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

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.

1.1. 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.

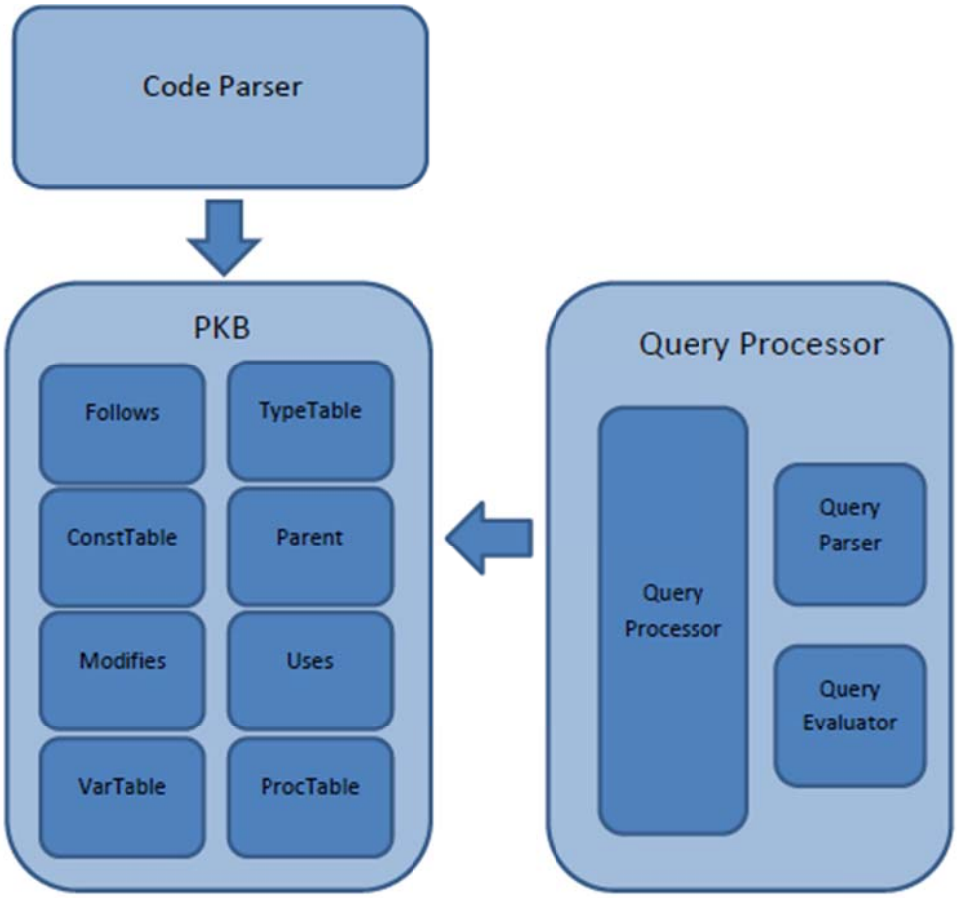


Figure 1

1.2. INTERACTION

CodeParser works by evaluating each line of the given source code. It creates AST Node, set the pointers accordingly; set the tables and the appropriate databases in PKB.

The tables in PKB will then be used by Query Evaluator to answer queries. Below, we see the interaction of the CodeParser with tables in PKB. For clarity, the interaction of the CodeParser with each table is shown separately, but in reality, the CodeParser interacts only with a set of methods provided by the PKB. (e.g. setToParent())

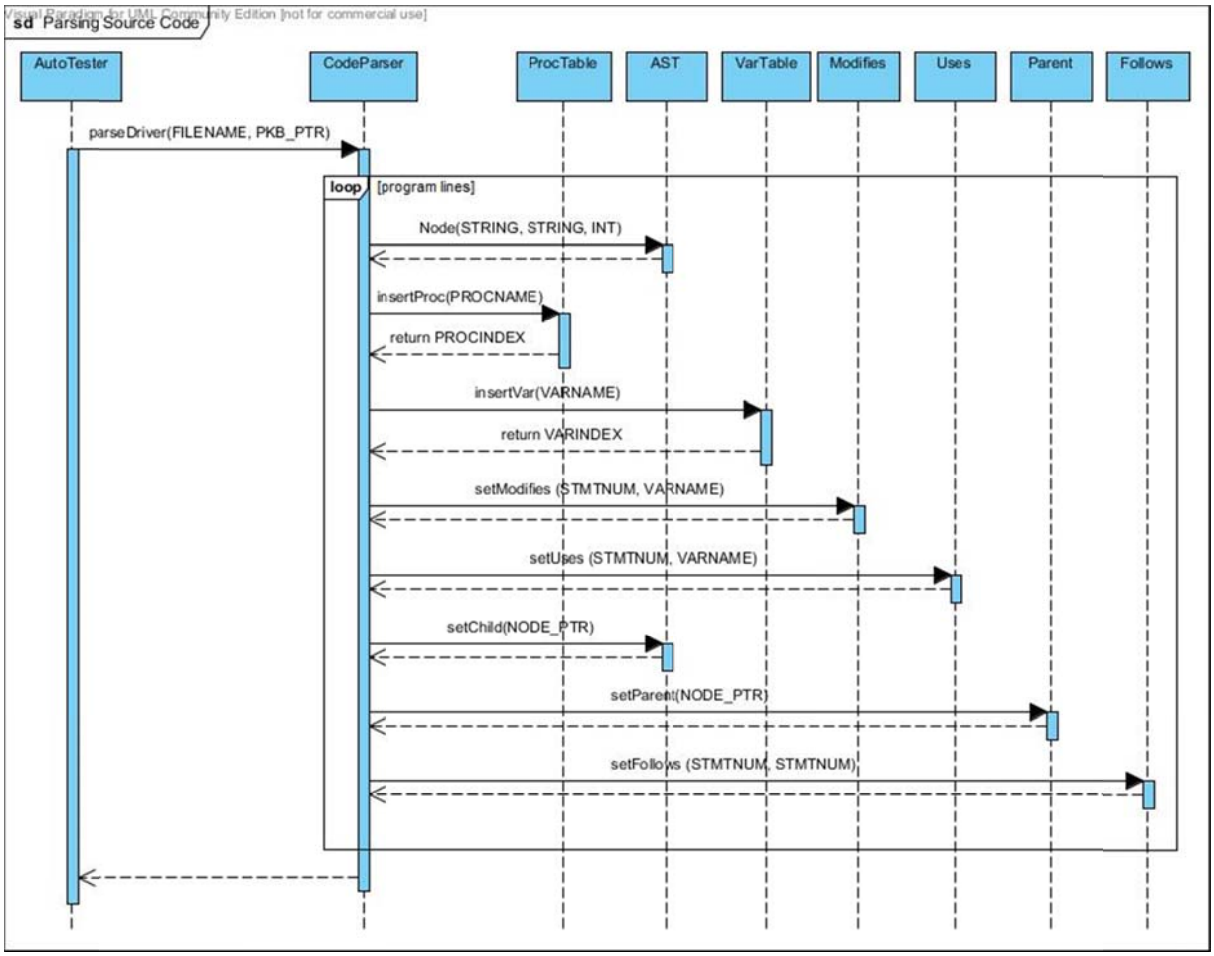


Figure 2

Figure 3 shows the sequence diagram for the query evaluation process. This diagram is useful in demarcating the responsibilities of each PQL group member. For example, QueryEvaluator directly assumes that the Query it receives is valid and syntactically correct. Therefore, it is the responsibility of QueryParser to validate each query before passing it to the evaluator.

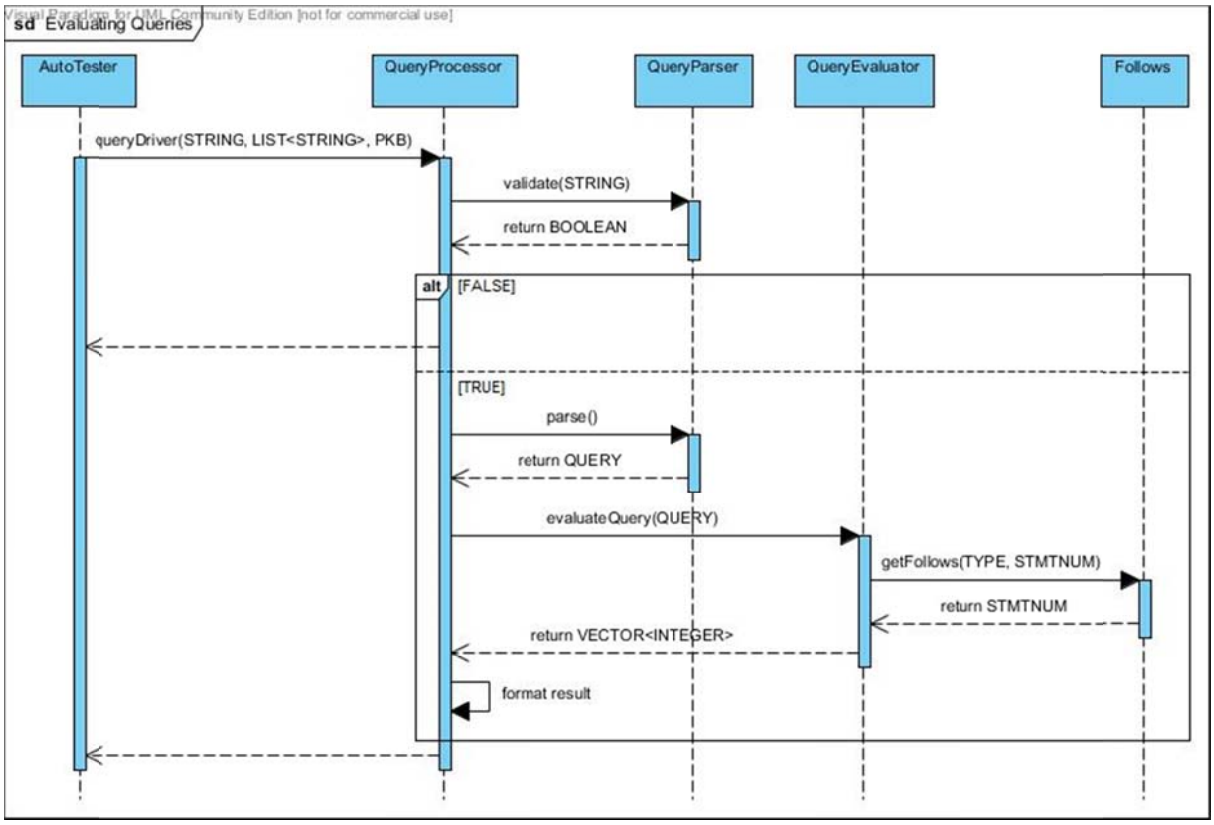


Figure 3

This diagram also helps to keep track of the dependencies between components. This is especially useful during debugging process of integration testing. When QueryProcessor fails to return the correct result, the team knows that the errors could come from at least one of three places, i.e. QueryParser, QueryEvaluator, and PKB.

1.3. Development Plan

1.3.1. 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				

1.3.2. For Iteration 1

	Iteration 1									
Team member	**Optimize internal data structure for PKB**	**Implement Information hiding in PKB**	**Implement Modifies (proc), Uses (proc), and Calls relations in PKB**	**Implement Design Extractor functions for extracting relationships**	**Extend Query Parser to support with clause**	**Come up with design alternative for optimization of Query Evaluator**	**Redesign internal structure of Query Evaluator**	**Extend Query Evaluator to support with clause**	**Enhance system testing mechanism**	**Write system test cases for iteration 1**
Adinda										
Lacie										
Hisyam										
Steven										
Ipsita										
Yohanes										

1.3.3. For Iteration 2

	Iteration 2							
Team member	**Implement Next relationship**	**Revamp PKB data structure for better performance**	**Implement CFG to be stored in PKB**	**Extract Next relationship from CFG and store in PKB**	**Extend Query Parser to support multiple clauses**	**Extend Query Evaluator to support Next and Next***	**Extend Query Evaluator to support multiple clauses**	**Write system test case for iteration 2**
Adinda								
Lacie								
Hisyam								
Steven								
Ipsita								
Yohanes								

2. COMPONENTS

2.1. CODE PARSER

Code Parser's main functions are to read in the source code, build the AST, and set the tables (VarTable, ProcTable, TypeTable, Follows, Parent, Modifies, Uses) in PKB according to the input source code.

To build the AST, Code Parser depends on the implementation of node, which is the node structure being used to build AST. A node can have many children, therefore, we decided to use NODE_PTR_LIST for dynamic storage of the children pointers, rather than array with fixed size.

Code Parser keeps track of every relevant parent of each node being built, by storing pointers to their parents. For example when there is a while statement, a pointer to the while statement will be stored and then link all the children to their parent node.

Code Parser works by tokenizing the source code as string line by line, and then detecting the tokens for each line, and generating the types, program lines, and setting the tables accordingly.

For an assignment statement, Code Parser will convert the expression from infix to postfix, and then create the expression tree.

Code Parser does its validation by keeping track of the curly brackets (i.e. "{" and "}"). It keeps track of the number currently present open curly bracket, "{". When Code Parser encounters an open curly bracket, it will push it to a stack. When it encounters a closed curly bracket, it will pop from the stack.

When Code Parser reaches the end of the source code, it will return invalid if the stack is not empty, or if Code Parser is trying to pop from an empty stack. It means there is a mismatch in the number of curly brackets.

When the stack is empty, Code Parser will accept a line which defines a procedure. If it encounters any other statement while the stack is empty, it will return invalid.

Example:

Source code (source1.txt):

```
procedure Mini {
A1 = 29;
a1 = 31;
i = 51; }
```

The following describes how the Code Parser parses the above SIMPLE program:

1. Code Parser starts reading the source code at line 1, it will check whether the stack is empty. If the stack is empty, it will be expecting a procedure declaration.
2. It then parses procedure Mini, creates an ASTNode, sets it as root, and pushes the curly bracket "{" into the stack. Insert "Mini" into the ProcTable.
3. At line 2, Code Parser tokenizes and checks the type of statement. Since it starts with a variable, it detects the statement as an assignment statement.
4. Code Parser will check if there exists a semicolon at the end of the line because it is compulsory to have a semi colon at the end of an assignment statement. If it exists, it will create a node containing "=", and link "A1" as the first child. The expression on the right hand side will be converted into a postfix expression, and then build the expression tree.
5. Code Parser will link the root of the expression tree as the second child of "=", and then link the "=" to its parent, which in this case, is "procedure Mini".
6. Code Parser will also set the VarTable, Modifies, Uses, Follows, and Parent accordingly, in this case, it will set Modifies (line 1, and variable A1), and insert A1 into the VarTable.
7. At line 3, it detects that it is an assignment statement, and repeat step 4 to 6.
8. At line 4, it detects that it is an assignment statement, and repeat step 4 to 6. In addition, Code Parser detects a closed curly bracket. Therefore, Code Parser will pop the stack.
9. End of source code is reached. It will now check whether the stack is empty.
10. Since the stack is empty, and there is no violation of the rules stated earlier. Code Parser has built the AST successfully and stored the design abstractions in the relevant tables.

CodeParser's Unit testing is done by checking the content of each table, whether it has set the tables properly, and by checking the contents of each node in the AST, whether it matches the expected AST.

2.2. PKB

2.2.1. Design Decisions

PKB is implemented using the singleton pattern. One instance of PKB will be initialised during the construction phase of the UI (which is AutoTester). Afterwards, we will only pass the PKB pointer to other components which need to alter the PKB or call the PKB's methods. This is to ensure that other components are always editing or accessing the same PKB object. Using the same rationale, all the sub-components of PKB (VarTable, ProcTable, ConstTable, Follows, Parent, Modifies and Uses) are singleton classes and only their pointers are passed around.

To make things clearer during communication, we used some new definitions:

- Typedef int
  - STMTNUM: for statement number
  - VARINDEX: for variable index
  - CONSTVALUE: for constant value
- Typedef string
  - VARNAME: for variable name
  - PROCNAME: for procedure name
- Typedef Enum SynType {ASSIGN, IF, WHILE, STMT, BOOLEAN, CALL, VARIABLE, CONSTANT, PROGLINE, INVALID} TYPE
  - to discern the type of each statement number
- Typedef pair<PROCINDEX,STMTNUM> CALLSPAIR;
  - To store the index of the procedure being called and the statement number where the call is invoked

The data structure used for the tables and the relationships is unordered_map as listed below.

PKB – Design Abstractions	
Tables	**Data Structures**
ProcTable	unordered_map <PROCINDEX,PROCNAME>
ConstTable	unordered_map <CONSTINDEX,CONSTVALUE>
TypeTable	unordered_map<STMTNUM,SynType>
VarTable	unordered_map<VARINDEX,VARNAME>
Relationships	**Data Structures**
Follows	unordered_map<STMTNUM,STMTNUM>
Parent	unordered_map<STMTNUM,vector<STMTNUM>> & unordered_map<STMTNUM,STMTNUM>
Uses	unordered_map<STMTNUM,vector<VARINDEX>>
Modifies	unordered_map<STMTNUM,vector<VARINDEX>>
Calls	unordered_map<PROCINDEX, vector<CALLSPAIR>>
Next	unordered_map<STMTNUM,vector< STMTNUM >>

In deciding the data structures to be used in PKB, our main consideration was speed – in inserting and searching. In making our design decision, we made a comparison between vector, ordered map and unordered map.

	Vector of Size N	**Ordered Map of Size N**	**Unordered Map of Size N**
Insert	O(1) or O(N) if need resizing	O(log N)	O(1) (Average case)
Search	O(1)	O(log N)	O(1) (Average case)

We can see that unordered map is the fastest of the three which explains our decision to use it. The main drawback is that it will consume more memory but we prioritize speed over memory consumed.

2.2.2. Interaction with other Components

The PKB mainly interacts with the Code Parser and the Query Evaluator.

Interaction with Code Parser

After PKB is initialised, the PKB pointer is passed to Code Parser to fill in the tables and relationships into the PKB.

For example, when the Code Parser calls insertVar("x"), the following is done:

1. Check whether the given variable, "x", exists in the table by iterating through the element in the table one by one.
2. If yes, we will simply just return the index.
3. Otherwise, insert the element at the back of the table and return the index (table size - 1) of the variable.

For example, when the Code Parser calls setUses(12,"x")the following is done:

1. Get the variable index of "x" from VarTable.
2. If the variable index is -1, it means that there is no such variable and thus the method will terminate.
3. If the variable index is more than -1, it means that the variable exists, therefore insert into the uses table at key 12, the value of variable index "x".

Interaction with Design Extractor

The PKB pointer is then passed to Design Extractor to extract more design abstractions and relationships that were not picked up by the CodeParser during the parsing stage. In short, the Design Extractor sets Modifies and Uses relationships for procedures and calls statements as well as the Next relationship for program lines.

Interaction with Query Evaluator

PKB pointer is then passed to Query Processor so that Query Evaluator can call the public API provided by PKB. Query Processor will need to get the tables or relationships that it needs first and only then it can call the corresponding API that it needs.

For example, if we want to call getChildren(WHILE, CALL), Query Processor needs to get parent from PKB and then calls parent->getChildren(WHILE, CALL). When it is called, it will result a list of STMTNUM x such that for each x, Parent(CALL, x) holds and x is a WHILE statement. If there exists no such statement x, an error code is returned. The steps are as follows:

1. Get parent pointer from PKB using getParent()
2. Calls the method getChildren(WHILE,CALL) from parent
3. Iterate the children table inside parent from beginning to end. The index of the vector, i, will indicate the statement number of the children.
4. Get j, the value of the vector at the specified index which is the statement number of the parent
5. If j is -1, continue with the next index from step 1.
6. Use isType(WHILE, j) to check the type of j from the TypeTable to see whether it is of type WHILE or not.
7. If not, continue with the next index from step 1.
8. If yes, ise isType(CALL, i) to check the type of I from the TypeTable to see whether it is of type CALL or not.
9. If not, continue with the next index from step 1.
10. If yes, push i into the vector of answer and continue with the next index from step 1.
11. After iterating through the whole children map, return the vector of answer.
12. If the vector of answer is empty, return the vector with -1 as the only element.

2.3. Design Extractor

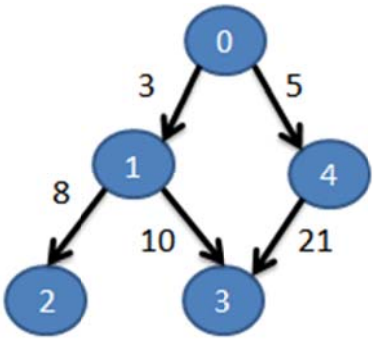
The main role of the design extractor is to extract relationships about the SIMPLE program that could not be extracted in the one-time parsing done by the Code Parser. This includes:

- Extracting information about Modifies and Uses for procedures and for program lines that are calls statements
- Building the Control Flow Graph (CFG) from the AST and subsequently storing it in the PKB and storing the Next relationship in the PKB

2.3.1. Extracting Relationships

The following shows the steps required to extract relationships like Modifies and Uses for multiple procedures in a single SIMPLE program:

1. Obtain the Calls Table from the PKB. An example of the calls Table is shown below where procedure with index 0 calls procedure with index 1 from program line 3 and procedure with index 4 calls procedure with index 3 from program line 21.
   0 → (1, 3) (4, 5)
   1 → (2, 8) (3, 10)
   2 →
   3 →
   4 → (3, 21)
   This translates into a Calls Tree as follows where the nodes represent procedure indices and the edges represent the program line at which there is a calls statement.



2. Run Depth First Search (DFS) on the Calls table in order to obtain a topological sort order of the procedure indices in a queue (below) which starts on the left. Each Queue contains a procedure index and a vector of program lines in which the procedure is called either directly or indirectly.

(2, [3, 8])	(3, [3, 10])	(1, [3])	(3, [5, 21])	(4, [5])	(0, [ ])

3. Starting from the head of the queue, find all the variables that are modified and all the variables that are used in the procedure. For each of the program lines in the Queue Item, set these program lines (p) to modify and use the respective variables.

For each of the program lines (p), if they are contained in another container statement (c), then set these program lines (c) to modify and use the respective variables too.

2.3.2. Building CFG

Given the AST, the following shows the pseudo code to build the CFG. Then, the CFG is traversed and the Next relationships are stored in the PKB's Next table.

Maintain a currCFGNode pointer.

BuildCFG()

Create CFG Root with progLine = 0
Iteratively traverse each of the type = procedure nodes in AST

Update the currASTNode to stmtLst node under the procedure node
Update the currCFGNode to point to the rootCFGNode
CreateCFGForStmtLst(currASTNode)

CreateCFGForStmtLst(ASTNode) [where ASTNode points to the :stmtLst node]

Iteratively traverse each of the children AST nodes of the currASTNode

If type==ASSIGN

CreateCFGForAssign(currASTNode.progLine)

Else if type == CALL

CreateCFGForCall(currASTNode.progLine)

Else if type == WHILE

CreateCFGForWhile(currASTNode.getChild())

Else if type == CALL

CreateCFGForIf(currASTNode.getChild())

CreateCFGForAssign(progLine)

CreateNewNodeAndAttachToCFG()

CreateCFGForCall(progLine) [omitted because it is the same as CreateCFGForAssign]

CreateCFGForWhile(ASTNode) [where ASTNode points to the :stmtLst node]

CreateNewNodeAndAttachToCFG()
Save the toNode to be the currCFGNode
createCFGForStmtLst(ASTnode.getChild())
Find the fromNode in the CFG using DFS.
Create an arrow in the CFG from fromNode to toNode.
Update currCFGNode to toNode

CreateCFGForIf(ASTNode) [where ASTNode points to the :stmtLst node]

CreateNewNodeAndAttachToCFG()
For then:stmtLst, createCFGForStmtLst()
Store pointer to currCFGNode in a vector called leafNodes
For else:stmtLst, createCFGForStmtLst()
Store pointer to currCFGNode in a vector called leafNodes

Create an “end-of-if” node with progLine = -1
For each of the CFGNodes in leafNodes, make it the parent of the “end-of-if” node
Update the currCFGNode to point to the “end-of-if” node

CreateNewNodeAndAttachToCFG()

Create new node and attach to existing CFG

SetNextRelationship(currCFGNode)

Mark the currCFGNode as visited
For each child of the currCFGNode

toNode = child
If the toNode.progLine != -1 and the fromNode.progLine != -1

Set the NEXT relationship

If toNode.progLine == -1

Look for the next child which has child.progLine != -1
If such a node exists, set the NEXT relationship

If toNode is not visited

SetNextRelationship(toNode)

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 parser has two major functionalities: query validation and query parsing, and they are implemented as functions in the **QueryParser** class. The controller calls query validator to check if the given query is syntactically correct. If it is, query controller will then parse the query by calling the query parser.

Query Validation

Query validation is done using regular expression method using the grammar rules written in the handbook.

We have considered to validate while parsing the query (tokens), but decided to use regular expression to validate the query first then parse, because of the following reasons:

1. The code is much neater and simpler
2. It is faster to detect errors rather than parsing and validating (especially if the error is towards the end of the query)
3. Query parsing becomes simpler if we already know the exact possible format of the string query that needs to be parsed.

All types of queries have been defined in a static string, following the grammar rules in the textbook. The strings are then used to validate the queries using regular expression, where the definitions earlier are used. It is very convenient because the grammar rules in the book is close to a regex grammar rules.

As an example, consider this valid query

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

Query validator will first break down the query into statements, separated by semicolon. The query above will be broken down into three statements:

1. assign a
2. while w
3. Select a such that Follows(w, a) pattern a (“x”, _”x+y”_)

The validator will then use regular expression to check the validity of the statement and retrieve the tokens. Valid declaration statements will be converted into a map with the synonym as the key and its type as the value. This map is called the synonym map and will be used later by the parsing function. In this example, the synonym-map will look as follows:

Synonym	Type
a	ASSIGN
w	WHILE
BOOLEAN	BOOLEAN

This map enables easy look up when the query evaluator evaluates the query. Note that the BOOLEAN type always exists in the synonym table because user can use “BOOLEAN” in his select statement without any declaration.

Select statement will be broken down into such-that and/or pattern clauses, whose parameters will be checked against the grammar rule.

Consider statement number 3 from the example above. Query validator will use regular expression to check the validity of the statement and retrieve the appropriate tokens.

If the regular expression matching fails, (for example, the number of arguments in the clause is not exactly two) the validator will instantly terminate and declare the query invalid. In the case where statement is valid, all the tokens from the select statement will be stored in a vector, *selectStatement*. This vector will be accessed by the parsing function later on. For efficiency, the *selectStatement* vector will only contain relevant tokens from the statement. Therefore, the unnecessary syntactic punctuation will be removed.

From the example above, the value of *selectStatement* will be:

selectStatement[]

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
string selected-synonym vector<Relationship> **relationships** *map* synonym-map

The selected synonym, in this example is 'a', will be stored inside a string in the Query object. The synonym map that was created earlier will also be included in the Query object. Both the such-that and pattern clauses will be stored as another object, Relationship, as the following.

Relationship
enum relationship-type *string* argument-1 *string* argument-2 *string* pattern-synonym

Since each query can contain many select clauses, these clauses are stored inside a vector for scalability purposes. All the synonyms present in the *selectStatement* 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. From the 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
argument-1	w
argument-2	a

rel2	
relationship-type	PATTERN
argument-1	“x”
argument-2	_”x+y”_
pattern-synonym	a

map1	
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 token/argument-2 in a relationship object.

Left-hand-side includes p.procName and will be stored in token/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
argument-1	p
argument-2	“Second”

“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
argument-1	_
argument-2	_
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
argument-1	“x”
argument-2	_”x+y”_
pattern-synonym	a

Both Relationship will then be stored in a Query class and passed to Query Evaluator.

2.4.3. Query Evaluator

Revamp of Basic Query Evaluation(BQE)

Creation of Pair Class: To ease storing of answers, we created a Pair class to store pairs of answers for Relationship objects

To facilitate the interaction between multiple Relationship objects, we created 3 new static variables

- Linkages: an unordered map of string keys and vector<int> values. If a Relationship object has a synonym as parameter, then its index will be added into the vector of that particular syn
    i. E.g in Follows(a, 1) A pair of <a, vector that contains relIndex of Follows> will be inserted into linkages
- RelAns: an unordered map of int keys and vector<Pair> values that keeps track of each set of Pair of answers that are evaluated from the Relationship object
    i. E.g if set<1,2,3> are answers to Follows(a,1) then relAns will have vector<Pair>: <1,1><2,1><3,1>
- RelParmameters: an unordered map that keeps track of the tokens(or the parameters) of each Relationship object
    i. E.g in Follows(a,1) both a and 1 are tokens of the object

The above implementation ensures accuracy and flexibility when multiple tuples and clauses are introduced

Current Implementation for Basic Query Evaluation (BQE)

1. evaluateNext

To implement Next, we first check whether the tokens are alphabets. If they are, we check whether they exist in linkages. If they exist, we will retrieve the previously evaluated answers of the tokens and substitute it into functions of PKB to find out the next or previous programme line depending on the query.

2. evaluateNextStar

To implement Next*, we follow the same method as evaluateNext. To carry out the * action, we wrote a recursiveNext function which will find all the next program lines of the first token and then find all the next program lines of the program lines of the first token.

Analysis of the current BQE design decisions

1. Ease of Changing/Flexibility

The current implementation of BQE is flexible as if new relationship types are added, only minimal functions are required to be added before evaluateQuery will be able to evaluate it and merge the results with the rest of the answers.

2. Reusability

As the functions are created according to the type of relationships that can be queried, BQE has a high reusability. If there is a new relationship to be defined, a new separated function can also be created under the evaluateQuery function for that clause. This is useful when implementing relationships such as Affect in the next few relationships

3. Memory Utilization

As we are using vectors, memory utilization is kept to a minimum as compared to arrays. The amount of memory needed is proportional to the size of answers returned and the Query object.

4. Performance

The running time of BQE is O(nx) where n is the number of relationship clauses in the Query object and x is the running time taken by the PKB to find the solutions.

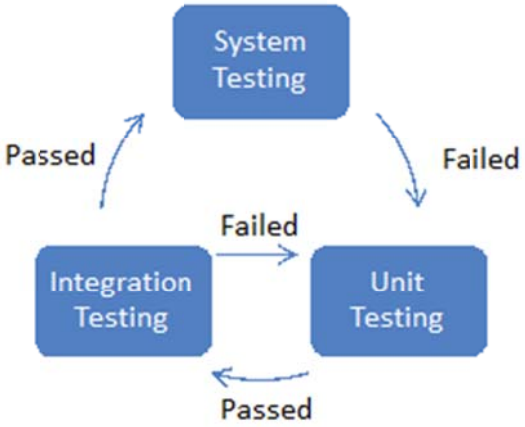
3. TESTING

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.

3.1. Testing Plan For Iteration 2

Iteration 2 adopts Test-Driven Development, with the diagram below illustrating the testing cycle of iteration 2. It started with the writing of system test cases at the beginning of iteration 2, which served as an executable specification of the system. The team then started to implement each functionality with the test cases in mind.



Each team member is responsible for his/her own unit testing. And once he/she has passed the unit testing, he/she can submit his/her codes for integration testing. Once integration testing is passed, the system will be tested again with system testing. All of these tests were carried out in an agile way, which makes for a flexible testing timeline and life cycle.

3.2. UNIT TESTING

Unit Testing was done on every sub-component of the SPA.

For the Front-End, some examples would be the TestNode.cpp, which is used to unit test our ASTNode object, and the TestParser.cpp, which is used to unit test all source code parsing methods. in the source code, whether they contains the expected values.

```
void ParserTest::testModifyTable(){
    PKB *pkb;
    pkb = PKB::getInstance();
    parserDriver("CodeParserTestIn.txt",pkb);

    VarTable* varTable = pkb->getVarTable();
    ProcTable* procTable = pkb->getProcTable();;
    Follows* follows = pkb->getFollows();
    Parent* parent = pkb->getParent();
    TypeTable* typeTable = pkb->getTypeTable();
    Modifies* modifies = pkb->getModifies();
    Uses* uses = pkb->getUses();
    Node* root = pkb->getASTRoot();

    string expected = "x";
    CPPUNIT_ASSERT_EQUAL(expected, varTable->getVarName(0));
    expected = "z";
    CPPUNIT_ASSERT_EQUAL(expected, varTable->getVarName(1));
    expected = "i";
    CPPUNIT_ASSERT_EQUAL(expected, varTable->getVarName(2));

    expected = "First";
    CPPUNIT_ASSERT_EQUAL(expected, procTable->getProcName(0));
    expected = "Second";
    CPPUNIT_ASSERT_EQUAL(expected, procTable->getProcName(1));

    CPPUNIT_ASSERT_EQUAL(false,parent->isParent(3,2));
    CPPUNIT_ASSERT_EQUAL(true,parent->isParent(6,7));
    CPPUNIT_ASSERT_EQUAL(true,parent->isParent(6,8));
    CPPUNIT_ASSERT_EQUAL(true,parent->isParent(10,12));
    CPPUNIT_ASSERT_EQUAL(false,parent->isParent(11,12));

    CPPUNIT_ASSERT_EQUAL(true,follows->isFollows(1,2));
    CPPUNIT_ASSERT_EQUAL(true,follows->isFollows(5,6));
    CPPUNIT_ASSERT_EQUAL(false,follows->isFollows(6,7));

    CPPUNIT_ASSERT_EQUAL(TypeTable::ASSIGN,typeTable->getType(1));
    CPPUNIT_ASSERT_EQUAL(TypeTable::ASSIGN,typeTable->getType(2));
    CPPUNIT_ASSERT_EQUAL(TypeTable::CALL,typeTable->getType(3));
    CPPUNIT_ASSERT_EQUAL(TypeTable::WHILE,typeTable->getType(6));

    CPPUNIT_ASSERT_EQUAL(false,modifies->isModifies(2,"x"));
    CPPUNIT_ASSERT_EQUAL(true,modifies->isModifies(13,"z"));
    CPPUNIT_ASSERT_EQUAL(true,modifies->isModifies(16,"z"));
    CPPUNIT_ASSERT_EQUAL(false,modifies->isModifies(21,"v"));

    CPPUNIT_ASSERT_EQUAL(true,uses->isUses(7,"x"));
    CPPUNIT_ASSERT_EQUAL(true,uses->isUses(7,"y"));
    CPPUNIT_ASSERT_EQUAL(false,uses->isUses(2,"z"));

    Node* curr = root;
    curr = root->getChild()[0];
    curr = curr->getChild()[0];
    curr = curr->getChild()[0];

    string expected = "=";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getData());
    expected = "assign";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getType());

    curr = curr->getChild()[0];

    expected = "x";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getData());
    expected = "variable";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getType());
}
```

For the Query Processor, we have the QueryEvaluatorTest.cpp, which is used to unit test all evaluation after Query Pre-Processing, and the QueryParserTest.cpp, which is used to unit test methods involved in parsing the queries into QueryTree objects.

```
#ifndef TestQueryEvaluator_h
#define TestQueryEvaluator_h

#include <cppunit/extensions/HelperMacros.h>

class QueryEvaluatorTest: public CPPUNIT_NS::TestFixture
{
    CPPUNIT_TEST_SUITE( QueryEvaluatorTest );
    CPPUNIT_TEST( testEvaluateFollows );
    CPPUNIT_TEST( testEvaluateParent );
    CPPUNIT_TEST( testEvaluateModifies );
    CPPUNIT_TEST( testEvaluateUses );
    CPPUNIT_TEST_SUITE_END();

public:
    void setUp();
    void tearDown();

    void testEvaluateFollows();
    void testEvaluateParent();
    void testEvaluateModifies();
    void testEvaluateUses();

};

#endif
```

For the PKB, every single implemented relationship (Parent, Follows, Uses, and Modifies) has a UnitTest specific to the relationship.

3.3. Integration Testing

Integration Testing was split into two parts, Parser-PKB and PKB-Query Processor.

```
#ifndef TestIntegrate_h
#define TestIntegrate_h

// Note 1
#include <cppunit/extensions/HelperMacros.h>

class IntegrateTest : public CPPUNIT_NS::TestFixture // Note 2
{
    CPPUNIT_TEST_SUITE( IntegrateTest ); // Note 3
    CPPUNIT_TEST( testParserSource2 );
    CPPUNIT_TEST( testPQLSource1 );
    CPPUNIT_TEST_SUITE_END();

public:
    void setUp();
    void tearDown();

    // method to test the assigning and retrieval of grades
    void testParserSource2();
    void testPQLSource1();
};
#endif
```

For Parser-PKB testing, a sample source is parsed and assertions are made to see the correctness of said parsing.

```
void IntegrateTest::testParserSource2()
{
    PKB* pkb;
    pkb = PKB::getInstance();
    parserDriver("Source2.txt",pkb);

    VarTable* varTable = pkb->getVarTable();
    ProcTable* procTable = pkb->getProcTable();;
    Follows* follows = pkb->getFollows();
    Parent* parent = pkb->getParent();
    TypeTable* typeTable = pkb->getTypeTable();
    Node* root = pkb->getASTRoot();

    Node* curr = root;

    string expected = "root";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getData());
    expected = "program";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getType());

    curr = curr->getChild()[0];
    expected = "ABC";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getData());
    expected = "procedure";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getType());

    curr = curr->getChild()[0];
    expected = "ABC";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getData());
    expected = "stmtLst";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getType());

    curr = curr->getChild()[3];
    expected = "4";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getData());
    CPPUNIT_ASSERT_EQUAL(4, curr->getProgLine());
    expected = "while";
    CPPUNIT_ASSERT_EQUAL(expected, curr->getType());

    return;
}
```

For PKB-Query Processor testing, queries are parsed by the QueryParser and then evaluated in the QueryProcessor. The answers provided by the QueryProcessor are asserted to check for correctness.

```
// TODO: run query parser;
QueryParser qp;
PKB* pkb = PKB::getInstance();
QueryEvaluator qe(pkb);
Follows* f = qe.pkb->getFollows();
TypeTable* t = qe.pkb->getTypeTable();

//Query 1 assign a; Select a such that Follows(1, 2)
qp.validate(s1);
Query q1 = qp.parse();
v = q1.getRelVect();
expected = "a";
Relationship r = v[0];
unordered_map<string, TypeTable::SynType> m = q1.getSynTable();

CPPUNIT_ASSERT_EQUAL(Relationship::FOLLOWS, r.getRelType());
CPPUNIT_ASSERT_EQUAL(expected, q1.getSelectedSyn());
expected = "1";
CPPUNIT_ASSERT_EQUAL(expected, r.getToken1());
expected = "2";
CPPUNIT_ASSERT_EQUAL(expected, r.getToken2());

/*f->setFollows(1,2);
t->insertStmtNumAndType(1, TypeTable::ASSIGN);
t->insertStmtNumAndType(2, TypeTable::ASSIGN);*/
CPPUNIT_ASSERT(qe.evaluateQueryBoolean(q1)==true);
```

3.4. System Testing

System testing is carried out at the beginning and end of each testing life cycle, as already explained in chapter 3.1.

Iteration	Level	Availability	Function	Source	Test	Status
2	Basic	Available	Next	Source 2A	QBasic2A-next	Pending
2	Basic	Available	Next*	Source 2B	QBasic2B-next-star	Pending
2	Basic	Available	Invalid queries	<any source>	QBasic2C-invalid-queries	Pending
2	Intermediate	Available	Multiple Clause	Source 2C	QInterm2A-such-that-such-that	Pending
2	Intermediate	Available	Multiple Clause	Source 2C	QInterm2B-such-that-with	Pending
2	Intermediate	Available	Multiple Clause	Source 2C	QInterm2C-with-pattern	Pending
2	Advanced	Available	Multiple Clause	Source 2D	QAdv2A-such-that-with-pattern	Pending

To ensure that the test cases are collectively exhaustive, scenarios that are covered by the test cases are also documented systematically. The table on the next few pages will detail the scenarios used for iteration 2.

Iteration	Level	Function	Availability	Requirement 1	Requirement 2	Status
2	Basic	Next	Avalable	#, n2	select n2	Pending
			Avalable	#, n2	select boolean	Pending
			Avalable	n1, #	select n1	Pending
			Avalable	n1, #	select boolean	Pending
			Avalable	#, _	select n2	Pending
			Avalable	#, _	select boolean	Pending
			Avalable	_, #	select n1	Pending
			Avalable	_, #	select boolean	Pending
			Avalable	_, _	select n	Pending
			Avalable	_, _	select boolean	Pending
			Avalable	n1, n2	select n1	Pending
			Avalable	n1, n2	select n2	Pending
			Avalable	#, #	select boolean	Pending
2	Basic	Next*	Available	#, n2	select n2	Pending
			Available	#, n2	select boolean	Pending
			Available	n1, #	select n1	Pending
			Available	n1, #	select boolean	Pending
			Available	#, _	select n2	Pending
			Available	#, _	select boolean	Pending
			Available	_, #	select n1	Pending
			Available	_, #	select boolean	Pending
			Available	_, _	select n	Pending
			Available	_, _	select boolean	Pending
			Available	n1, n2	select n1	Pending
			Available	n1, n2	select n2	Pending
			Available	#, #	select boolean	Pending
2	Basic	Invalid Queries	Available	Next	type mismatch	Pending
			Available	Next*	type mismatch	Pending
2	Intermedi	Multiple	Available	<Matrix		Pending

	ate	Clause		A>		
2	Advanced	Multiple Clause	Available	Modifies	with, pattern	Pending
			Available	Modifies*	with, pattern	Pending
			Available	Uses	with, pattern	Pending
			Available	Uses*	with, pattern	Pending
			Available	Calls	with, pattern	Pending
			Available	Calls*	with, pattern	Pending
			Available	Parent	with, pattern	Pending
			Available	Parent*	with, pattern	Pending
			Available	Follows	with, pattern	Pending
			Available	Follows*	with, pattern	Pending
			Available	Next	with, pattern	Pending
			Available	Next*	with, pattern	Pending

**) Matrix A

Clause 1/2	Modifies	Modifies*	Uses	Uses*	Calls	Calls*	Parent	Parent*	Follows	Follows*	Next	Next*	With	Pattern
Modifies	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Modifies*		15	16	17	18	19	20	21	22	23	24	25	26	27
Uses			28	29	30	31	32	33	34	35	36	37	38	39
Uses*				40	41	42	43	44	45	46	47	48	49	50
Calls					51	52	53	54	55	56	57	58	59	60
Calls*						61	62	63	64	65	66	67	68	69
Parent							70	71	72	73	74	75	76	77
Parent*								78	79	80	81	82	83	84
Follows									85	86	87	88	89	90
Follows*										91	92	93	94	95
Next											96	97	98	99
Next*												100	101	102
With													103	104
Pattern														105

4. Coding Standards

Our team members adopted similar coding standards which are adjusted appropriately and respectively according to the design specifications of various components. Some of the coding standards that the components possess are listed below:

1. Indentation and whitespace
    a. a. Indication of code segments
2. Comments to enhance understanding and communication
3. Descriptive variable declarations
    a. Always start with lower case
    b. Use CamelCase
    c. Use only letters and numbers
4. Informative function naming conventions
    a. All getters start with “get”
    b. All setters start with “set”
    c. All functions that start with “is” returns a Boolean value
5. Keep it simple and effective
    a. Avoid complex code fragments
6. Refactoring

Standards between abstract APIs and concrete APIs

The correspondence between the abstract and concrete APIs was enhanced by doing the following:

1. The abstract APIs provides the interface for the concrete APIs
2. Making abstract APIs as comprehensive as possible a. Offering an Extensive description of the abstract APIs b. Specifying the complete parameters needed for the function

5. API

Please view our Doxygen at:

www.comp.nus.edu.sg/~kester/CS3202