

**National University of Singapore**

**School of Computing**

**CS3202: Software Engineering Project II**

**TEAM 05: Flying Cockroach**

Semester 1, AY2014/2015

|  |  |  |  |
| --- | --- | --- | --- |
| **Matriculation Number** | **HP Number** | **Student Name** | **Email** |
| **Group-PQL:** | | | |
| A0099214B | 8518 2707 | Adinda Ayu Savitri | savitri.adinda@gmail.com |
| A0098139R | 9082 0864 | Hisyam Nursaid Indrakesuma | indrakesuma.hisyam@gmail.com |
| A0103494J | 9620 7018 | Lacie Fan Yuxin | lacie.jolene.fan@gmail.com |
| **Group-PKB:** | | | |
| A0101286N | 9833 2474 | Ipsita Mohapatra | ipsita@nus.edu.sg |
| A0080415N | 9148 6248 | Steven Kester Yuwono | a0080415@nus.edu.sg |
| A0099768Y | 9178 6540 | Yohanes Lim | yohaneslim93@gmail.com |

Consultation Day/Hour: Monday 6-6.30pm

Contents

[1. SPA 3](#_Toc400895570)

[1.1. Architecture 3](#_Toc400895571)

[1.2. Development Plan 4](#_Toc400895572)

[1.2.1. For Whole Project 4](#_Toc400895573)

[1.2.2. For Iteration 3 5](#_Toc400895574)

[2. Components 6](#_Toc400895575)

[2.1. Code Parser 6](#_Toc400895576)

[2.2. PKB 7](#_Toc400895577)

[2.3. Design Extractor 8](#_Toc400895578)

[2.4. Query Processor 9](#_Toc400895579)

[2.4.1. Query Processor 9](#_Toc400895580)

[2.4.2. Query Parser 9](#_Toc400895581)

[2.4.3. Query Evaluator 9](#_Toc400895582)

[3. Testing 10](#_Toc400895583)

[3.1. Testing Plan For Iteration 3 11](#_Toc400895584)

[3.2. System Testing 11](#_Toc400895585)

[4. API 11](#_Toc400895586)

# 1. SPA

Static Program Analyser (SPA) is a program to answer queries about an input SIMPLE program. In this report, we will be describing the design and implementation decisions made during the development of the basic SPA and any bonus features.

## Architecture

The architecture consists of 3 main components: the Code Parser, the PKB and the Query Processor. Code Parser parses the code and stores design abstractions in each of the 8 tables in the PKB. After Query Parser has parsed the query, the Query Evaluator consults the PKB API to answer queries.

Both the Code Parser and the Query Processor are dependent on PKB but not dependent on each other. This ensures low coupling between components within SPA. At the same time there is high cohesion within each component, as described in the subsequent parts.



Figure 1

## 1.2. Development Plan

### 1.2.1. For Whole Project

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Iteration 1** | | | | | **Iteration 2** | | | |
| Team member | **Implement Calls in PKB** | **Implement Modifies and Uses in PKB** | **Extend Query Parser to support with clause** | **Extend Query Evaluator to support with clause** | **Write system test cases for the enhancement** | **Implement Next in PKB** | **Extend Query Parser to support multiple clauses** | **Extend Query Evaluator to support multiple clauses** | **Write system test cases for the enhancement** |
| Adinda |  |  |  |  |  |  |  |  |  |
| Lacie |  |  |  |  |  |  |  |  |  |
| Hisyam |  |  |  |  |  |  |  |  |  |
| Steven |  |  |  |  |  |  |  |  |  |
| Ipsita |  |  |  |  |  |  |  |  |  |
| Yohanes |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Iteration 3** | | | |
| Team member | **Implement Affects in PKB** | **Extend Query Parser to support Affects and tuple results** | **Extend Query Evaluator to support Affects and tuple results** | **Write system test cases for the enhancement** |
| Adinda |  |  |  |  |
| Lacie |  |  |  |  |
| Hisyam |  |  |  |  |
| Steven |  |  |  |  |
| Ipsita |  |  |  |  |
| Yohanes |  |  |  |  |

### 1.2.2. For Iteration 3

# 2. Components

## 2.1. Code Parser

(Same as Before)

## 2.2. PKB

(Same as Before)

## 2.3. Design Extractor

(Iteration 3 Work)

The Design Extractor extracts the nextPair relationships using the Next relationships stored in PKB. It then stores these nextPair relationships in the PKB for future use by the Query Evaluator. By implementing a nextPair relationship, we are able to do the NextStar query faster in the Query Evaluator, without having to call the getNext() relationship multiple times when the program lines are sequentially connected in the CFG.   
  
**Extracting “NextPair” relationship**

NextPair relationship is defined as a follows:  
getNextPair() should return a vector of pairs where each pair denotes a range of program lines. So if getNextPair(x) returns [(a, b), (c, d)], then there is a path in the CFG from x to any program line between a and b (inclusive). There is also a path in the CFG from x to any program line between c and d (inclusive).

E.g. if the Next table is as follows:  
…  
3 🡪 4  
4 🡪 5, 7  
5 🡪 6  
6 🡪 9  
…

Then, the following nextPair relationships should be returned.   
getNext(3): [(4, 6)]  
getNext(4): [(5, 6), (7, 12)]  
getNext(5): [(6, 6)]  
getNext(6): [(9, 12)]

## 2.4. Query Processor

Query Processor consists of three parts: Query Processor (controller), Query Parser, and Query Evaluator.

### 2.4.1. Query Processor

Query Processor is a façade class for the whole component. Its responsibilities include:

1. Query Processor calls QueryParser to create a Query object from the given query string.
2. Query Processor then passes the Query object to the QueryEvaluator.
3. Query Evaluator will compute all necessary relations and return the results in the form of a list of integers.
4. Query Processor transforms the result into the correct display format and returns the answer to the user.

### 2.4.2. Query Parser

**Query Validation**

//no change from iteration 2 report

….

assign a; while w; Select a such that Follows(w, a) pattern a (“x”, \_”x+y”\_)

….

**Query Parsing**

The parser processes the *selectStatement* vector from the earlier. The *selectStatement* vector will be processed to construct a Query object with the following structure.

|  |
| --- |
| **Query** |
| *vector<string>* selected-synonym  *vector<Relationship>* relationship-vector  *map<string, synonym-type>* synonym-table |

Since each query can contain many select clauses, the selected synonyms and clauses are stored inside a vector for scalability purposes. All the synonyms present in the *selected-synonym* vector will be detected and validated once again on whether they have been declared earlier. If it is not declared, the query is invalid and **QueryParser** will indicate and return invalid. The string vector will then be stored in a **Query** class.

The synonym map that was created earlier will also be included in the **Query** object. Both such-that and pattern clauses will be stored as another object, **Relationship**, as the following.

|  |
| --- |
| **Relationship** |
| *enum* relationship-type  *enum* token-type  *string* token1  *string* token2  *TokenType* token1-type  *TokenType* token2-type  *string* pattern-synonym |

**Relationship** arguments will also be semantically checked whether they are valid. For example, both arguments in Follows clause have to be a statement (i.e. stmt, while, if, assign, call). If the arguments contain a constant synonym (e.g. constanct c; while w; Select w such that Follows (c,w), then **QueryParser** will detect the error and return invalid.

The types of the tokens/arguments are detected by **QueryParser** and store them in the **Relationship** class. For example, if the token is “Second”, then its type will be IDENTIFIER, if the token is 1, then its type will be INTEGER, and if the token is w, then its type will be SYNONYM.

From the valid example above, the select-statement vector will be processed to produce the following.

|  |  |
| --- | --- |
| **Query** | |
| selected-synonym | a |
| relationships | [rel1, rel2] |
| synonym-table | map1 |

|  |  |
| --- | --- |
| **rel1** | |
| relationship-type | FOLLOWS |
| token1 | w |
| token2 | a |
| token1-type | SYNONYM |
| token2-type | SYNONYM |

|  |  |
| --- | --- |
| **rel2** | |
| relationship-type | PATTERN |
| token1 | “x” |
| token2 | \_”x+y”\_ |
| token1-type | IDENTIFIER |
| token2-type | UNDERSCOREEXPR |
| pattern-synonym | a |

|  |  |
| --- | --- |
| **map** | |
| Synonym | Type |
| a | ASSIGN |
| w | WHILE |
| BOOLEAN | BOOLEAN |

When the controller calls the parsing function, the function will return a query object. This object will then be passed to query evaluator.

For with-clause, Query Parser with detect the conditions whether they are valid.

For example:

procedure p,q; Select q such that Calls (p,q) with p.procName=”Second”

p.procName = ”Second” will be parsed into two parts, left-hand-side and right-hand-side. Right-hand-side includes “Second” and will be stored in token2/argument-2 in a relationship object. Left-hand-side includes p.procName and will be stored in token1/argument-1 if token is valid. Synonym p will be checked against the map whether it exists. Since the attribute name is procName, it will also be checked on whether it is of type procedure.

Left-hand-side and right-hand-side will be checked on whether they are of the same type, either character strings or integers. If they are of different type, return invalid.

If it passes both validations, then the query is valid and stored in the relationship below:

|  |  |
| --- | --- |
| **rel2** | |
| relationship-type | WITH |
| token1 | p |
| token2 | “Second” |
| token1-type | SYNONYM |
| token2-type | IDENTIFIER |

“procName” is not stored because it is known that a synonym of type procedure can only have “procName” as its attribute name.

For multiple clauses (such that, pattern, with), Query Parser works by keeping the previous clause keyword (e.g. “pattern”, or “with”), and use it to detect the clause type when it encounters “and”.

For example:

assign a; Select a pattern a(\_,\_) and a(“x”,\_”x+y”\_);

Parser will detect the first clause and store in the relationship class accordingly

|  |  |
| --- | --- |
| **rel1** | |
| relationship-type | PATTERN |
| token1 | \_ |
| token2 | \_ |
| token1-type | UNDERSCORE |
| token2-type | UNDERSCORE |
| pattern-synonym | a |

It also keeps track of the last relationship-type. Therefore, when Query Parser reaches the word “and”, it knows that it will be parsing a pattern clause again, translating the “and” keyword into “pattern”. Query Parser will then validate and parse accordingly with respect of the clause type, in this case it is “pattern”.

|  |  |
| --- | --- |
| **rel2** | |
| relationship-type | PATTERN |
| token1 | “x” |
| token2 | \_”x+y”\_ |
| token1-type | IDENTIFIER |
| token2-type | UNDERSCOREEXPR |
| pattern-synonym | a |

Both relationships will then be stored in a **Query** class and passed to **QueryEvaluator**.

### 2.4.3. Query Evaluator

# 3. Testing

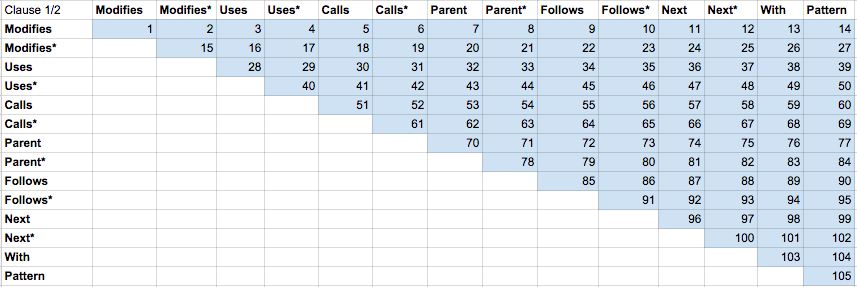
We did testing on 3 different levels, namely unit testing (using CPPUnit), integration testing (using CPPUnit) and system testing (using AutoTester). Unit Testing was done while coding the components, while integration testing was done between SIMPLE program parser and PKB and between PKB and Query component.

From the testing experience in this project, we realised the need for timely and consistent unit, integration and system testing. By testing individual components early, we detect bugs earlier in the project’s lifetime, thus, saving us time towards the end of the project. We also did regression testing by reusing our unit tests and system tests. This helped us to quickly identify bugs that could have been introduced while we were trying to solve other bugs.

## 

## Testing Plan For Iteration 3

## System Testing



# 4. API

Please view our Doxygen at:

[www.comp.nus.edu.sg/~kester/CS3202](http://www.comp.nus.edu.sg/~kester/CS3202)