## VarTable API

|  |
| --- |
| VarTable |
| Overview: VarTable is to keep all the variables appearing in the program |
| Public Interface: |
| INDEX *insertVar* (STRING VarName)  Description:  If “varName” is not in the VarTable, insert it into the VarTable and return its index value. Otherwise, return -1 (special value) and the table remains unchanged. |
| STRING *getVarName* (INDEX ind)  Description:  If there is record in VarTable having index value “ind”, return its variable name.  If “ind” is out of range:  Throws: InvalidReferenceException |
| INDEX *getVarIndex* (STRING varName)  Description:  If there is record in VarTable having name “varName”, return its index value.  Otherwise, return -1 (special value) |

## FollowTable API

|  |
| --- |
| Follow |
| Overview: Follow is to keep the relationship Follows of any pair of statements appearing in the program into a table. |
| Public Interface: |
| BOOLEAN *isFollows* (STMT\_NUM s1, STMT\_NUM s2)  Description:  If the relation Follows(s1, s2) is recorded in Follow Table, return true. Otherwise return false. |
| INDEX *insertFollows* (STMT\_NUM s1, STMT\_NUM s2)  Description:  If the relation Follows(s1, s2) is not in Follow Table, insert it into the table and return its index value.  Otherwise: return -1 (special value) and the table remains unchanged. |
| LIST<STMT\_NUM> *getFollowingStmt* (STMT\_NUM s1)  Description:  If s1 > 0, return an array of all statement numbers recorded in table that follow statement “s1” (Follows(s1, s)).  Otherwise, return NULL |
| LIST<STMT\_NUM> *getFollowedStmt* (STMT\_NUM s1)  Description:  If s1 > 0, return an array of all statement numbers recorded in table that are followed by statement “s1” (Follows(s, s1)).  Otherwise, return NULL |
| LIST<STMT\_NUM> *getFollowedStarStmt* (STMT\_NUM s1)  Description:  If s1 > 0, return an array of all statement numbers “s” recorded in table that Follows\*(s, s1) exists.  Otherwise, return NULL |

## ParentTable API

|  |
| --- |
| Parent |
| Overview: ParentTable is to keep the relationship Parent of any pair of statements appearing in the program into a table. |
| Public Interface: |
| BOOLEAN *isParent* (STMT\_NUM s1, STMT\_NUM s2)  Description:  If the relation Parent(s1, s2) is recorded in Parent Table, return true.  Otherwise return false. |
| INDEX *insertParent* (STMT\_NUM s1, STMT\_NUM s2)  Description:  If the relation Parent(s1, s2) is not in Parent Table, insert it into the table and return its index value.  Otherwise: return -1 (special value) and the table remains unchanged. |
| STMT\_NUM *getParentStmt* (STMT\_NUM s1)  Description:  If s1 > 0, return statement number “s” recorded in table that is direct parent of statement “s1” (Parent(s, s1)).  Otherwise, return NULL |
| LIST<STMT\_NUM> *getChildStmt* (STMT\_NUM s1)  Description:  If s1 > 0, return all statement numbers recorded in table that are direct children of statement “s1” (Parent(s1, s)).  Otherwise, return NULL |
| LIST<STMT\_NUM> *getParentStarStmt* (STMT\_NUM s1)  Description:  If s1 > 0, return all statement numbers recorded in table that are parent (direct or indirect) of statement “s1” (Parent\*(s, s1))  Otherwise, return NULL |
| LIST<STMT\_NUM> *getChildStarStmt* (STMT\_NUM s1)  Description:  If s1 > 0, return all statement numbers recorded in table that are direct or indirect children of statement “s1” (Parent\*(s1, s)).  Otherwise, return NULL |

## Modify API

|  |
| --- |
| Modify |
| Overview:   1. Modify for assignment statements is to keep the relationship Modifies(a, x) of statement a and variable x appearing in the program into a table. The table keeps Modifies(a, x) by recording the statement number “a” and index value of variable “x” in the VarTable. 2. Modify for statements is to keep the relationship Modifies(“if”, x) or Modifies(“while”, x) of containers “if” or “while” and variable x appearing in the program into a table. The modifies table keeps Modifies relationship by recording the container statements number a and index value of variable “x” in the VarTable. We can check if a container includes a statement by checking the Parent\* relationship of that statement number. This table for Modifies is the same table used in point a). 3. Modify for procedures just checks if the the statement is contained in the procedure by checking against the AST and then using the Modify table.This table for Modifies is the same table used in point a). |
| Public Interface: |
| BOOLEAN *isModifies* (STMT\_NUM s, INDEX varIndexOfx)  Description:  If there is no record of relation Modifies() of statement “s” and variable “x”, return FALSE.  Otherwise return TRUE. |
| INDEX *insertModifies* (STMT\_NUM s, INDEX varIndexOfx)  Description:  If the relation Modifes(s, “x”), is not in Modify Table, insert it into the table and return its index value.  Otherwise: return -1 (special value) and the table remains unchanged. |
| LIST<INDEX> *getModifiedVarAtStmt* (STMT\_NUM s)  Description:  If s > 0  just return an array of all index values recorded in table whose variable are modified by  statement “s”.  Otherwise, return NULL. |
| LIST<STMT\_NUM> *getStmtModifyingVar* (INDEX varIndexOfx)  Description:  If variable name “x” is recorded in VarTable, return an array of all statement numbers recorded in table that modify variable having index value “ind” in VarTable.  Otherwise, return NULL. |

## Uses API

|  |
| --- |
| Uses |
| Overview:  UsesTable is to keep the relationship Uses() of any pair of statements appearing in the program into a table. |
| Public Interface: |
| BOOLEAN *isUses* (int s, INDEX varIndexOfx)  Description:  If there is no record of relation Uses() of statement “s” and index of variable “x”, return FALSE.  Otherwise return TRUE. |
| NDEX *insertUses* (STMT\_NUM s, INDEX varIndexOfx)  Description:  If the relation Uses(s, “x”), is not in Uses Table, insert it into the table and return its index value.  Otherwise: return -1 (special value) and the table remains unchanged. |
| LIST<INDEX> *getUsedVarAtStmt* (STMT\_NUM s)  Description:  If s > 0  just return an array of all index values recorded in table whose variable are used by  statement “s”.  Otherwise, return NULL. |
| LIST<STMT\_NUM> *getStmtUsingVar* ( INDEX varIndexOfx)  Description:  If variable name “x” is recorded in VarTable, return an array of all statement numbers recorded in table that use variable having index value varIndexOfx” in VarTable.  Otherwise, return NULL. |

## Statement Table API

|  |
| --- |
| StatTable |
| Overview: StatTable is to keep all the variables appearing in the program |
| Public Interface: |
| INDEX *insertStmt* (STRING name)  Description:  If “varName” is not in the VarTable, insert it into the VarTable and return its index value. Otherwise, return -1 (special value) and the table remains unchanged. |
| STRING *getStmtName* (INDEX ind)  Description:  If there is record in StatTable having index value “ind”, return its statement name.  If “ind” is out of range:  Returns “variable not found” message. |
| LIST<INDEX> *getStmtIndex* (string stmtName)  Description:  If there is record in StatTable having name “stmtName”, return its index value.  Otherwise, return -1 (special value) |

## PKB API

|  |
| --- |
| PKB |
| Overview: The PKB contains all the components required for the storage of data. Such as the tables and AST. |
| Public Interface: |
| static BOOLEAN isFollows(STMT\_NUM s1, STMT\_NUM s2)  Description:  Method to return if statement s1 is followed by statement s2. Return true if relationship holds, otherwise return false. |
| static BOOLEAN isFollowsStar(STMT\_NUM s1, STMT\_NUM s2)  Description:  Method to check Follows\*(s1, s2) holds. Return true if relationship holds, otherwise return false. |
| static BOOLEAN insertFollows(STMT\_NUM s1, STMT\_NUM s2)  Description:  Method to insert a pair of following statement numbers in FollowTable. Return true if successful, otherwise return false. |
| static STMT\_NUM getFollowingStmt(STMT\_NUM s1)  Description:  Method to get the following statement to statement number s1.  Return the statement number if found, otherwise return -1. |
| static STMT\_NUM getFollowedStmt(STMT\_NUM s1)  Description:  Method to get statement which is followed by statement s1. Return the statement number if found, otherwise return -1. |
| LIST<STMT\_NUM> getFollowingStarStmt(STMT\_NUM s1)  Description:  Method to get the list of  following statements to statement number s1 with relationship Follows\*.  Otherwise, return NULL. |
| LIST<STMT\_NUM> getFollowedStarStmt(STMT\_NUM s1)  Description:  Method to get the list of statements which are followed star by statement s1.  Otherwise, return NULL. |
| LIST<STMT\_NUM> getAllFollowingStmt()  Description:  Method to get statement which is followed star by statement s1.  Otherwise, return NULL. |
| LIST<STMT\_NUM> getAllFollowedStmt()  Description:  Method to get statement which is followed star by statement s1.  Otherwise, return NULL. |
| INTEGER getFollowTableSize()  Description:  Returns the number of records in FollowTable. |
| BOOLEAN isModifies(STMT\_NUM s1, INDEX varIndex)  Description:  Method to check if modifies relationship exists. Return true if exists, otherwise return false. |
| BOOLEAN insertModifies(STMT\_NUM s1, INDEX varIndex)  Description:  Method to insert a pair of statement number and variable holding the Modify relationship in ModifyTable.  Return true if successful, otherwise return false. |
| LIST<INDEX> getModifiedVarAtStmt(STMT\_NUM s1)  Description:  Method to get the variables modified in statement s1. Otherwise, return NULL. |
| LIST<STMT\_NUM> getStmtModifyingVar(INDEX varIndex)  Description:  Method to get the list of statements that modify var. Otherwise, return NULL. |
| LIST<STMT\_NUM> getAllModifyingStmt()  Description:  Method to get all statements modifying some variable. Otherwise, return NULL. |
| LIST<INDEX> getAllModifiedVar()  Description:  Method to get all variables being modified. Otherwise, return NULL. |
| INTEGER getModifyTableSize()  Description:  Returns the number of records in ModifyTable. |
| INDEX insertVar(STRING name)  Description:  Inserts variable in VarTable and returns its index.  Otherwise, return -1. |
| INDEX getVarIndex(STRING name)  Description:  Returns the index of the variable.  Otherwise, return -1. |
| STRING getVarName(INDEX index)  Description:  Returns the variable name given its index.  Otherwise, return NULL. |
| INTEGER getVarTableSize()  Description:  Returns the number of records in VarTable. |
| BOOLEAN isParent(STMT\_NUM s1, STMT\_NUM s2)  Description:  Method to return if statement s1 is parent of statement s2. Return true if relationship holds, otherwise return false. |
| BOOLEAN isParentStar(STMT\_NUM s1, STMT\_NUM s2)  Description:  Method to return if statement s1 is parent star of statement s2. Return true if relationship holds, otherwise return false. |
| BOOLEAN insertParent(STMT\_NUM s1, STMT\_NUM s2)  Description:  Method to insert a pair of parent and child statement numbers in ParentTable.  Return true if successful, otherwise return false. |
| STMT\_NUM getParentStmt(STMT\_NUM childStmt)  Description:  Return the statement number that is parent of child statement if found in ParentTable. Otherwise return -1. |
| LIST<STMT\_NUM> getChildStmt(STMT\_NUM parentStmt)  Description:  Return an array of child statement numbers of parent statement if found in ParentTable. Otherwise return NULL. |
| LIST<STMT\_NUM> getParentStarStmt(STMT\_NUM childStmt)  Description:  Return an array of statement numbers that are parent star of child statement if found in ParentTable. Otherwise return NULL. |
| LIST<STMT\_NUM> getChildStarStmt(STMT\_NUM parentStmt)  Description:  Return an array of child statement numbers of parent star statement if found in ParentTable. Otherwise return NULL. |
| LIST<STMT\_NUM> getAllParentStmt()  Description:  Return an array of all statements that are parent of some child statement found in ParentTable. Otherwise return NULL. |
| LIST<STMT\_NUM> getAllChildStmt()  Description:  Return an array of all statements that are child of some parent statement found in ParentTable. Otherwise return NULL. |
| INTEGER getParentTableSize()  Description:  Returns the number of records in ParentTable. |
| BOOLEAN insertStmt(STRING name)  Description:  Inserts statement in StatTable. Return true if successful, otherwise return false. |
| LIST<STMT\_NUM> getStmtIndex(STRING name)  Description:  Return index of statement having name in StatTable. Otherwise, return NULL. |
| STRING getStmtName(INDEX index);  Description:  Return name of statement having index in StatTable. Otherwise, return NULL. |
| INTEGER getStatTableSize()  Description:  Returns the number of records in StatTable. |
| BOOLEAN isUses(STMT\_NUM s1, INDEX varIndex)  Description:  Method to check if uses relationship exists.  Return true if exist, otherwise return false. |
| BOOLEAN insertUses(STMT\_NUM s1, INDEX varIndex)  Description:  Method to insert a pair of following statement number and variable.  Return true if successful, otherwise return false. |
| LIST<INDEX> getUsedVarAtStmt(STMT\_NUM s1)  Description:  Method to get the variables used in statement s1.  Otherwise, return NULL. |
| LIST<STMT\_NUM> getStmtUsingVar(INDEX varIndex)  Description:  Method to get the list of statements using variable.  Otherwise, return NULL. |
| LIST<STMT\_NUM> getAllUsingStmt()  Description:  Method to get the list of all statements using some variable.  Otherwise, return NULL. |
| LIST<INDEX> getAllUsedVar()  Description:  Method to get all variables being used in some statement.  Otherwise, return NULL. |
| INTEGER getSize()  Description:  Method to get the number of records in UsesTable. |

## AST API

|  |
| --- |
| AST |
| Overview: The AST stores the source file in a tree representation. |
| Public Interface: |
| TNode findNodeOfStmt(INDEX index)  Description:  Returns the node with statment number "index" in AST. Otherwise, return null. |
| BOOLEAN hasSubTree(Tree tree)  Description:  Checks if AST has the subtree “tree”. Returns true if found, otherwise return false. |

## Tree API

|  |
| --- |
| Tree |
| Overview: The Tree is used as the underlying structure for the implementation of the AST and Query Tree |
| Public Interface: |
| TNode createNode(STRING type, STRING value)  Description:  Creates a Tnode with type and value and returns the node. |
| TNode getRoot()  Description:  Returns the root node of the tree. |
| INTEGER getNumChildren(TNode node)  Description:  Returns the number of child nodes of the node |
| TNode getChildAtIndex(TNode node, INDEX index)  Description:  Returns the child node of node at the “index” position. |
| void setRoot(TNode node)  Description:  Sets the node as the root of the tree. |
| INDEX setChild(TNode parent, TNode child)  Description:  Sets the node as the child node of parent node and returns the index position of the child node. |

## Query Representator API

|  |
| --- |
| Query Representator |
| Overview: This acts as the data storage for the evaluation of the queries. |
| Public Interface: |
| SymbolTable getSymbolTable(INDEX index)  Description:  Returns SymbolTable of query at index "index". |
| QueryTree getQueryTree(INDEX index)  Description:  Return QueryTree of query at index "index". |
| BOOLEAN getQueryValidity(INDEX index)  Description:  Checks query's validity at index "index". Return true if query is valid, otherwise return false. |
| INTEGER getSize()  Description:  Return the number of queries. |
| INTEGER addQuery(SymbolTable table, QueryTree tree, BOOLEAN isQueryValid)  Description:  Adds new query into QueryRepresentator and returns the updated number of queries. |
| void reset()  Description:  Resets all data in QueryRepresentator. It only needs to be called by QueryPreprocessor. |

## Query Evaluator API

|  |
| --- |
| Query Evaluator |
| Overview: The Query Evaluator works through the individual queries to evaluate them. |
| Public Interface: |
| void Evaluate()  Description:  Starts the query evaluating process. |
| LIST<STRING> getResult(INDEX index)  Description:  Returns result of query at index "index”. |
| LIST<LIST<STRING>> getAllResult()  Description:  Returns results of all queries. |

## Query Preprocessor API

|  |
| --- |
| Query Preprocessor |
| Overview: The Query Preprocessor has the responsibility of validating the queries. |
| Public Interface: |
| void Preprocess()  Description:  Starts preprocessing the file whose directory is saved in fileDirectory. |
| void Preprocess(STRING query)  Description:  Preprocesses query. |

# UML Diagrams

The UML sequence diagrams presented in this section display how the SPA program flow works between the Parser, PKB and QP. A detailed explanation of the interactions between these components will be provided in section 7.

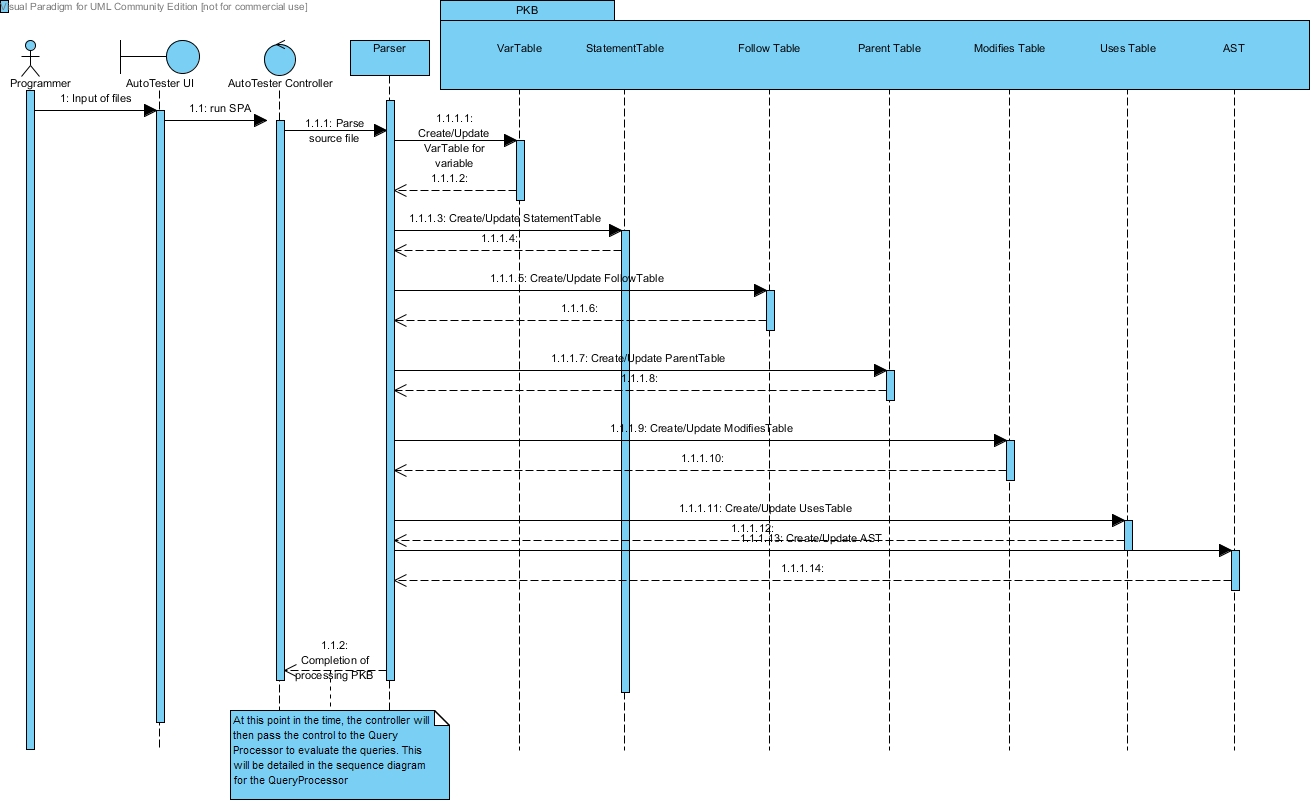
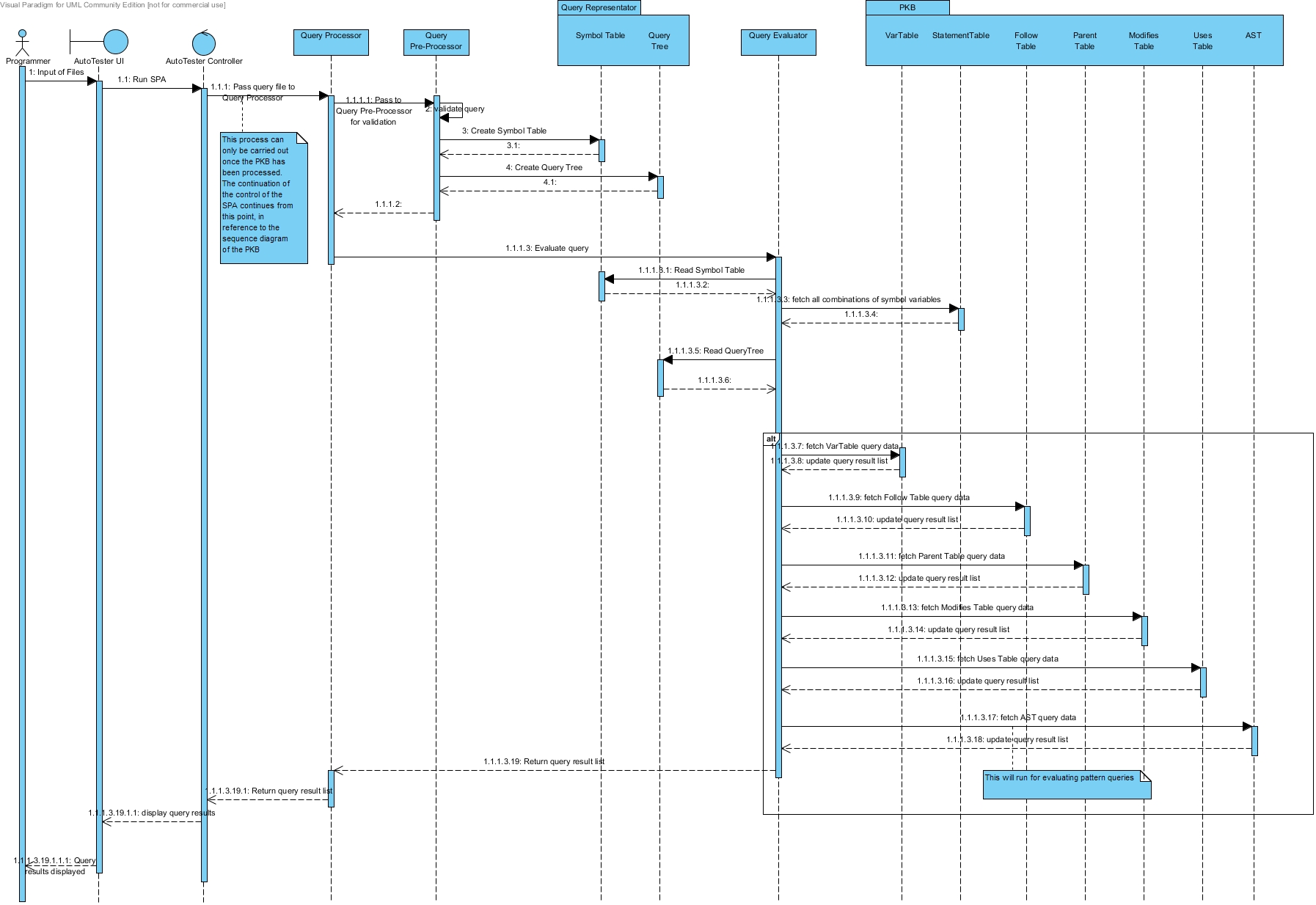


Figure 1: Sequence Diagram for Processing PKB

Figure 2: Sequence Diagram for Query Processor Flow

# Comments on Design Decisions

The first alteration that we made in terms of design was in terms of evaluating the queries. The handbook suggests that we traverse the AST to evaluate the queries. However our group looked into it and saw that it would be better to use the information from the PBK, based on the tables, to evaluate the ‘such that’ queries and to use a combination of the PKB and the AST to respond to ‘pattern’ queries.

The implementation of the Query Processor, QP, also varies from that suggested in the handbook. In our implementation we have a total of 3 sub-components in the QP: Query Preprocessor, QPP, Query Evaluator, QE and a Query Representator, QR. The main aim of the QR is to act as a sort of a database for the QP. The QR holds the query tree and the symbol table. Much like the PKB which holds various tables and the AST. The reason for implementing the QR was to have a more object oriented design and also facilitate in information hiding. These design principles are vital for a good software. A visual representation of our implemented QP is shown in Fig. 3 below.

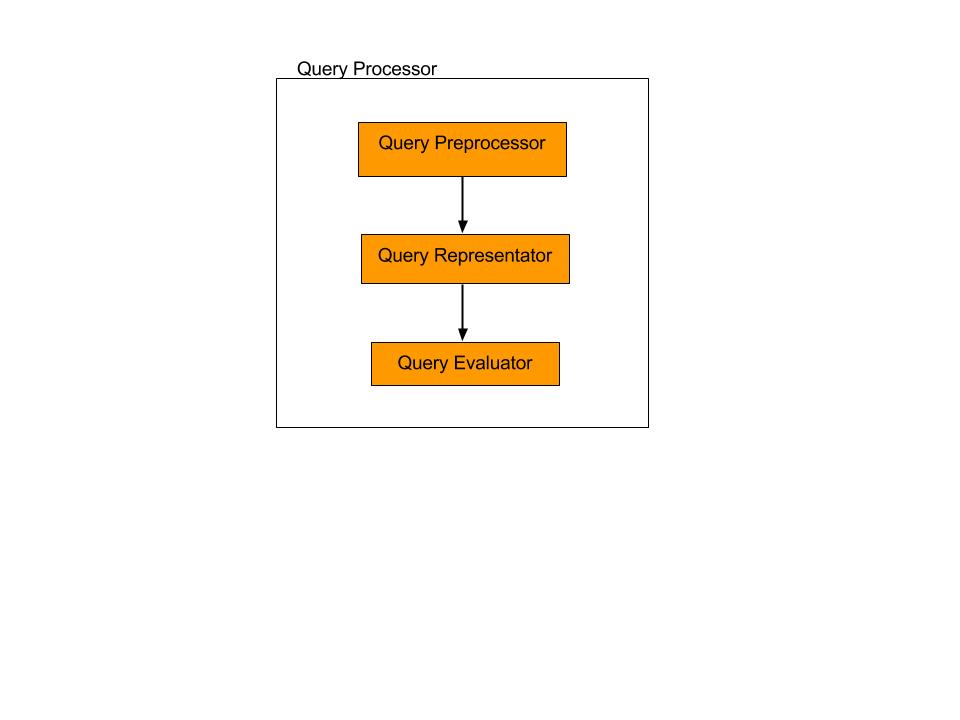


Figure 3: Query Processor Representation

In terms of the implementation of the tables in the PBK, we choose represent them as vectors. This decision was made since the size of the table is unknown so vectors would be easier to keep track of and manage.

Finally we also excluded the implementation of the Design Extractor from this prototype. This decision was made since this prototype did not implement the Next and Affects design abstractions.

# Coding Standards & Experiences

In terms of the coding standards, our group had decided to adopt the following naming conventions described in section 6.1.

## 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 sould 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().

1. Methods with return types of other values should have the form getXXX()

E.g. getVarName()

1. Methods that return the number of records inside a table/ list should have the form getSize().
2. Methods that change the values or status of an object should have the form: setXXX()
3. 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 that the method will adopt.

In terms of enhancing the correspondence, we have ensured that we use the same names to document the classes and the respective methods.

# Component Design & Interaction

## Overall Interaction & Design

For this prototype we have used the AutoTester as the main UI and Controller. The AutoTester accepts the user inputs as defined by the assignment requirements. In terms of the implementation, the user will input 3 files, those being the source file, query file and the output file. Once the AutoTester has been provided with the abovementioned files, it will pass the control to the Parser to go through the source files and save the relevant data into the PKB. This process is shown the sequence diagram Fig. 1 of section 4. The PKB consist of the following data representations: Var Table, Statement Table, Follow Table, Parent Table, Modifies Table, Uses Table and the AST. All of the tables have been implemented using C++ vectors and the AST has been implemented by designing a Tree class. This Tree class then in turn has been implemented by TNode class as suggested by the handbook. We have implemented the SPA in such a manner that the queries will only be evaluated when the Parser has completed processing information into the PKB.

Once the data structures inside the PKB have been filled with the relevant information, control of the SPA will return back to the AutoTester. The AutoTester will now pass the query file to the QP to validate and evaluate the queries. How both of these are done is discussed in the following sub-sections.

## Query Processor Design & Interaction

As mentioned in section 5, the QP has been designed to consist of 3 other components, namely the Query Preprocessor, QPP, the Query Representator, QR, and the Query Evaluator, QE. This implementation differs slightly from that suggested in the project handbook, as we have included an additional component, that being the QR. The rational for doing so was explained in section 5 as well.

When a query is input into the AutoTester, the AutoTester will pass this query to the QP. The QP will then call the QPP. The main job of the QPP is to validate the program queries and save data that will be required to answer the query. This is done by saving the relevant parts of the query in the QR, which the QPP will then call. The QR consists of the Symbol Table and also the QueryTree. The QPP first looks at the declaration part of the query. It then saves the various synonym names into the Symbol Table. It will then move on to extract information from the select statement to be saved in the Query Tree. For the implementation of the Query Tree we have followed the implementation suggested by the handbook. A representation of this whole process was presented in the sequence diagram of Fig 2, in section 4.

## Query Preprocessor & Validating Program Queries

As mentioned in section 7.2, the validation of program queries is carried out by the QPP. The first thing that the QPP does is to validate the source query file, when it receives it from the AutoTester. Next it validates each query one at a time.

### Source Query File Validation

(insert image for this part)

Check format line by line:

* Check if 1st line provides the number of queries.
* Do nothing for the 2nd line.
* For every 5 lines after, do the following:
  + 1st Line: Do nothing.
  + 2nd & 3rd Lines: Save data into temporary variables.
  + 4th & 5th Lines: Do nothing.

### Individual Query Validation

Check each query line by line and get the relevant works in the line, following these rules:

* On finding the declaration, send this part to the preprocessDeclaration() method. This will then verify the syntax of the declaration part of the query.
* On finding the query, send this part to the preprocessQueryPart() method which will then verify the syntax of the individual query itself. This includes the select, such that and pattern type queries.

Now as each query is successfully validated, the information extracted from the individual queries is then saved into the Query Representator, from the temporary variables, as mentioned in the previous section.

## Query Representator Design & Interaction

Once the queries have been validated, control of the program will then be passed to the QR. The variables extracted from the declaration part of the individual query will be saved in the Symbol Table. This table will record the type of statement, assignment or while, and the variable alongside it. Following this, the second part of the query, being the select, such that and pattern statement will be stored in the Query Tree. The design of the Query Tree is similar to the AST, with respect to the type of object that it is. Just like the AST, the Query Tree is also an object of type Tree, a class that we created. An instance of a Query Tree is shown below in Fig. 4 for the example query 1.

Example Query 1:

stmt s; assign a;

Select a such that Follows(a,s)

Figure 4: Query Tree for Example Query 1

## Query Evaluator Design & Interaction

Once The QR has completed processing and storing all of the necessary information it sends a signal to the QP, which then calls the QE which can then start the process of evaluating the queries. First of all the QE will access the QR to read from the Symbol Table to get the type of variable required for the evaluation of the query. It then accesses the PKB, to get a list of all the possible combinations of that type of variable from the Statement Table. Next it will read the Query Tree node by node.

When it encounters a such that node, it will loop through all the possible combinations for that particular relationship, having retrieved this from the relation node, based on the 2 children nodes of the such that node and the list retrieved from the Statement Table. It will then access the relevant table of the relationship, in the PKB, to check if such a relationship does in fact exist.

Similarly, when the QE encounters a pattern node, it will compare the pattern, which will be represented by the descendent nodes of the pattern node, with that in the AST.

In both cases if a result evaluates to be true, it will be saved in the result list. If it is false then nothing will be input into the result list.

In the end when the whole query has been evaluated, this result list will be returned to the QP, which will in turn return it back to the AutoTester to be displayed back to the user.

# Testing

We have broken down the testing into 3 components. Unit, integration and system testing. The approach that we used was as follows. Whoever implements a component of the SPA will implement unit testing for it. Moving on to the integration testing aspect, the individuals responsible for implementing communicating components will work on the integration testing for those specific components. Finally integration testing would be carried out by all group members. The idea of testing also includes writing test cases by the respective people carrying out those tests.

As a whole we have come to realize that testing is a vital aspect in every software development project. It is only through testing at all levels, did we find out bugs from our components, that otherwise would have remained hidden.

## Unit Testing

### Test Case 1

Test Purpose:

Required Test Input:

Expected Test Results:

### Test Case 2

Test Purpose:

Required Test Input:

Expected Test Results:

## Integration Testing

### Test Case 1

Test Purpose:

Required Test Input:

Expected Test Results:

### Test Case 2

Test Purpose:

Required Test Input:

Expected Test Results:

## System Testing

### Test Case 1

Test Purpose:

Required Test Input:

Expected Test Results:

### Test Case 2

Test Purpose:

Required Test Input:

Expected Test Results:

# Discussion

Looking at the experience of working on this project as a whole, we have gained more insight as to what goes on behind the process of coming up with a software. Though this was just a prototype, the management skills and technical knowledge required is paramount to the success of any project.

The most challenging aspect of this project was the fact that we had to code the software in C++. A language none of us had really worked with before at this level. As a result, we had to allocate extra time to understanding how things in C++ work, in terms of its syntax. That being said, since all of us are familiar with Java, an object oriented language like C++, we had no problems with understanding the concepts.

Overall, our group’s communication was quite effective. In the beginning, we had some trouble in understanding each other due to a language barrier. This was resolved as we interacted more with each other. Everyone contributed equally towards the completion of this project and this made working together very pleasant.

One thing that we would do differently is organize ourselves better from the beginning i.e. when the assignment first came out. This would also allow us to pace ourselves better.

# Comments on the 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.