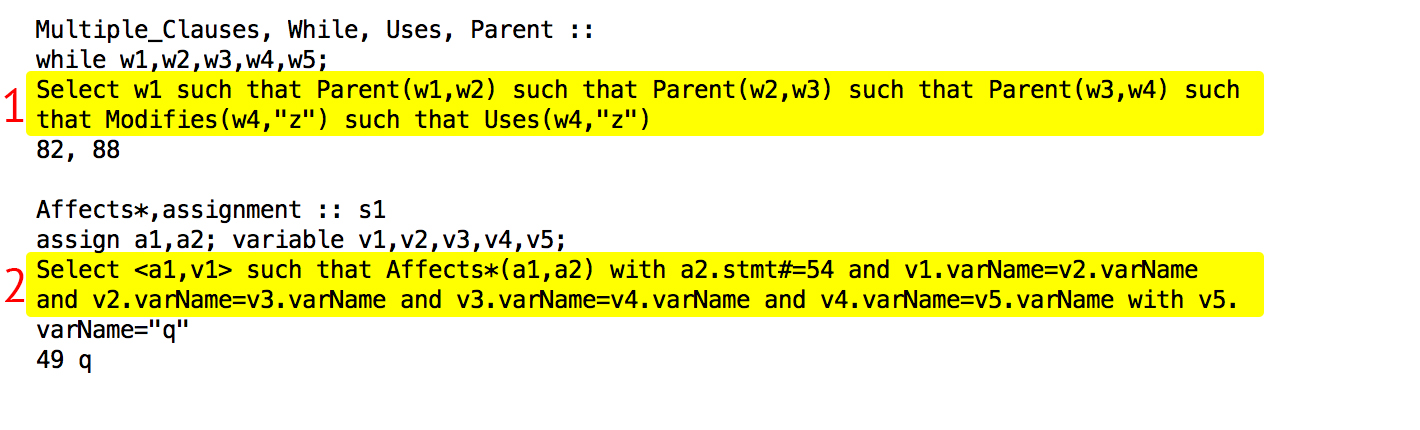


Figure 7.4: Queries for Validation testing

# Discussion

## Possible Improvements to Project

## Project Experience

### Team Work

### Incremental Development

### SPA Complexity & Proj solution

### Things learnt

## Tools used

-usefulness of recommended tools

-other tools used

…..

# Appendix A: Abstract PKB API

# Appendix B: Comments on Handbook

Overall, we found the handbook very helpful in giving us ideas for the implementation of the SPA. One thing that could have been better was if there had been more examples and perhaps some sample exercises for us to practice our concepts on. The extensions such as Contains and Siblings could also be explained briefly in the handbook instead of introducing them later on. This means we would have rough idea of how these relationships work beforehand and when it is time to consider whether or not to implement them, we can focus on finding the best way to integrate them rather than spending time understanding and grasping the new concepts of these relationships, which reduces crucial time which could be spent on the actual implementation.