School of Computing

CS3203 Software Engineering Project

AY22/23 Semester 1

Project Report – System Overview

Team 28

Team Members	Student No.	Email
Lua Yi Da	A0217373E	e0543409@u.nus.edu
Benedict Chua Jun Jie	A0217474B	benedictchuajj@u.nus.edu
Chan Shi Yuan Galvin	A0217812J	e0543848@u.nus.edu
Chen Xiaotong	A0211185N	e0492455@u.nus.edu
Yang Xiquan	A0211233B	yangx@u.nus.edu
Cao Ngoc Linh	A0174119E	e0205114@u.nus.edu

Consultation Hours: Tuesday 3pm-4pm
Tutor: Bennett Clement

1 System Architecture

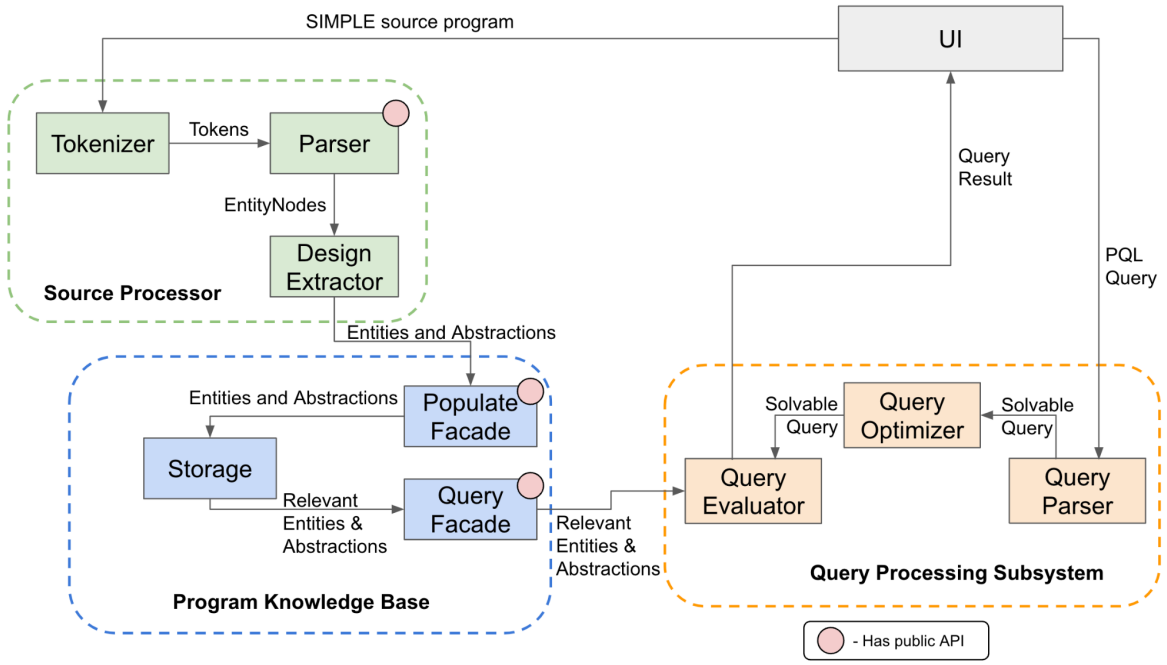

Figure 1 Architecture Diagram of SPA

SPA can be split up into 4 main components: UI, Source Processor (SP), Program Knowledge Base (PKB) and Query Processing Subsystem (QPS). As of Milestone 1, the focus will be on Source Processor, Program Knowledge Base and Query Processing Subsystem.

SP is responsible for parsing a source program (written in SIMP`LE) and extracting information from the program. This information will then be used to answer queries regarding the source program. This is achieved through a series of steps: Firstly, the Tokenizer break up the program into tokens, which represent the smallest semantic unit for processing. The tokens are then passed to the Parser, which constructs an Abstract Syntax Tree (AST), a representation that encapsulates the relationships between tokens. Finally, the Design Extractor then extracts out relevant information from the EntityNodes (our implementation of AST) such as the Entities involved in the source program as well as the relationships that can be found (e.g. Follows, Uses, etc) and stores those into PKB.

PKB serves as the database for SPA, and acts as the single source of truth for all the extracted abstractions from the source program. It is responsible for communicating with SP (to store all extracted abstractions) through the PopulateFacade, and QPS (to return all relevant information for answering a given query) through the QueryFacade. Internally, all extracted information is stored into smaller tables that each capture an abstraction (e.g. Follows, Uses etc.), and this process of storing information is controlled by Storage.

QPS is responsible for taking in PQL queries on the source program and returns the results of the query back to the UI. The Query Parser parses and extracts out keywords of the query, forming a SolvableQuery which is then passed to the Query Evaluator. From there, the Query Evaluator retrieves relevant information that is needed to answer the query and subsequently joins the results to get the final query result.

2 Component Architecture and Design

2.1 SP

Class Diagram

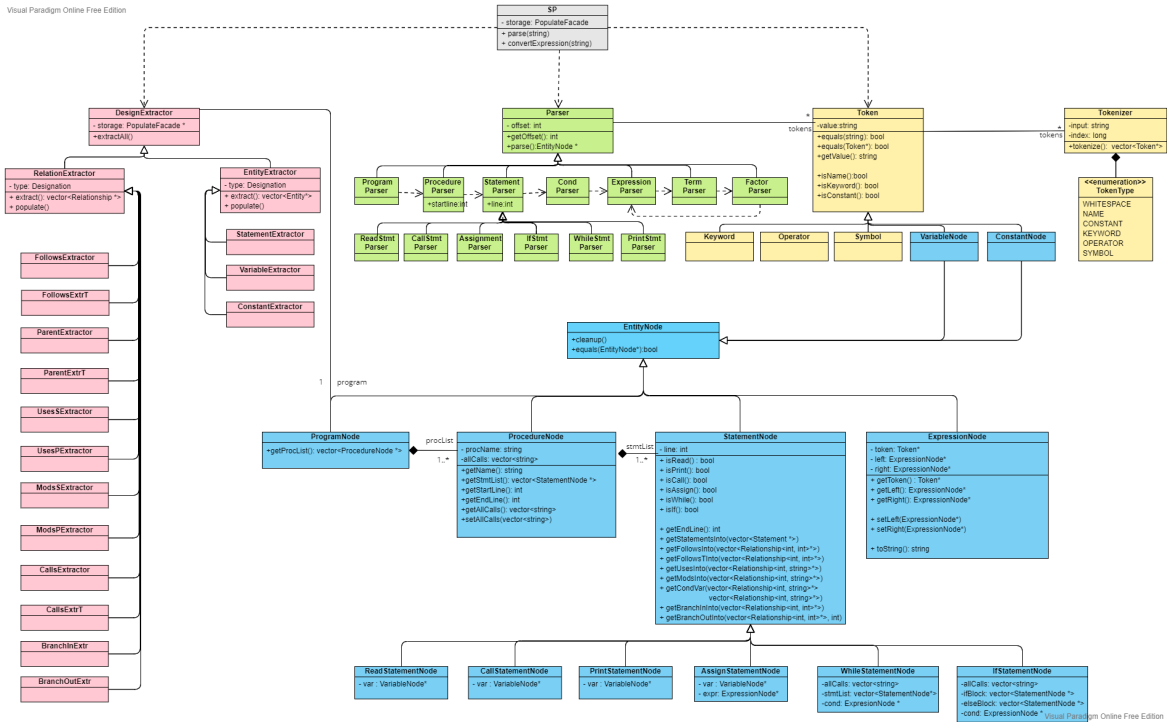

Figure 2 Class diagram of SP

Sequence Diagram

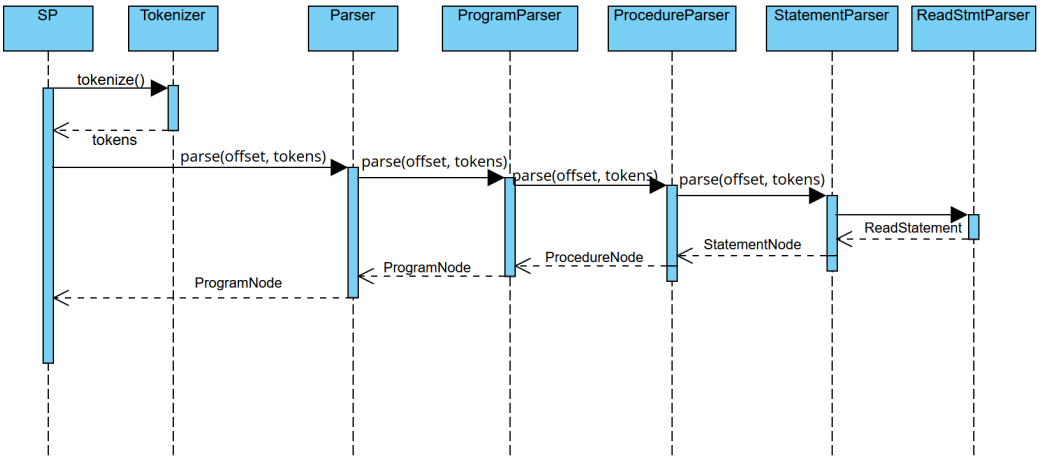

Figure 3 Sequence diagram of SP parsing a SIMPLE source program

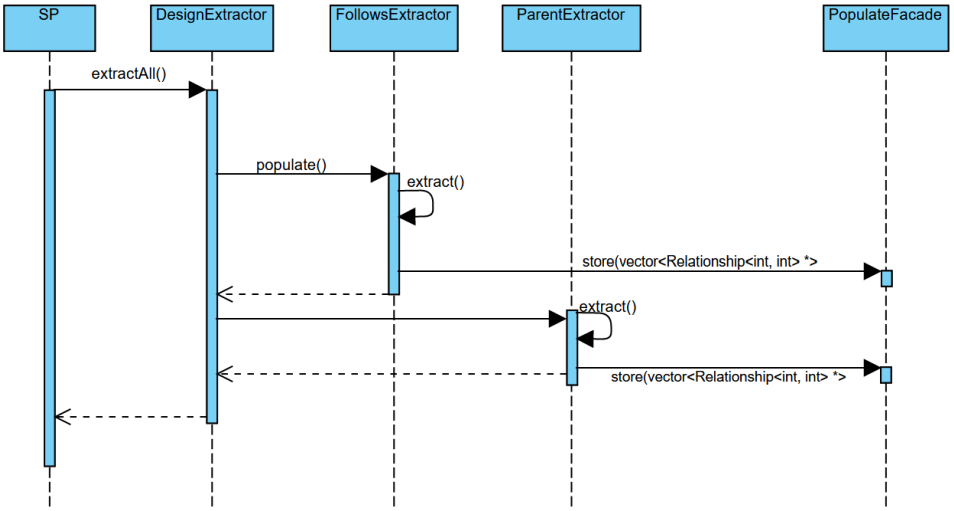

Figure 4 Sequence diagram of SP extracting from a parsed program

Application of design principles

- Single Responsibility Principle: SP is separated into Tokenizer, Parser, and DesignExtractor, each handling a step of analyzing the source program. Tokenizer tokenizes the source program into a list of tokens; parser parses the tokens into a EntityNode structured to be a tree; DesignExtractor extracts entities and abstractions from the Node and populates the data to PKB through PopulateFacade.
- Open-Closed Principle: All tokens inherit from the Token class; There is one parser class for each entity, which inherits from the Parser class; All extractor classes inherit from the DesignExtractor class. These allow for extensions of more tokens, entities, and extractors.
- Liskov Substitution Principle: Tokens, parsers, and extractors subclasses all implement their superclass's functions such as *getValue()* in Token, *parse()* in Parser, and *extract()* in EntityExtractor and RelationExtractor so that the subclasses' functions performs the same task as the superclasses. Thus subclasses behave the same way as their superclasses.
- Don't Repeat Yourself: Logics that are used in many extractors such as iterating through each StatementNode are abstracted out to reduce code duplication.

Use of design patterns

- Abstraction Occurrence Pattern: Each type of statement can be a subclass of EntityNode. However, they share the same information as statements. Thus, we create StatementNode as the abstraction, make various types of statement classes inherit from this class, and later create occurrences of these subclasses.
- Strategy Pattern: Parsing is an algorithm that reads a list of Tokens and processes them to produce an EntityNode. There are different parsing algorithms for each EntityNode, and they are each put into a separate parser class as function *parse()*. These parser classes all inherit from a superclass Parser, making their objects interchangeable.

Design Decisions

- AST: The source program is parsed into a tree structure since this can best represent the structure of a program. ie. every entity comprises several child entities. For example, a program is made up of several procedures, and a procedure is made up of several statements. Moreover, there are nested statements such as While and If-then-else clauses. These make it more important to keep the structure as a tree since a tree can easily represent a nested relationship by adding children to a node.

- Use of template in DesignExtractor: The original implementation finds each extractor class having its own *populate()* function despite having the same behaviour because of different return types. Template is therefore used so that *populate()* is only implemented once in the parent class.

Optimisation

- Memoization: When extracting Uses and Modifies, used and modified variables of a procedure are recomputed wherever there is a Call statement. Memoization of these variables is therefore implemented to avoid such recomputation.

2.2 PKB

Class Diagram

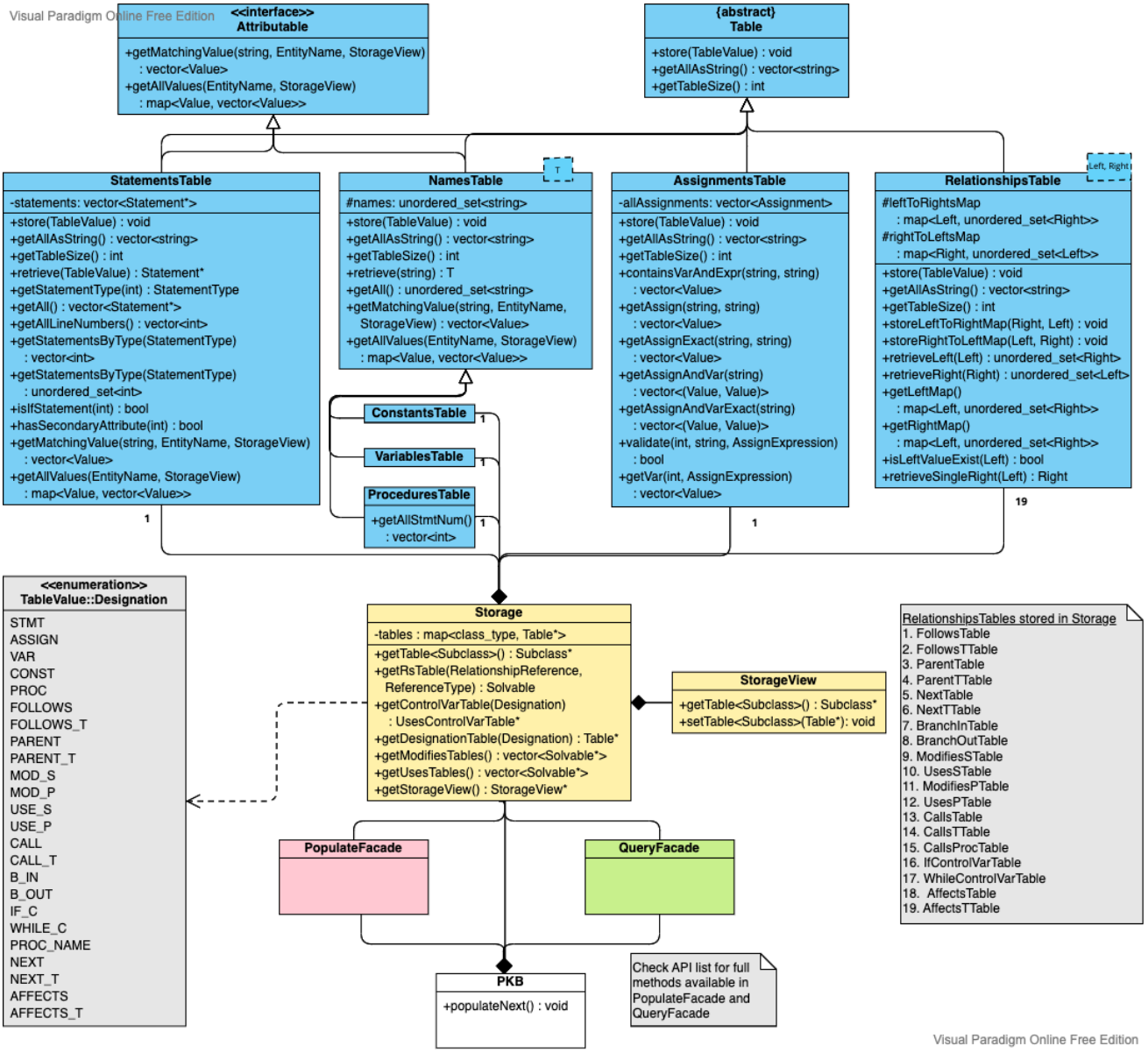

Figure 5 Class diagram of PKB

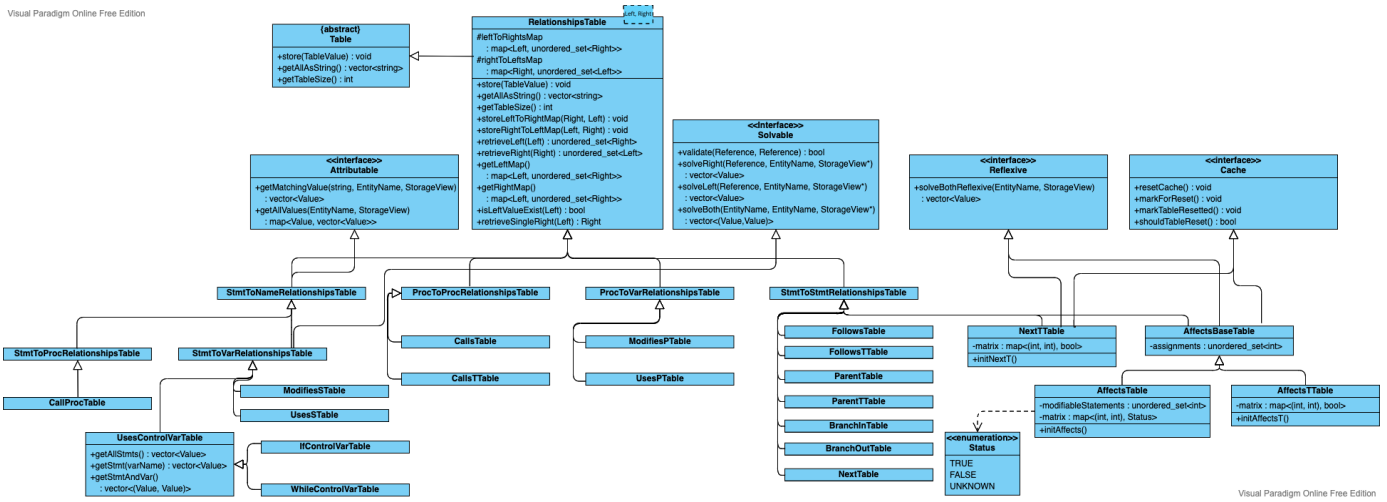

Figure 6 Class diagram of RelationshipsTables

Sequence Diagram

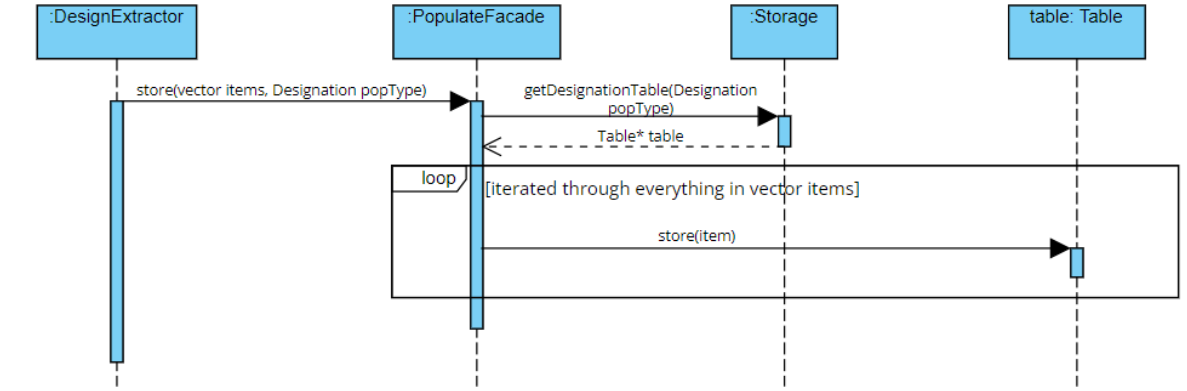

Figure 7 Sequence diagram of Design Extractor calling store in PopulateFacade

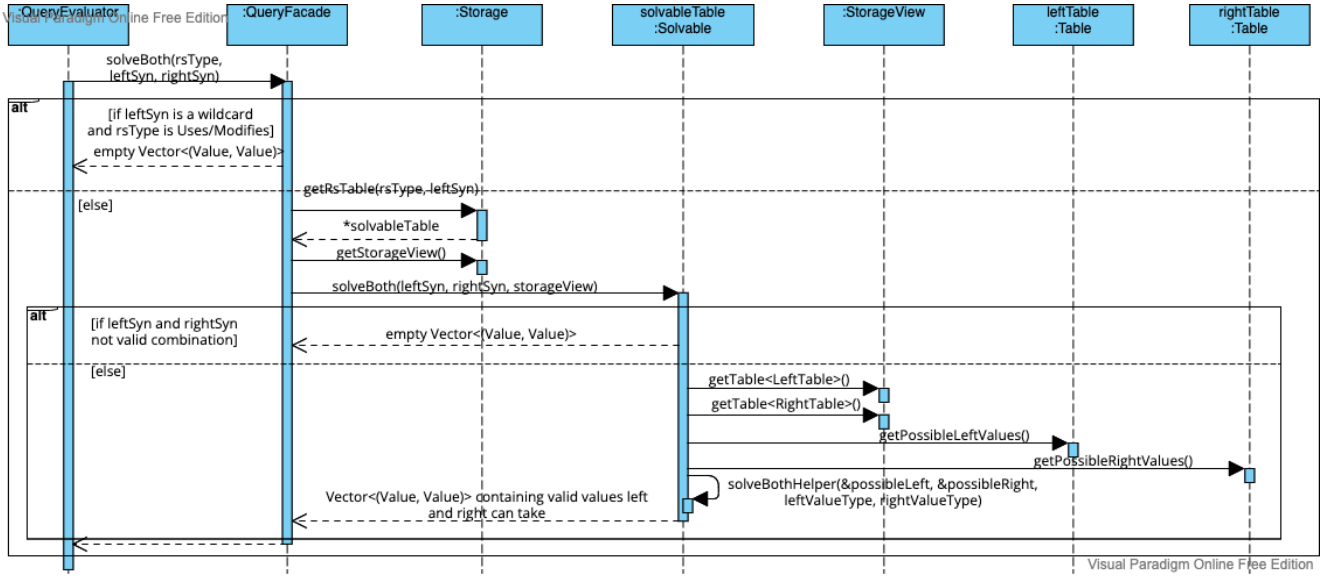

Figure 8 Sequence diagram of QueryEvaluator calling solveBoth in QueryFacade

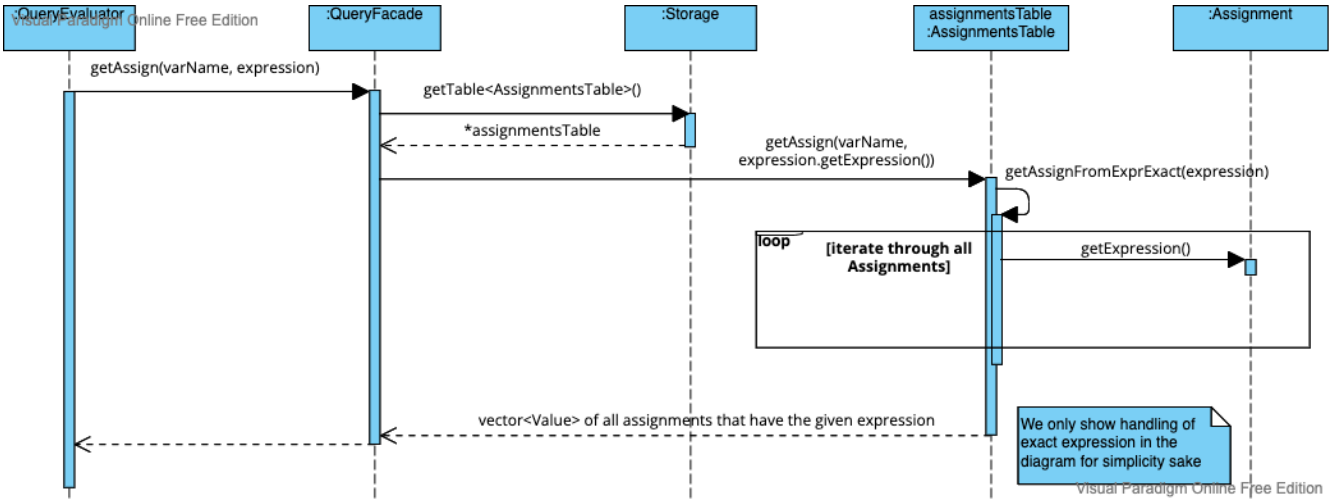

Figure 9 Sequence diagram of QueryEvaluator calling getAssign in QueryFacade to find all possible assignments that have the given expression

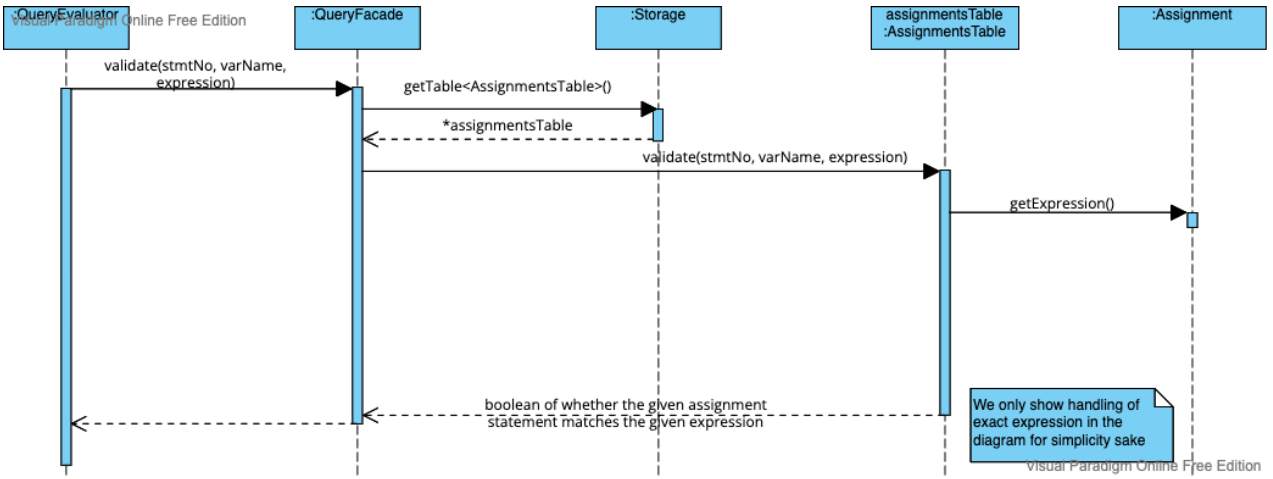

Figure 10 Sequence diagram of QueryEvaluator calling validate in QueryFacade, which is an optimised version of getAssign by checking the given statement number

Application of design principles

- Single Responsibility Principle: PKB has multiple responsibilities, namely: Interacting with Source Processor to store information extracted from the source program, interacting with Query Processor to return relevant information for answering a query, and handling storage of information into the respective tables. Thus, PKB is split into several smaller subcomponents each responsible for a portion of the functionality required. QueryFacade and PopulateFacade are designed to handle interactions with Source Processor and Query Processor respectively and Storage is designed to handle the storing of information to the different underlying tables. This separation of concerns enables code changes to only impact affected subcomponents, eliminating unexpected behaviour arising from propagation of code changes.
- Open-Closed Principle: This is done through abstracting out base classes like Table and Solvable, and also derivative classes like the subclasses of RelationshipsTable. These base classes represent common behaviour, and are closed for modification while open for extension to support new functionality.
- Liskov Substitution Principle: PKB ensures LSP by adhering to a proper inheritance hierarchy from Table, Solvable, Reflexive and Cache, while ensuring that substituting a subclass does not change the program behaviour in unexpected ways.

- Don’t Repeat Yourself: Abstractions are used to extract methods and classes from frequently occurring code and logic, reducing code duplication.

Use of design patterns

Facade design pattern: Used to hide the implementation details of PKB from the clients (Source Processor and Query Processor). It allows for the provision of a simplified interface to the clients through the form of specific APIs used for performing actions that each client will need, and prevents unintended behaviour that could occur if we allow clients to interact directly with the various PKB subcomponents. This allows for quick and easy leverage of PKB and decouples PKB implementation from clients.

Design Decisions

- No AST stored: Storing the entire AST of the source program to do retrieval is inefficient as compared to having proper tables that encapsulate each design entity/abstraction. Thus, we opt to use tables to ensure efficient partitioning of data. Assignment expressions are also converted to string for us to do string/substring matching as well.
- Use of templates in table design: When designing PKB, we realised that many of the tables have very similar designs (e.g. ConstantsTable, VariablesTable etc.), and only differed slightly in implementation. Hence, we decided to make use of templates in order to reduce code duplication and improve adherence to software engineering principles.
- Data structures used for storing data: As data storage and retrieval is one of PKB’s main roles in SPA, it is important to choose good data structures for efficiency and extensibility. Since ordering is not important in the final result, we used unordered sets and maps to store the various entities and relationships in SPA in order to achieve O(1) lookup and retrieval. However, in certain contexts where iteration through data is expected to occur frequently as opposed to lookup and retrieval (e.g. AssignmentsTable and StatementsTable), we used vectors since they would be faster due to contiguous memory allocation.

Optimisation

- Type of data stored: In storing the extracted information, we chose to use maps to store primitives (such as int and string) to reduce the amount of space overhead required to store other more complicated data like Relationships.
- Populating of NextTables internally as preprocessing step: After SP is done extracting the relationships in the program, we run a DFS algorithm through our internal tables (Follows, BranchIn, BranchOut) to extract Next relationships within PKB. By having the CFG constructed internally in PKB as well, we can easily reuse this for Next* and Affects/*. This would also save parsing time and memory usage, since this internal construction can be simplified to store only essential information, and duplicate data is less common.
- Provision of optimisation APIs: The previous set of APIs in Milestone 2 did not support optimisation, and did extra work when it was unnecessary. To help optimise query time, new optimisation APIs were provided to QPS with better time complexity guarantees. For example, validation of patterns can now be done at a provided line number instead of iterating through all possibile line numbers to find a match, and variable substitutions can be made to reduce query time complexity. Furthermore, by providing the size of each table, QPS can make more informed decisions on which sub-query to prioritise.

2.3 PQL

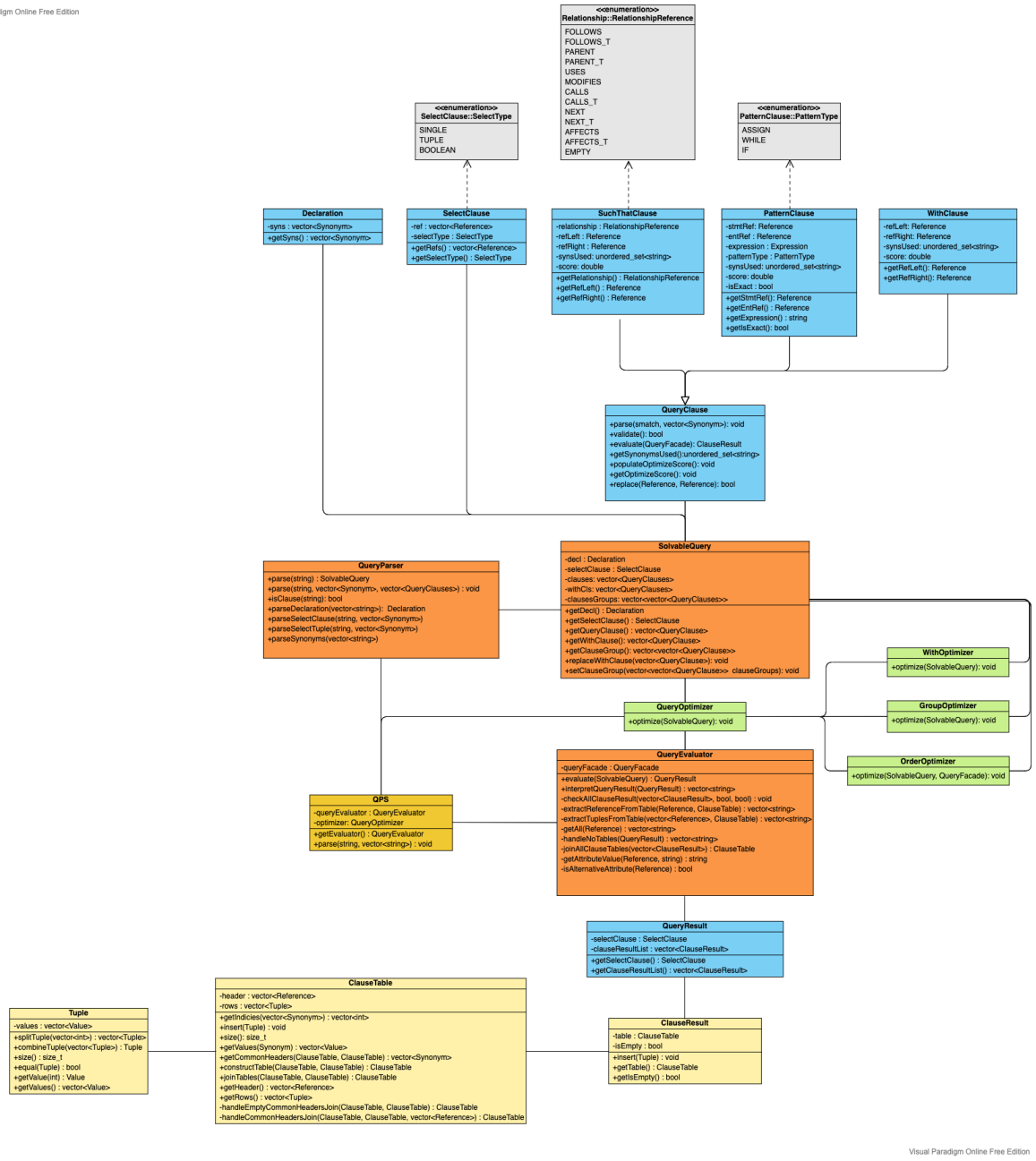

Figure 11 Class diagram of QPS

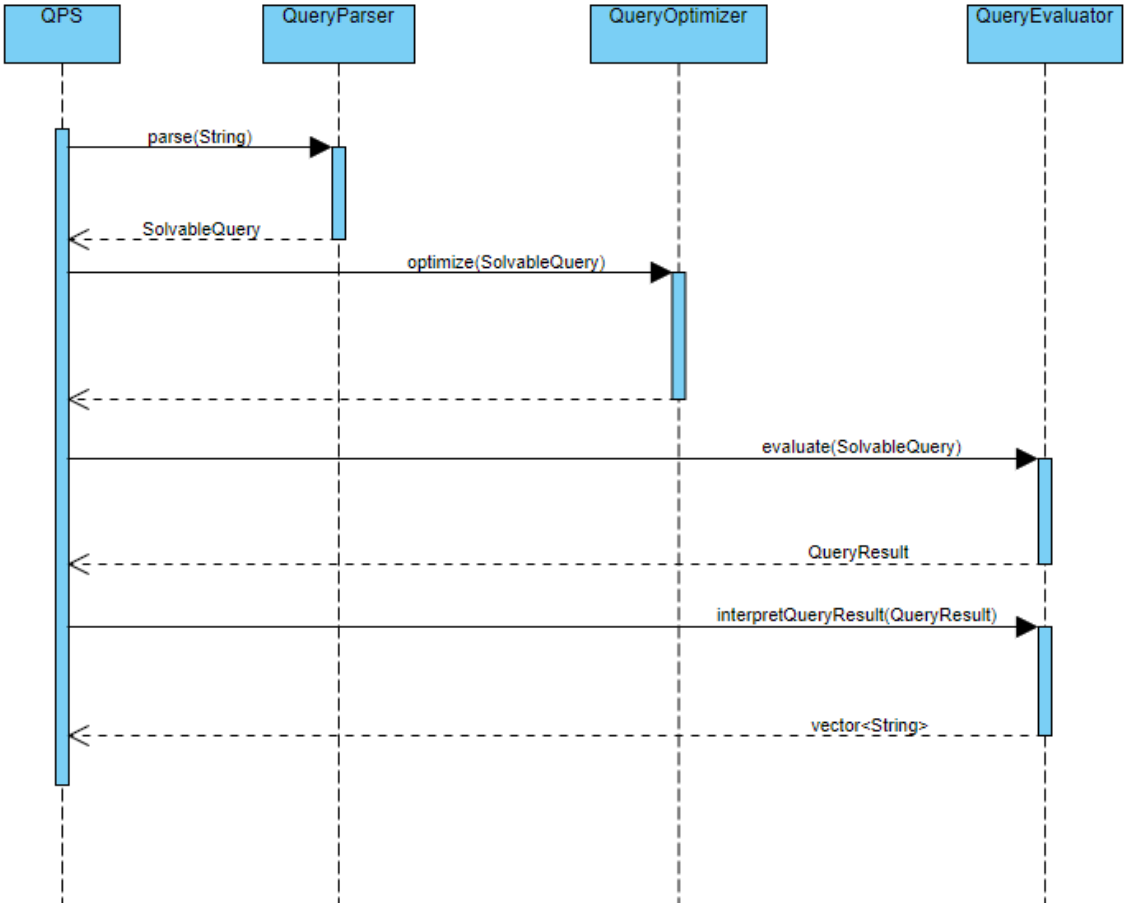

Figure 12 Sequence diagram for QPS

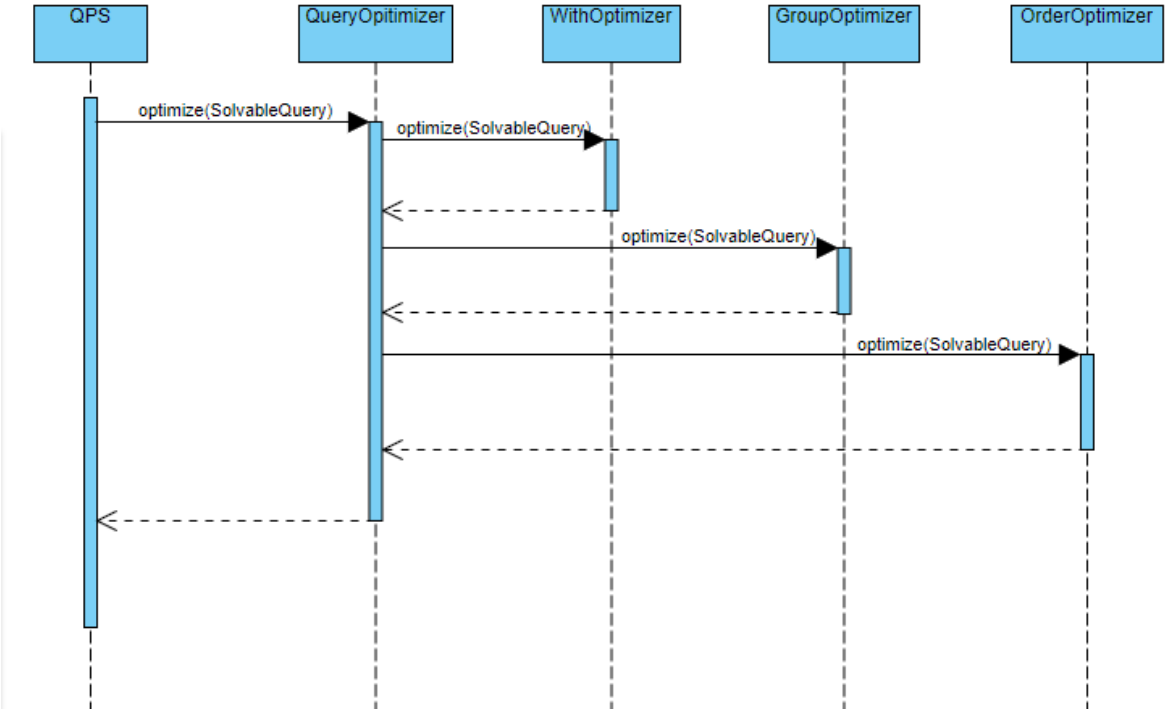

Figure 13 Sequence diagram of QueryOptimizer

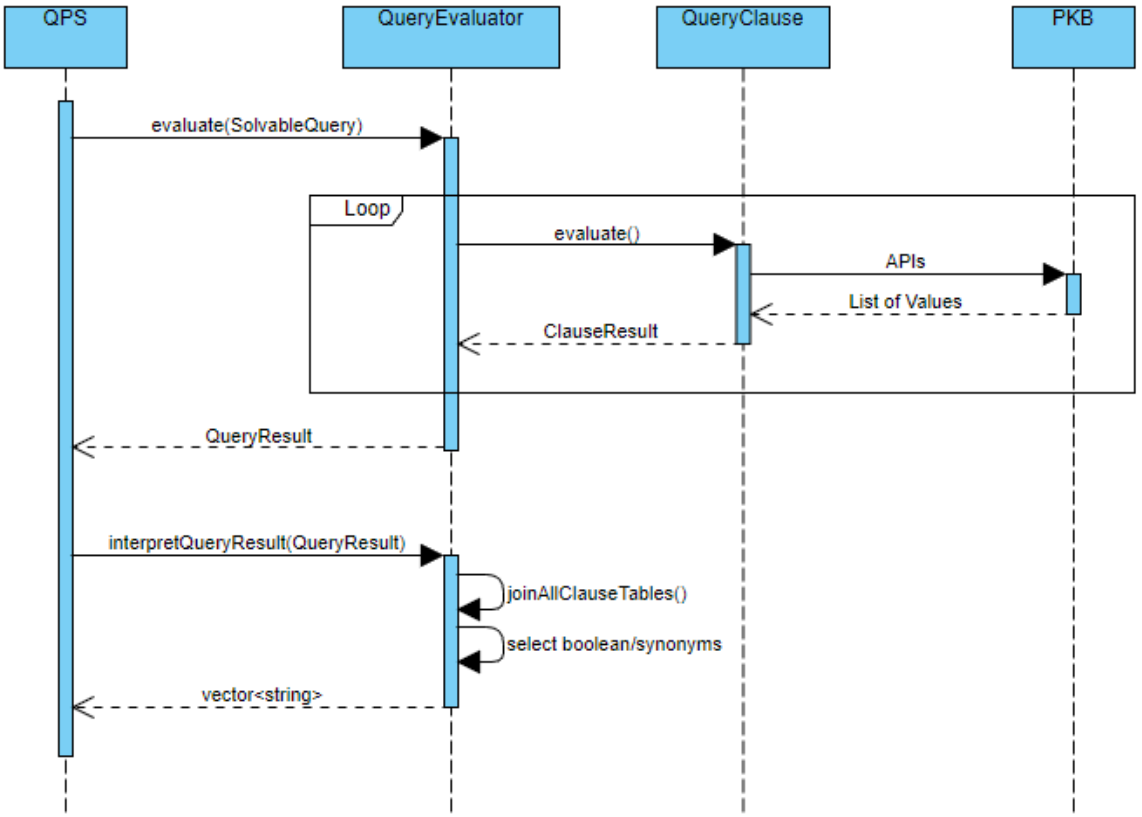

Figure 14 Sequence diagram of QueryEvaluator

Query (stmt s; assign a; variable v; read r)	Is replacement allowed?
Select s such that Parent(s, r) with r.stmt# = 2 => Select s such that Parent(s, 2)	**Can replace:** Replace all instances of synonym r in the such that clause with the value 2 and remove the with clause entirely.
Select v pattern a(v, _"x"_) with a.stmt# = 2 => Select v pattern 2(v, _"x"_)	**Can replace:** We are able to replace the assign, if and while synonym in pattern clause and evaluate it using new APIs from PKB.
Select r such that Parent(r, s) with r.stmt# = 2	**Cannot replace:** Synonym in with clause is present in select clause
Select s such that Parent(r, s) with r.varName = "x"	**Cannot replace:** With clause using secondary attribute
Select s such that Parent(r, s) with a.stmt# = 2	**Cannot replace:** Synonym in with clause is not present anywhere else

Figure 15 Cases for With Optimization

Application of design principles

- Open Closed Principle: SuchThatClause, PatternClause and WithClause all extend from a QueryClause base class which encapsulates the necessary behaviour for clause parsing, validation, optimization and evaluation. This base class is thus closed for modification while open for extension to support new behaviour of clauses.

- Liskov Substitution Principle: Each clause is responsible for implementing their superclass' function such as *evaluate()* but the return type and signature are all standardized. Thus, these subclasses behave the same way as the superclass.

Use of design patterns

- Strategy Pattern: The different clauses override the evaluate method of the base class. The QueryEvaluator can call QueryClass.evaluate() and at run-time, the different sub-classes can execute its own implementation of the evaluate method to get results from PKB.

Design Decisions

- ClauseTable representation: Each row of clause results are represented as a vector instead of using a column-oriented approach to reduce the time-complexity of result merging.

Optimisation

- Hash-Join Algorithm: We changed the algorithm for joining the intermediate tables from nested loop join to hash join.
- With Optimization: Optimization is done by replacing synonyms in the clauses with values defined in the with clause and removing the with clause from the SolvableQuery. However there are cases where this replacement would affect the meaning of the query and would affect correctness. Figure 15 above outlines the cases where replacement is allowed and not allowed.
- Group Optimization: Clauses that have common synonyms are grouped together for evaluation. The aim is to reduce the size of the intermediate result table formed during joining. There are 2 types of clause group: clauses without synonyms and clauses with overlapping synonyms. During group optimization, every clause returns its list of synonyms used. If it shares at least one synonym with any clause inside an existing group, the clause is added to this group.
- Intragroup Order Optimization: After the clauses are grouped, each of them is assigned a score based on how long it takes to be evaluated and how many result entries it might return. A lower score means they are given priority during evaluation. The scoring is based on the following criteria:
    - Type of clauses: with clauses are always evaluated first since they are the most restrictive, followed by such that and pattern clauses.
    - Number of synonyms: clauses with fewer synonyms are given lower scores.
    - Type of relationships: relationships with fewer results stored in PKB are given lower scores. Affects, Affects* and Next* are always evaluated last due to long evaluation time.
    - Type of synonyms: synonyms with fewer design entity entries stored in PKB are given lower scores.
- Intergroup Order Optimization: Clause groups are ordered by the cumulative score of the clauses inside the group.
- Early termination: With Order Optimization, we are able to order the clauses to be sent for evaluation from cheapest to most expensive for PKB to get the results. To fully reap the benefits, we implemented an EmptyTableError that is thrown the moment a clause is evaluated and the intermediate table is found to be empty. We can catch this error in the QPS class to return "FALSE" or the empty string depending on the select type. This allows early termination of the program and will prevent wasting resources on evaluating the more expensive clauses when a cheaper clause is found to be empty.

3 Test Strategy

In Milestone 3, there are 3 kinds of testing done, namely: Unit testing, Integration testing and System testing.

Unit testing is done for each method that is written, and includes both positive and negative testing. In order to avoid testing every possible combination, we wrote unit tests for common inputs, and tried to cover some edge cases. Through this, we are able to have reasonable coverage of the methods written whilst reducing the amount of bloat in our unit testing suites. Each subteam is responsible for writing unit tests that ensure the behaviour of their code is as expected.
Integration testing is done after a set of APIs promised by a subteam is delivered. The owner of the APIs is responsible for ensuring the outputs of the APIs are as intended and does not introduce bugs when they are called. The caller of the APIs is responsible for ensuring they can make the APIs call from their component and the outputs of the APIs are sufficient to deliver the new features. Test cases are jointly written by the relevant subteams (in adherence with the testing schema detailed in unit testing) to cover the APIs used for integration between the components.
System testing is done using the autotester. We came up with a checklist of cases to check for, as well as possible edge cases that we need to cover. Based on this checklist, we then wrote sample source programs and queries that cover the cases listed. Currently, the checklist consists of the following:

1. Number of such that/pattern/with clauses in query
    a. 0 clauses (Select clause only, test single return, tuple return, boolean return)
    b. 1 clause (Select clause + such that clause/Select clause + pattern clause + clause)
    c. Multiple clauses (Select clause + such that clause + pattern clause + with clause)
        i. 1 of each type
        ii. 2 of each type (explicit and, with)
        iii. Multiple of each type
2. Design abstractions
    a. Follows/FollowsT/Parent/ParentT (if/while, nested if/while)
    b. UsesS (+-, */%, brackets, print, assign, print nested in if/while, assign nested in if/while)
    c. ModifiesS (read, assign, read nested in if/while, assign nested in if/while)
    d. UsesP, ModifiesP, Calls, CallsT (multiple procedure calls, nested procedure calls)
    e. Next (if-then, if-else, while, nested if/while)
    f. Assign pattern clause (+-, */%, brackets)
    g. While pattern/If pattern (if/while, nested if/while)
    h. With clause (procedure.procName, call.procName, variable.varName, read.varName, print.varName, print.varName, constant.value, stmt#)
        i. Used in clauses (Select, with)
        ii. Invalid queries
3. Entities
    a. stmt/read/print/while/if/assign
    b. procedure/variable/constant
4. Number of wildcards
    a. 0/1/2/3/4 wildcards
5. Number of synonyms
    a. 0/1/2/3/4 synonyms
6. Overlaps in clauses
    a. 0/1/2 synonym overlap between with clauses such that and pattern clause
    b. Selected synonym appears in clauses
    c. Selected synonym does not appear in clauses

7. Level of nesting
    a. Direct (Nested in if/else/while block)
    b. Indirect (Nested in chained if/while/if + while blocks)

Edge cases:

1. Missing/additional special symbols: **“_;,.=<>()**
2. Missing/additional/misspelling special keywords: “select”, “and”, “pattern”...
3. Missing/additional arguments
4. Invalid arguments
5. Undeclared synonyms
6. Mismatched “and” (such that and pattern)
7. Same synonyms used in both arguments
8. Wrong usage of wildcard: in select clause, as argument,..
9. Spaces between tokens
10. With clause attribute with additional spaces
11. Reflexive with clause (e.g. “call” = p.procName and p.procName = “call” are equivalent)
12. Additional arguments in pattern-if/pattern-while

With the implementation of the aforementioned tests, we then used these test cases to guide development and for regression testing. Before any pull request is merged, team members are expected to run the unit_testing executable, integration_testing executable and the autotester script (detailed in automation approach) to ensure their changes do not break existing functionality, and are compatible with the expected behaviours of SPA. On top of that, team members are to ensure their code follows the codestyle guidelines of the project. To enforce this workflow, each pull request is opened with a template that documents these steps in a checklist. Team members are expected to complete all items on the checklist and check them off accordingly before the pull request goes into review. A CI pipeline (built with GitHub Actions) will ensure adherence with codestyle guidelines, passing of all unit and integration tests, and generate all system test files (detailed in automation approach). Before merging, team members are expected to have fulfilled all items in the CI pipeline.

Test statistics

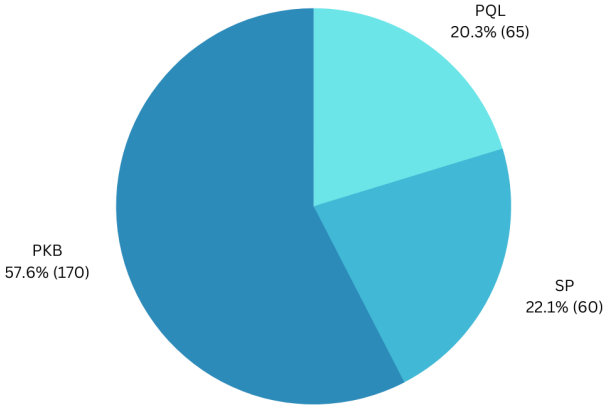

Figure 15 Test Breakdown by Component

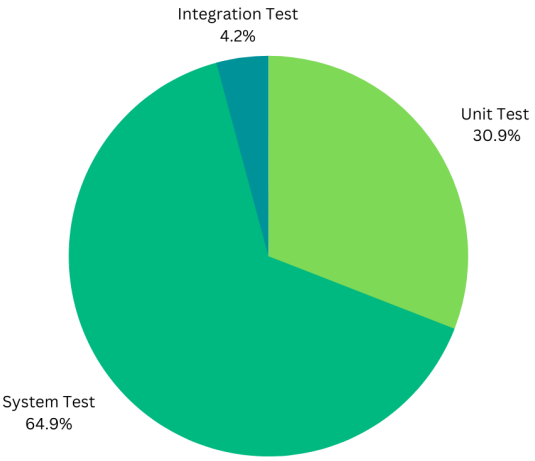

Figure 16 Test Breakdown by Type

Automation approach

To speed up pace of development, we automated the process of running tests and codestyle checks through a CI pipeline built with GitHub Actions. When a pull request is put up, 4 GitHub Actions are run:

1. Unit testing (To ensure all unit tests pass)

2. Integration testing (To ensure all integration tests pass)
3. System testing (To generate all system test results using autotester, and compile them into a zipfile)
4. CLang-Format (To ensure adherence with codestyle)

In particular, system testing is run through a Python script written to iterate through all sources and queries placed in the respective subfolders that follow a given format, and run the autotester to generate the XML result files. This way, new system tests can be automatically run as long as the naming conventions are followed.

All pull requests are expected to pass all 4 checks, as well as be given an approving review, before it can be merged into the codebase. Through this CI pipeline, less time is spent on configuring and running tests, allowing developers to focus on the code and saving precious development time whilst not compromising on code quality and correctness.

Bug handling approach

Any bugs surfaced during the development process will be reported to the GitHub issues tracker. The issues tracker has a standardised bug report format that team members adhere to, and use to provide comprehensive bug descriptions and replication instructions. Furthermore, bugs are tagged with the bug tag and a corresponding component tag (SP/PKB/QPS) so that each subteam is aware of which bugs have surfaced. Team members will then decide within their subteams how to handle the bug, and put up an appropriately-named pull request with the fix. Approval is given upon thorough testing following the aforementioned test approach. Once approved, the fix is merged and the issue is closed.

4 Reflection

Overall, project scheduling was smooth, and we managed to hit our deliverables at every sprint without much issue. However, at busier periods of the semester, there often were clashes in schedules, and development pace would be impeded by these clashes. To overcome these, we would discuss as a team about our schedules for the upcoming weeks (especially nearing submission deadlines and busy periods), and sound out any potential clashes and issues well ahead of time. We worked together to come up with reasonable compromises for everyone, and scheduled project deliverables to fit everyone's schedules. Another thing that helped was that all members had clear responsibilities regarding development, and we sought to minimise overlaps that could impede progress by clearly delegating work to each member. This way, development could occur in parallel and there is less chance of deadlocks that could slow development.

That being said, there were still times when unexpected regression bugs/roadblocks popped up nearing deadlines, resulting in us having to drop some features just before code demonstrations/submissions. Moving forward, a sensible approach to dealing with this issue would be twofold:

1. Be more conservative in feature planning, and aim for a higher quality product over speed of development
2. Employ test-driven development to clearly set out requirements before any features are developed

Regarding system design, we had a clear plan going into development, and each subteam had clearly set out a high-level architecture of each subcomponent. Throughout the project, we made sure to design our codebase in a way that is consistent with the high-level architecture, and did our best to adhere to software engineering best practices. Furthermore, we did cross-reviews of each subcomponent's code, and pointed out violations in code quality ahead of submissions to give

subteams time to rectify any issues. Overall, this approach to system design worked well for us, and the code quality is of a reasonably high level. However, we realised that cross-reviews were somewhat contentious in practice, and oftentimes led to disagreements on best practices. Moving forward, a more sensible approach to cross-review might be to clearly set out expectations for code quality, and review based on a more objective metric (e.g. a code review checklist).

Regarding team dynamics and management, we had a rocky start due to our unfamiliarity with each other and difference in working styles. Over time, we began to understand each others' working styles better, and adapted development and workload distribution to fit each member's needs. We ended the project on a high note, with each member contributing adequately. However, as with any project, conflicts have occurred at times due to workload distribution and scheduling conflicts. We resolved these conflicts by communicating openly about any issues we might have, and making compromises to ensure all members are satisfied with the resolution. Moving forward, a sensible approach would be to lay out more clearly the time commitments at the start of each sprint, and iron out potential issues as soon as they occur in order to improve team dynamics and facilitate open communication.

5 Appendix

5.1 APIs

5.1.1 SP

void parse(string filename)
Parses the program in the given filename, and stores all extracted Entities and Abstractions into PKB via PopulateFacade.6

std::string convertExpression(std::string input)
Converts an expression into a fully bracket-ed format. Provided to QPS to help parse expressions.

5.1.2 PKB

PopulateFacade APIs

void store(std::vector<T *> *items, Populate popType)
Stores all items of a given type T into the corresponding table specified by popType.

QueryFacade APIs

void resetCache()
Clears cached results for Next*, Affects and Affects*.

int getTableSize(Designation desType)
Returns the total number of entries stored for a given Designation

std::vector<Statement *> getAllStatementsByType(StatementType type)
Returns all statements inside the source program that correspond to a given StatementType.

std::vector<std::string> getAllEntities(Designation entity)
Returns all names inside the source program based on the designated entity.

bool validate(RelationshipReference relType, Reference left, Reference right)
Returns a boolean indicating if a provided relationship type holds between the left reference and right reference.

bool validate(int stmtNo, std::string varName, AssignExpression expr)
Returns a boolean indicating if a pattern-assign holds at a given stmtNo.

bool validate(Designation desType, int stmtNo, std::string varName)
Returns a boolean indicating if pattern-if/pattern-while at a given stmtNo contains varName.

std::vector<Value> solveRight(RelationshipReference relType, Reference leftRef, EntityName rightSynonym)
Returns all possible values that the right synonym can take on based on the provided relationship type.

std::vector<Value> solveLeft(RelationshipReference relType, Reference rightRef, EntityName leftSynonym)

Returns all possible values that the left synonym can take on based on the provided relationship type.

std::vector<std::pair<Value,Value>> solveBoth(RelationshipReference relType, EntityName leftSynonym, EntityName rightSynonym)
Returns all possible pairs of values that the left and right synonyms can take on based on the provided relationship type.

std::vector<Value> solveOneAttribute(Reference ref, Value value)
Returns all possible values that a synonym can be based on the given value and its attribute.

std::vector<std::pair<Value,Value>> solveBothAttribute(Reference left, Reference right)
Returns all possible pairs of values that the left and right synonyms can take on based on their attributes.

std::vector<Value> getAssign(std::string varName, AssignExpression expression)
Returns all possible values of assignments that satisfy the given varName and partial/wildcard expression.

std::vector<std::pair<Value,Value>> getAssignAndVar(AssignExpression expression)
Returns all possible pairs of assignments and variables that satisfy the given expression.

std::vector<Value> getCond(Designation desType, std::string varName)
Returns all possible values of conditional statements that satisfy the given varName based on the designated conditional desType (If or While).

std::vector<std::pair<Value, Value>> getCondAndVar(Designation desType)
Returns all possible pairs of Cond and Variable based on the designated conditional desType (If or While).

std::string getSecondaryAttribute(int stmtNum)
Returns the secondary attribute of the given stmtNum of a Print, Read or Call statement.

std::vector<Value> solveReflexive(RelationshipReference rsRef, EntityName stmtEntity)
Returns all possible values of conditional statements that satisfy a reflexive relationship.

std::vector<Value> getVar(int stmtNo, AssignmentExpression expr)
Returns variable used in an assign-pattern at given stmtNo.

std::vector<Value> getVar(Designation desType, int stmtNo)
Returns variables used in an assign-if/assign-while at given stmtNo.

5.1.3 PQL

void evaluate(std::string query, std::list<std::string> &results)
Evaluates the given PQL query and stores results in the list provided.

5.2 Extension Proposal

Definition

Exception (abbreviated Exc): An event that occurs during the execution of a SIMPLE program that disrupts the flow of the program's instructions. There will be 2 new design entities: **exception** and **throw (statement)**. The relevant updates to the grammar rule definition in SIMPLE would be as follows.

```
exception_name: NAME
throw: 'throw' exception_name
```

There will also be a new design abstraction:

Throws / Throws*

```
For any procedure p and exception e:
Throws(p, e) holds if procedure p directly throws e.
Throws*(p, e) holds if prodecure p directly or indirectly throws e. That is:
    -  Throws (p, e) or
    -  Calls(p, p1) and Throws*(p1, e) for some procedure p1
```

PQL will also have to be updated to support Such-that and with clauses for this new relationship. These relevant updates to the grammar rules in PQL would be as follows:

```
design-entity: 'stmt' | 'read' | 'print' | 'call' | 'while' | 'if' | 'assign' |
               'variable' | 'constant' | 'procedure' | 'exception'
attrName: 'procName'| 'varName' | 'value' | 'stmt#' | 'excName'
Throws: 'Throws' '(' entRef ',' entRef ')'
ThrowsT: 'Throws*' '(' entRef ',' entRef ')'
```

Namely, add exception synonyms and Throws/Throws* into the possible such that clause relationships. For with clauses (with e.excName), it will be similar to procName and varName where the attribute refers to NAME.

Changes required to existing systems and its components

For SP, support has to be added for the new grammar rules of the design entities outlined above. A new ThrowStmtNode class and its parser class will be created to parse throw statements. A ThrowExtractor class will extend DesignExtractor to extract exceptions each procedure throws.

For PKB, there are no major changes in design required to support the new design entities and abstractions. The current design of PKB can be extended without modification of existing classes to add support for Throws/Throws*, and will be detailed further in Implementation details.

For QPS, minor changes need to be added for the new PQL grammar rules outlined above. We used regex to match with such that clause syntax and extract out relationship references and the two arguments to the relationship, therefore we would need to modify the regex used for extracting information from such that clauses and with clauses, add "throws" and "throws*" into the map of string to relationship references and add the valid arguments into the list of relationships arguments to allow checking of valid semantics. We would also have to add the mapping of "excName" to "exception" as a possible attribute to ensure we can parse with clauses for exceptions correctly.

Implementation details

To support the new entity and abstractions in SPA, we have to update the enum classes in our commons library, such as the Designation enums to include the new design entity and abstractions (Designation::EXC, Designation::THROWS, Designation::THROWS_T), as well as extend other relevant enums such as RelationshipReference (RelationshipReference::THROWS and RelationshipReference::THROWS_T) to support the new relationships and the StatementType enum (StatementType::EXC) to include the new type of statements.

A new Exception class would have to be implemented in the commons library to encapsulate the new entity Exception and this Exception class will subclass from Entity to allow for polymorphism.

To support the new attribute excName for exception, we would also have to extend our EntityAttribute enum (EntityAttribute::EXC_NAME). Other than that, no other classes would have to

be implemented as existing classes such as Statement and Relationship would be able to encapsulate the extensions well due to the use of enums and templates.

For SP, we need new classes to parse throw statements and extract exception information. A new ThrowStmtNode and its corresponding parser class will be created. A new attribute 'throwList' will be added to ProcedureNode to store a list of exception names that are directly thrown in that procedure. To extract exception information, ThrowExtractor and ThrowExtrT will be implemented to go through the throwList of each ProcedureNode, for ThrowExtrT it will also go through all ProcedureNode that are directly or indirectly called using dfs.

PKB has to implement a new table to support storage of this new information.
The new entity Exception will be stored in an ExceptionsTable, which subclasses from NamesTable<Exception>, and this will allow us to store the exception_names to answer PQL queries that may ask for it, such as: `exception e; Select e`. By subclassing from NamesTable, we also ensure polymorphism for the QueryFacade's getAllEntities method, as well as APIs that allow us to retrieve the entity's attribute such as the currently unimplemented getAttribute to support with clauses.
To store the relationships, a ThrowsTable and a ThrowsTTable could be implemented as subclasses of ProcToExcRelationshipsTable that subclasses from RelationshipsTable since it preserves the structure of a Relationship. ProcToExcRelationshipsTable would also implement the Solvable interface, allowing QueryFacade to use polymorphism for its solve APIs for Such-that clauses.

PopulateFacade's store method would not require modification since we have extended the Designation enum class in the commons library, and ThrowsTable and ThrowsTTable only needs to be instantiated and added to Storage with the following mappings to ensure we are retrieving the correct tables in the QueryFacade:

```
Storage::tables[typeid(ExceptionsTable)] = ExceptionsTable
Storage::designTables[Designation::EXC] = ExceptionsTable

Storage::tables[typeid(ThrowsTable)] = ThrowsTable
Storage::designTables[Designation::THROWS] = ThrowsTable
Storage::rsTables[RelationshipReference::THROWS] = ThrowsTable
ThrowsTTable will be similar with its corresponding classes and enums.
```

For QPS, a few changes would be needed to support the new PQL grammer.
First, we need to allow for exception synonyms. we need to add

```
{"exception", EntityName::EXCEPTION}
```

into ENTITY_MAP in the QueryParserRegex.h file.

We need to modify the regex used for extracting information from such that clauses. More specifically, we need to add

```
"Throws|Throws\\*"
```

into SUCH_THAT_CL_REGEX, SUCH_THAT_REGEX in the QueryParserRegex.h file to allow the QueryParser to identify and match with the new such that clause syntax.

We also need to add the throws and throws* to the data structure that maps the relationship string to the RelationshipReference enum class for logical representation. Therefore we need to add

```
{"Throws", RelationshipReference::THROWS},
{"Throws*", RelationshipReference::THROWS_T}
```

into RELATIONSHIP_MAP in the QueryParserRegex.h file.

To support the new arguments for with clause, we will add a mapping from exception to excName to the data structure that maps the entityName to attribute type for logical representation. Therefore, we need to this mapping into ENTITY_ATTRIBUTE_MAP in Entity.h:
`{EntityName::EXCEPTION, EntityAttribute::EXC_NAME}`
At the same time, to parse this attribute correctly, we need to add the following mapping into ENTITY_ATTR_MAP `{"excName", EntityAttribute::EXC_NAME}`, and update the ATTR_REF_REGEX in the QueryParserRegex.h file to include

```
"\\s*(procName|varName|value|stmt#|excName)\\s*"
```

After that, we need to allow the QueryParser to identify semantic errors due to invalid arguments. This is done by adding the allowed argument types into RELATIONSHIP_LEFT_ARG_MAP, RELATIONSHIP_RIGHT_ARG_MAP, RELATIONSHIP_LEFT_REF_MAP and RELATIONSHIP_RIGHT_REF_MAP.
After the above changes, the QueryParser would be able to match with the new grammar and catch semantic errors and would be able to create the SolvableQuery object to represent the new relationship abstraction. The QueryEvaluator does not need any changes as it is decoupled from the syntax rules of PQL.

Possible challenge(s) and mitigation plans

Implementing new entities and abstractions would involve us being able to correctly update the grammar rules, extract them from the SIMPLE source, store them, parse and evaluate new PQL queries and be able to correctly retrieve these new abstractions from PKB. There are multiple points of failure when it comes to implementation, such as not being able correctly store or retrieve Throws* relationships due to its complicated nature, or not accepting 'throws throws' as a valid throws statement. This may make it challenging for us to correctly implement the extension, potentially missing out on edge cases.
Proper testing will help us combat the challenge mentioned above. By having a well-defined set of behaviours and a well-designed suite of test cases that check for edge cases and each type of possible behaviour from the very start, we would be able to verify whether our implementation is correct, and can even do test-driven development to tackle each behaviour systematically and without missing any out.

Benefits to SPA

Exceptions are a common feature in most modern programming languages precisely because the programs we write oftentimes require a point at which execution should not continue, and should instead be stopped. However, the current implementation of the SIMPLE programming language and SPA has no good way to handle cases where program execution is interrupted.
This extension gives SPA much more utility, since it allows users to figure out which portions of the SIMPLE program have a potential point of failure. This allows users to effectively separate details of what to do when something out of the ordinary occurs from the main logic of the program. The availability of Throws as a possible query in SPA allows users to see at a glance which portions of the SIMPLE program could lead to an interruption in the normal program flow. Consequently, users would be made more aware of these potentially dangerous sections of the program, and this would prompt them to be more careful in program design and making assumptions about the program.
With exceptions implemented, it also gives us the opportunity to add try-catch statements in SIMPLE, allowing us to create a more robust language and support even more complicated queries in PQL.