CS3203 Software Engineering Project

AY22/23 Semester 1

Project Report – System Overview

Team 34

Team Members	Student No.	Email
Tan Zhuo Yao	A0199730U	e0406711@u.nus.edu
Gao Haochun	A0194525Y	e0376955@u.nus.edu
Wang Pei	A0194486M	e0376916@u.nus.edu
Lye Jia Yang	A0217370L	e0543406@u.nus.edu
Nauman Sajid	A0196616U	e0389102@u.nus.edu
Gao Gui	A0194562X	e0376992@u.nus.edu

Consultation Hours: Tuesday, 12pm

Tutor: Fyonn Oh

1. System Architecture **3**

2. Component Architecture and Design **3**

2.1. SP 3

2.1.1. Diagrams 3

2.1.2. Design Decisions 5

2.2. PKB 6

2.2.1. Diagrams 6

2.2.2. Design Decisions 8

2.3. QPS 9

2.3.1. Diagrams 9

2.3.2. Design Decisions 15

3. Testing **16**

3.1. Automating Testing 17

3.2. Testing Statistics 17

3.3. Design Strategies 17

3.4. Types of Tests 18

3.5. Bug Handling 18

4. Reflection **19**

4.1. Problems encountered 19

4.2. Good practices maintained 19

4.3. Release retrospection 19

5. Appendix **20**

5.1 API Listing 20

5.1.1 List of APIs for SP 20

5.1.2 List of APIs for PKB 20

5.1.3 List of APIs for QPS 21

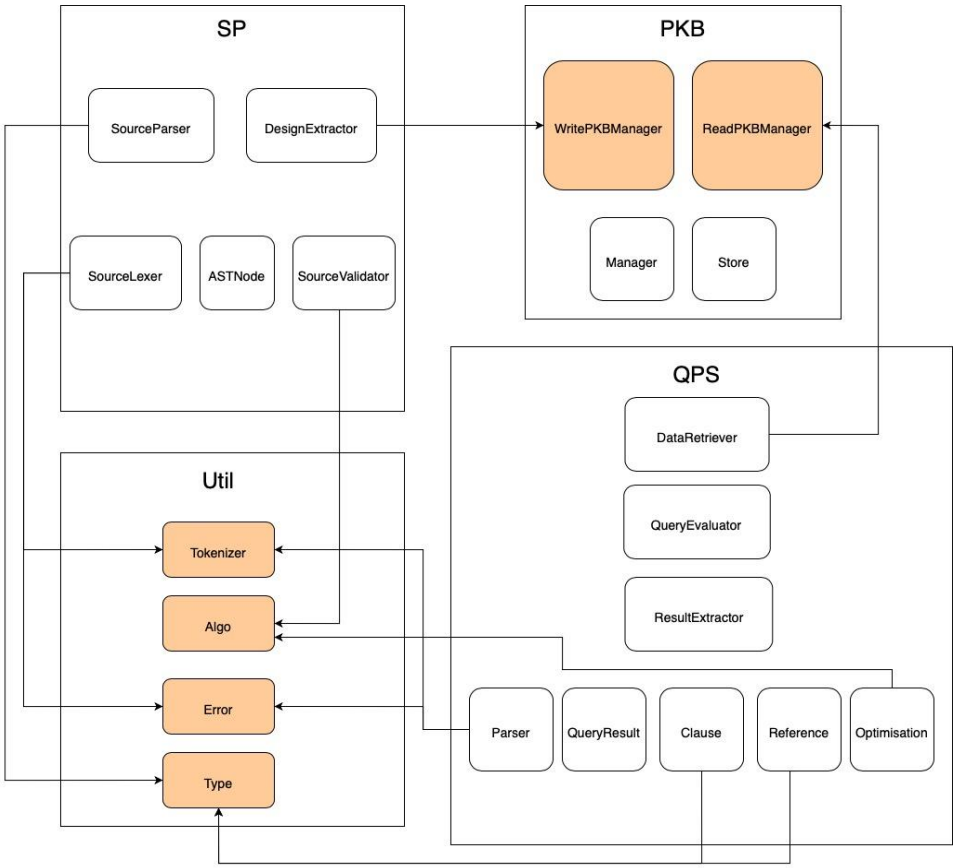
5.2 extension proposal 21

5.2.1. Definition of the extension 21

PQL Grammar 21

5.2.2. Changes required to your existing system and its components 22

1. System Architecture

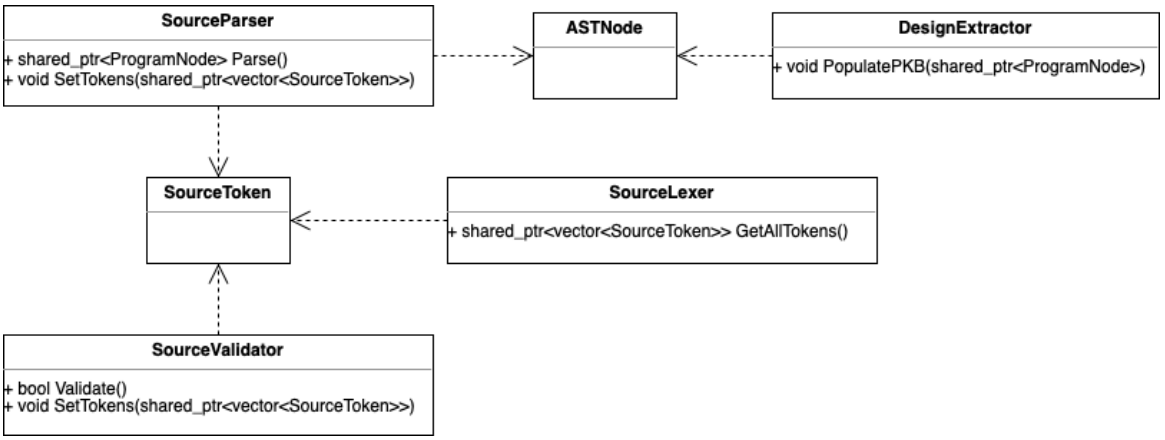


2. Component Architecture and Design

2.1. SP

2.1.1. Diagrams

High level overview of SP components



- `SourceLexer`: Tokenize the given source file to a vector of `SourceToken`.
- `SourceValidator`: Check that the tokens represent a valid SIMPLE program.
- `SourceParser`: Parse a valid vector of tokens into a tree of `ASTNode` representing the source program.
- `DesignExtractor`: Extract entities and design abstractions from `ASTNode` and write them into the PKB.

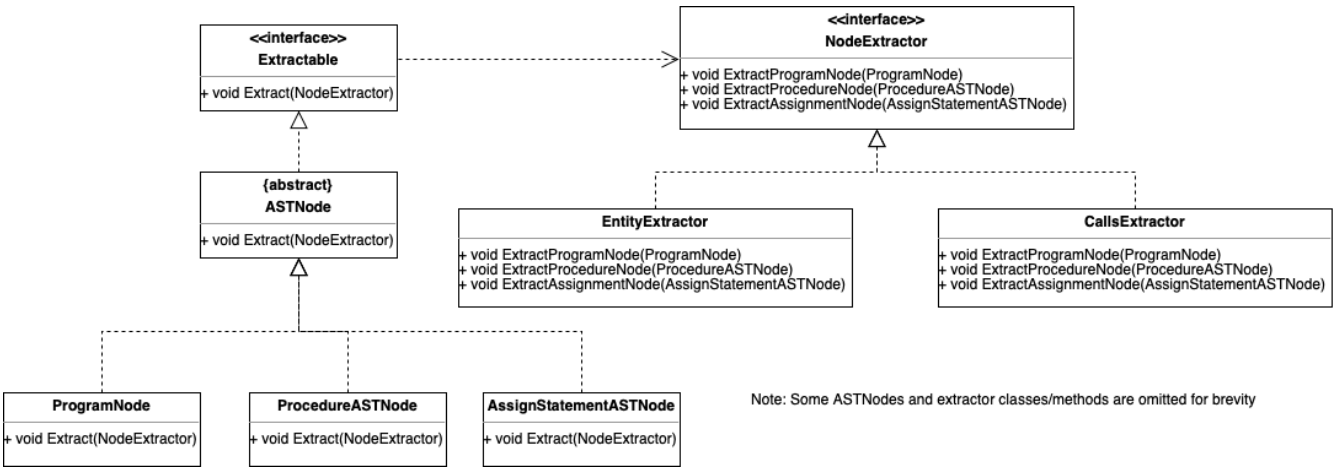
SP components are mainly designed following the Single Responsibility Principle, where each component has a well-defined role. This is in contrast with a one-pass design, where validation, parsing and extraction is done simultaneously.

Well defined components	Few components that do everything
Follows SRP	Violates SRP, possibly results in anti-pattern of God Object.
Eases parallel development	Hard for coordination within component
Requires multiple pass of source tokens, less efficient	Possible to extract and parse together, more efficient

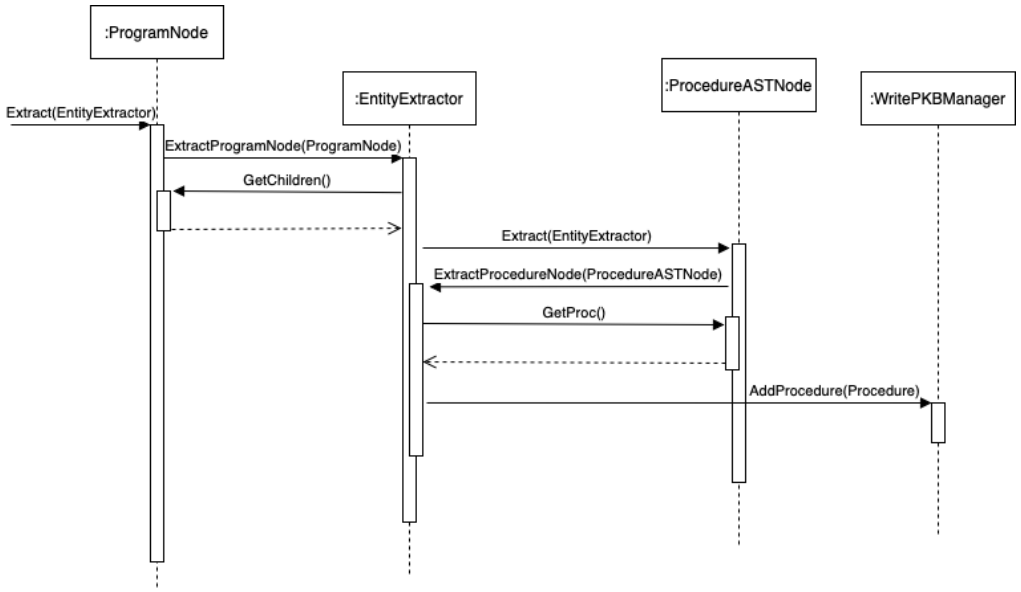
Link to class diagram for ASTNode:
https://drive.google.com/file/d/19svpp56Nf6A8P-IavVniWRkRZDayax36/view?usp=share_link

We use a clear hierarchy of `ASTNodes` to represent the SIMPLE program, such as using inheritance by having a parent class of `StatementASTNode` with specific subclasses for each type of statement.

Class Diagram for DesignExtractor
Application of Visitor pattern with `ASTNode` as the `Visitable`, and `NodeExtractor` as the `Visitor`



Example interaction during design entities extraction



2.1.2. Design Decisions

Application of design principles

- DRY: The use of a common base `Tokenizer` between SP and QPS components. Since both major components require parsing of input, the algorithm to support the tokenization can be shared using this common class. A lexer can then be used to generate language-specific tokens for each component.

- Open-closed principle: When there is a need to support new design abstractions, the `DesignExtractor` can simply implement a new concrete `NodeExtractor` to support it. Thanks to the Visitor pattern, the existing `ASTNode` classes will not have to be modified to support this extension.

- Single-responsibility principle: Instead of having a single node class in charge of all design entities, each class has its own responsibility through inheritance. For example, `StatementASTNode` serves as the base class of concrete statement nodes such as `IfStatementASTNode`. This allows these base classes to have their own attributes (condition expressions for if and while, variables for print and read)

- Inversion-of-Control: Concrete subclasses of `NodeExtractor` accept a `WritePKBManager` instead of creating an instance itself in the class. This allows us to pass in stubs to facilitate testing of extractor classes without dependence on PKB.

Uses of design patterns

- Choice of Visitor pattern for `DesignExtractor`. This was done in comparison with keeping logic for design entities and abstractions extraction within `ASTNode` sub-classes.

Basis	Implementing using a Visitor	Implementing in ASTNode classes
SRP	- Algorithm for extraction is separated from ASNode in extractor classes	- Algorithm mixed with normal functionality of ASTNode (to represent program structure), violating SRP
OCP	- Able to support new abstractions by subclassing a new extractor, without need to modify ASTNode.	- Require modification to existing ASTNode classes to support new abstractions, violating OCP
Information hiding	- Can only access public methods of ASTNode, resulting in the need to expose internal structure, violating OOP abstraction	- Able to access all members of ASTNode, reducing the need to expose unnecessary class members.

The main driver to choose the Visitor pattern was its ability to support new design abstractions, allowing us to extend the functionality of the SPA with minimal modifications to existing components.

Optimisations and design in Milestone 3

After performing stress tests on our system, we determined that the `DesignExtractor` was inefficient in extracting programs with deep procedure call stacks. This was due to the recursive nature of some design abstractions (e.g. Uses: All variables used in a procedure are also used in procedures calling it).

This resulted in the following changes to `CallsExtractor`, `UsesExtractor` and `ModifiesExtractor`

Before	After
Evaluate a procedure on every call statement, keeping a procedure call stack for nested procedure calls.	Maintain internal cache during extraction. When reaching a call statement, only extract the procedure if cached results don't already exist.

2.2. PKB

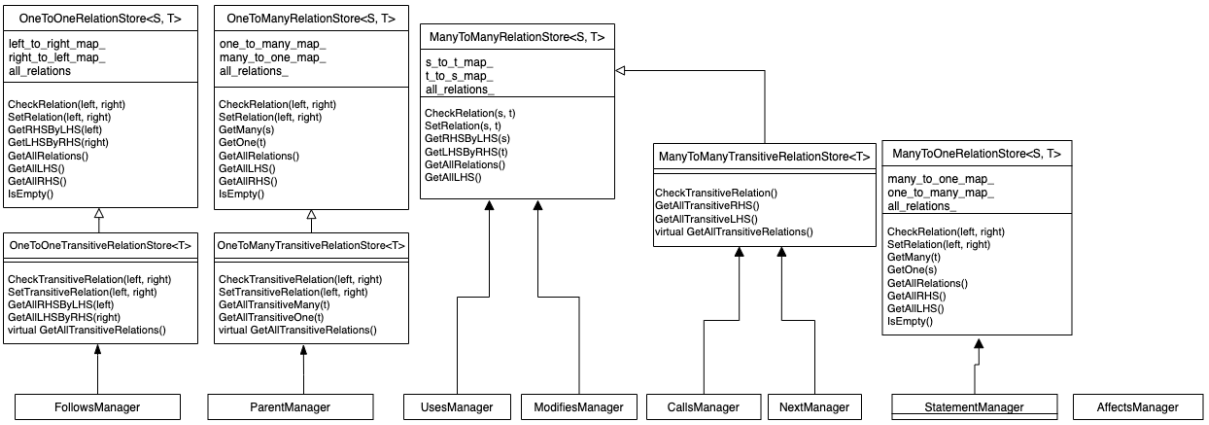
2.2.1. Diagrams

Overview of PKB architecture

Link to PKB class diagram:
https://drive.google.com/file/d/17TTu3XaYi6uFCZ9dzHNJY_YESYYGs_F1/view?usp=sharing

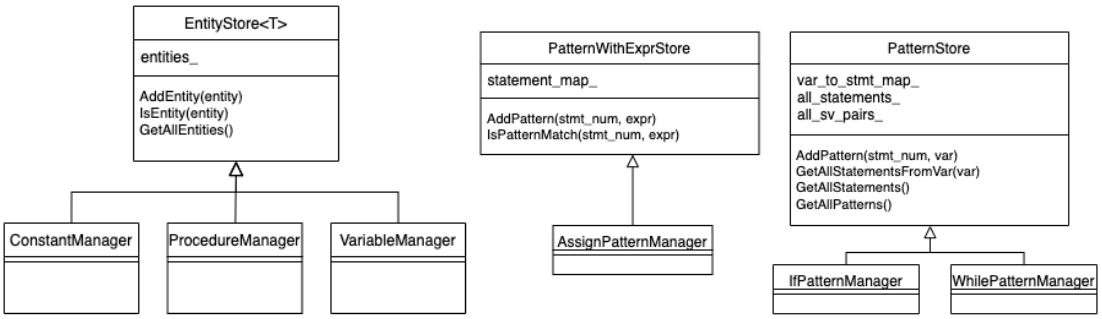
PKB class is instantiated and referenced to by `WritePKBManager` and `ReadPKBManager` objects. This allows for read/write of the PKB to be done through APIs in `WritePKBManager` and `ReadPKBManager`. PKB contains respective design relations and entities supported by SPA.

PKB Relation Managers class diagram



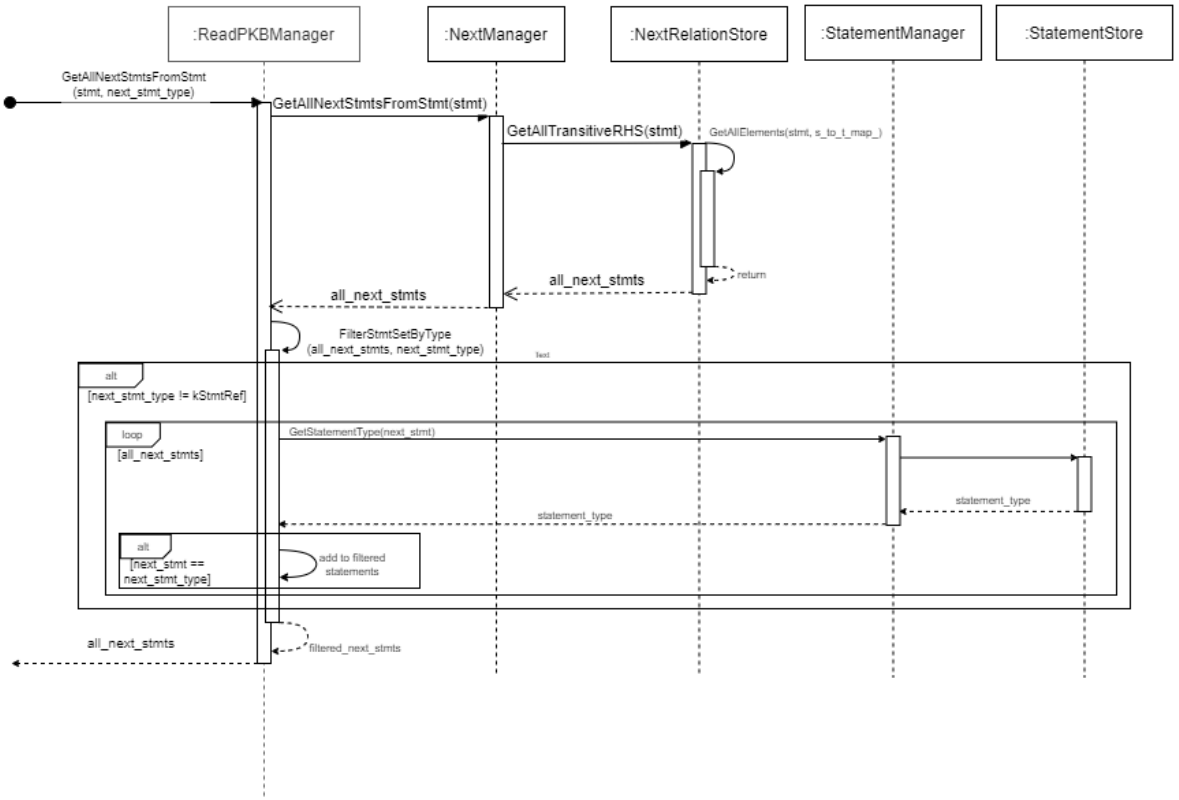
- Design relations can be represented through one-to-one, one-to-many, many-to-one and many-to-many relations
- These relation store classes can easily be extended to support transitive relations as seen in `OneToOneTransitiveRelationStore`, `OneToManyTransitiveRelationStore` and `ManyToManyTransitiveRelationStore`
- Generic classes are used to support relationships of any data types for extensibility
- Relationship managers are created that contain various relationship stores and support methods to read and write to that relation that can be called through the `ReadPKBManager` and `WritePKBManager` APIs

PKB Entity Managers class diagram



- `PatternWithExprStore` and `PatternStore` classes are created for various pattern managers to inherit
- A generic `EntityStore` is created for `ConstantManager`, `ProcedureManager` and `VariableManager` to inherit. Generics is applied here for flexibility of inheritance and creation of entity managers

Sequence Diagram to show QPS calling PKB's GetAllNextStmtsFromStmt method through ReadPKBManager



- QPS `DataRetriever` is able to call `GetAllNextStmtsFromStmt()` API from `ReadPKBManager` class and retrieve a set of statements filtered by statement type.
- `ReadPKBManager` handles underlying calls to various sub-components like `NextManager` and `StatementManager`.
- QPS `DataRetriever` queries through a single endpoint without knowing the underlying logic and interactions between sub-components (Facade Pattern).

2.2.2. Design Decisions

Application of design principles

- Single Responsibility Principle: Every Manager class is created and instantiated to fulfil the storing and retrieving of a single relation or entity
- Don't Repeat Yourself (DRY): Repeatable code is avoided by implementing generic relation/entity store classes that managers can instantiate and call shared methods
- Open-Closed Principle:
    - The classes within PKB are open for extension but closed for modification
    - One example would be `OneToOneRelationStore`, where a subclass can be extended from the parent class to support transitive relation
    - This can be then extended to support other relationships that SPA might incorporate in the future with little to no modification of existing classes
- Interface Segregation:
    - Separate interfaces classes named `ReadPKBManager` and `WritePKBManager` contain strictly read and write methods respectively

    - `ReadPKBManager` and `WritePKBManager` segregates reading and writing to PKB to ensure that QPS and SP will only access PKB components with APIs that are of relevance to them

- Dependency Inversion:
    - Abstract class named `ExprSprec` has an abstract method named `IsMatch()`, containing matching logic (wildcard, partial, and exact expression match)
    - PKB does not have to depend on low-level modules and would just rely on polymorphism when taking in an `ExprSpec` instance in the parameter to be evaluated during runtime

Uses of design patterns

- Facade Pattern:
    - `ReadPKBmanager` and `WritePKBManager` classes provide external facing API and a single entry point for SP and QPS to be able to interact with the PKB component with ease without having to instantiate and directly call methods within PKB subcomponents
    - Reduces coupling and developers working on other components would be able to interact and utilise PKB APIs without knowledge on its underlying components

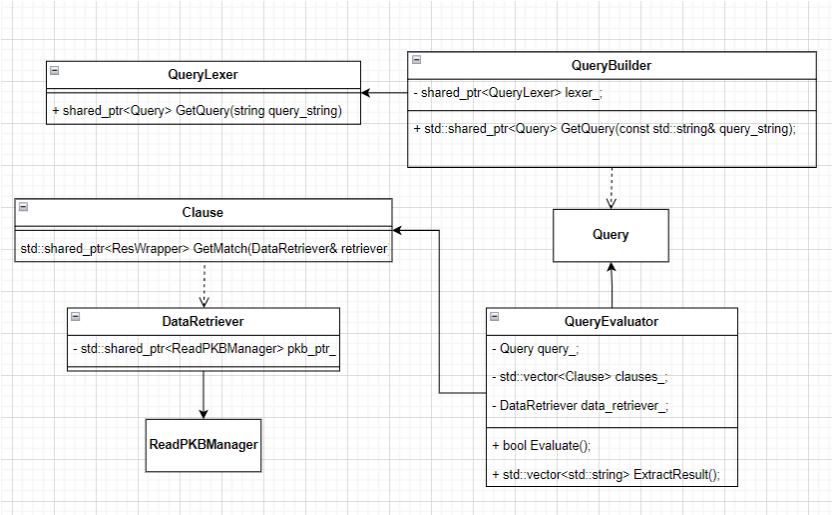
Optimisations and design in Milestone 3

Optimisation	Justification
Creating new PKB APIs to check for existence of queries with one concrete design entity and one wildcard as arguments (e.g. Next(1, _)). API now returns a boolean instead of returning a set of relations and checking if the set is empty in QPS.	Improvement in memory as unnecessary data is not stored and passed to QPS `DataRetriever`. Improvement in runtime as search can early-stop once a relation has been found in PKB.
To get all cause statements in Affects relations, we previously iterated through all Effect statements and added all Cause statements to a set. To optimize, we iterate through all assign statements as possible Cause statement candidates to see if there exists any Effect statement instead.	Finding Cause statements from Effect statements requires a reverse BFS on the CFG for every variable used in the Effect statement. However, when searching from Cause statements instead, we are guaranteed only one variable modified so only one BFS is needed per statement. This improves runtime, especially on source code with many variables used in assign statements.
To find all Affects* pairs (i.e. Affects*(s1, s2)), we needed to repeat the computation of finding RHS of a direct Affects pair (i.e. s2). To reduce the time complexity of this computation, we decided to use a cache for this computation.	Since an assign statement can use many variables, it is highly likely for a statement to be the RHS of multiple other statements. This means that the computation to find subsequent RHS for the statement will be repeated. A cache allows us to prevent this recomputation and improves speed.

2.3. QPS

2.3.1. Diagrams

Overview of QPS sub-components:



`QueryLexer` and `QueryBuilder` collaboratively convert the query string to a `Query` object. `QueryEvaluator` gets the `Query` object and evaluates the clauses of the query by getting data from PKB with the help of `DataRetriever`, and combines the results to produce outputs.

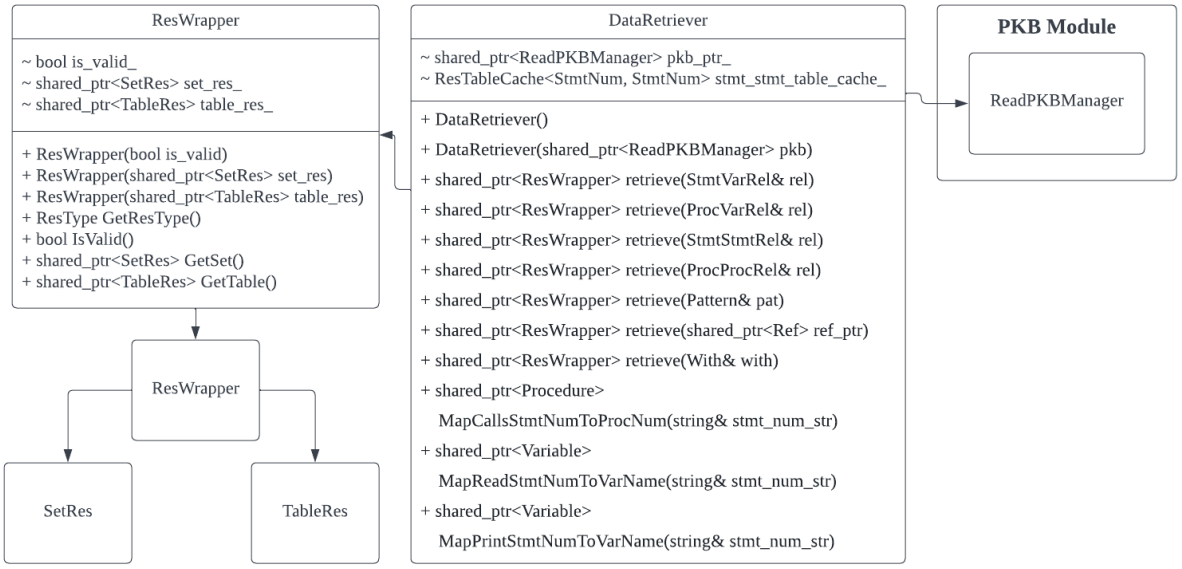
Class diagrams for QPS
Link to reference modeling classes diagram:
https://drive.google.com/file/d/15PO84fDGdJV-Bqcxx7soZJ8yx4RTu_n1/view?usp=share_link

The references in the query are modeled by a hierarchy of classes, each reference holds a value, value type, and attribute type.

Link to clause modeling classes diagram:
https://drive.google.com/file/d/1EHu94EBHm8Y5tPxhuf5yIv9PuPMYgsDo/view?usp=share_link

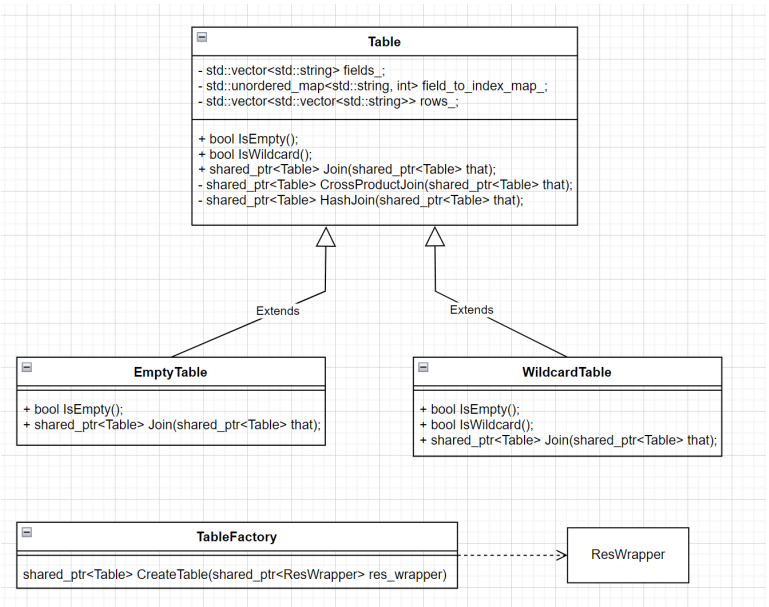
The constraint clauses (such-that, pattern, with clauses) in the query are modeled by a hierarchy of `Clause` classes, each holding a pair of references.

DataRetriever and its relevant classes



`DataRetriever` directly interacts with PKB and retrieves matching data of clauses from PKB.

Table classes



- `Table` classes are used as containers of intermediate evaluation results and implement efficient join algorithms between two tables.
- The `TableFactory` class is responsible for converting `ResWrapper` objects returned by `DataRetriever` to the corresponding `Table` objects.
- `EmptyTable` and `WildcardTable` are two special tables. `EmptyTable` represents an empty result while `WildcardTable` means there are no constraints on the value of the result.
- In the `Join()` function, `CrossProductJoin()` will be called if there are no common fields in the two tables. Otherwise, `HashJoin()` will be called.

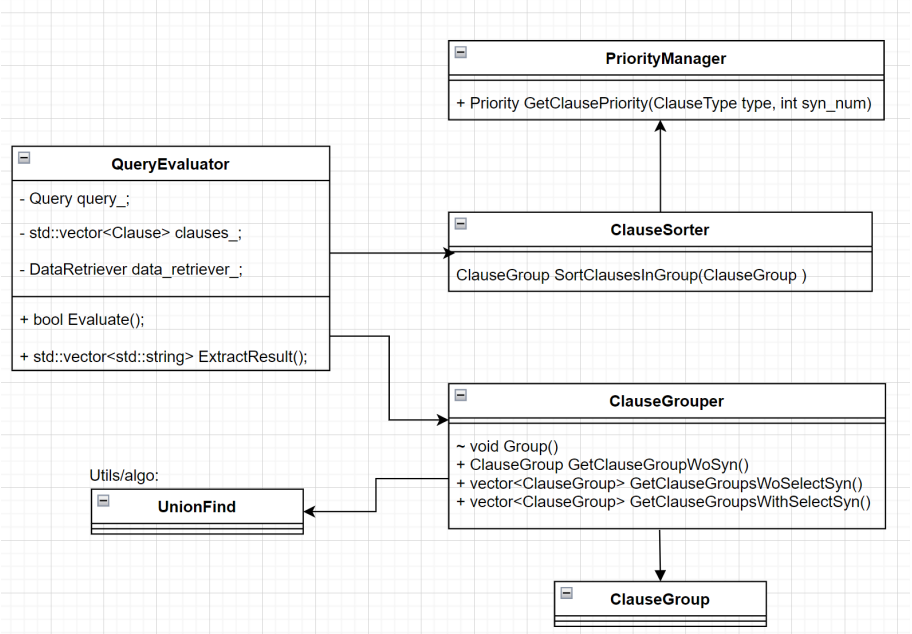
- `HashJoin()` is an efficient function to join tables, the algorithm pseudocode is as follows:

```
procedure HashJoin(this_table, other_table)
    result ← Table()
    map ← unordered_multimap()
    for each row r ∈ this_table do
        key ← ComputeJoinKey(r)
        map.put(key, r)
    end for
    for each row r′ ∈ other_table do
        key ← ComputeJoinKey(r′)
        match_rows ← map.get(key)
        for each row r ∈ match_rows do
            combined_row ← Combine(r, r′)
            result.add(combined_row)
        end for
    end for
    return result
end procedure
```

Changes made in Milestone 3

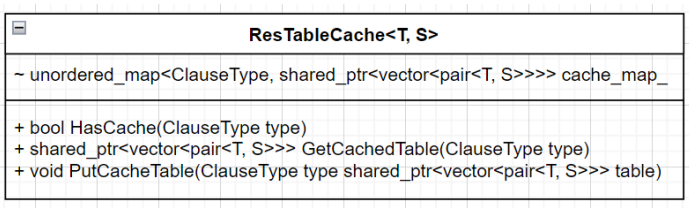
Before	**After**
Many classes were unorganized and put directly under the QPS folder. Besides, the stub classes were put together with the source code.	We reorganized the source code to make the structure of QPS clean and neat. Besides, stub classes were moved to the UnitTesting project.
Classes and methods related to attribute-selecting were partially implemented.	We implemented attribute-selecting features by adding attribute and default attribute information to `Ref` classes, and added attribute mapping APIs in `DataRetriever`.
Classes and methods related to join optimization were partially implemented.	We implemented classes and algorithms for join optimization: `PriorityManager`, `ClauseGroup`, `ClauseGrouper`, and `ClauseSorter`. New methods in `Ref` and `Clause` classes were added to support interaction with these classes.
The checking of wildcard clauses was done by getting a set of all valid results and then checking for set emptiness in `DataRetriever`.	We optimized the data retrieval of wildcard clauses by moving the existence checking to PKB which can early-stop the procedure once a valid object is found.

Optimization-related classes



In Milestone 3, the classes `ClauseGroup`, `UnionFind`, `ClauseGrouper`, `ClauseSorter` and `PriorityManager` are added to optimize the evaluation process. `ClauseGrouper` groups together connected clauses using the `UnionFind` data structure and classifies the group into three categories: group without synonyms, group with only synonyms unconnected to selected synonyms, and group with synonyms connected to selected synonyms.

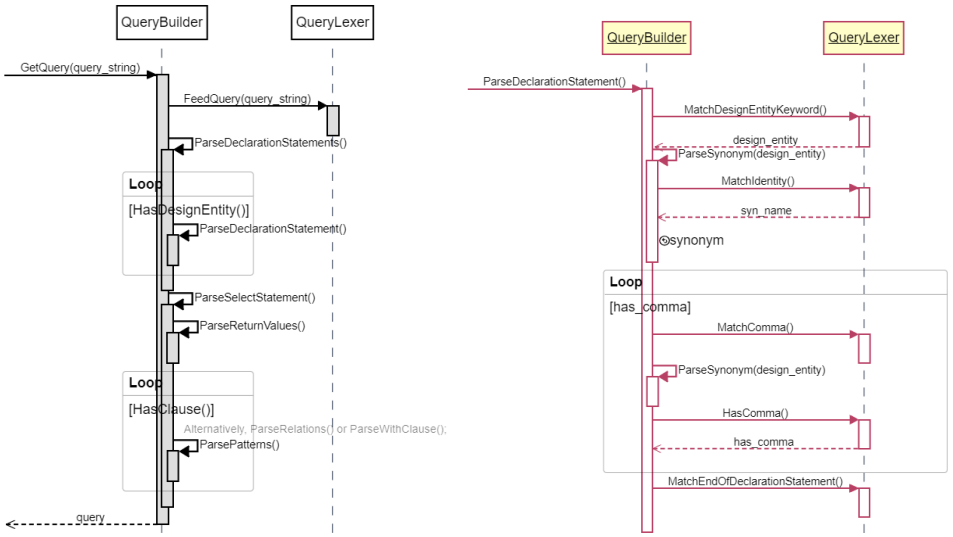
`QueryEvaluator` also makes use of `ClauseSorter` to sort the order of clauses to evaluate within a clause group. Each clause with synonym(s), has its priority according to the clause type and number of synonyms in the clause. All the priority information is managed by `PriorityManager`. Besides the use of clause priority, `ClauseSorter` also ensures the clause order does not produce cross-product join within a clause group.



Besides optimizing the table joins, we also added a `ResTableCache` class to cache the retrieved results of 2 synonym clauses. We find the data retrieval of clauses like Next, Next*, Affects, Affects* with 2 synonyms are especially slow due to the large number of results and the complex retrieval logic in PKB. To avoid repeating the expensive operations within the query, we cache the retrieved data and reuse them for similar clauses. This trade-off larger memory working set with evaluation speed for complex queries.

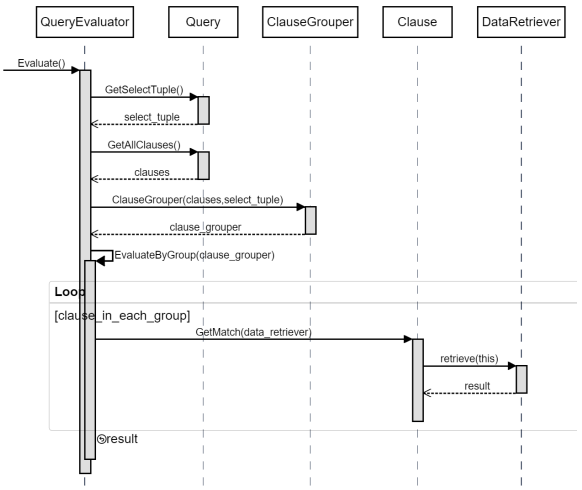
Interaction between QueryBuilder and QueryLexer to parse a query

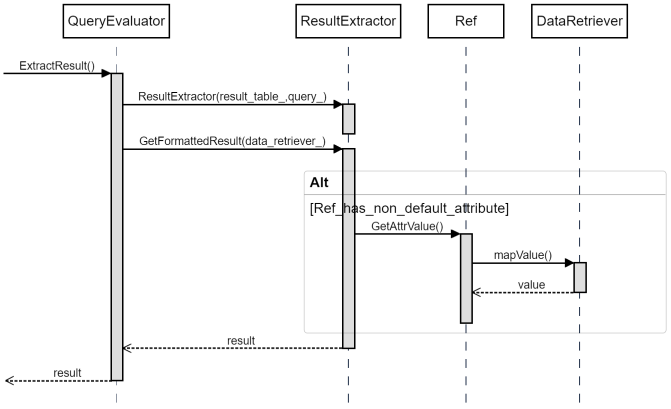
Below is a high-level overview and an example of parsing declaration statement:



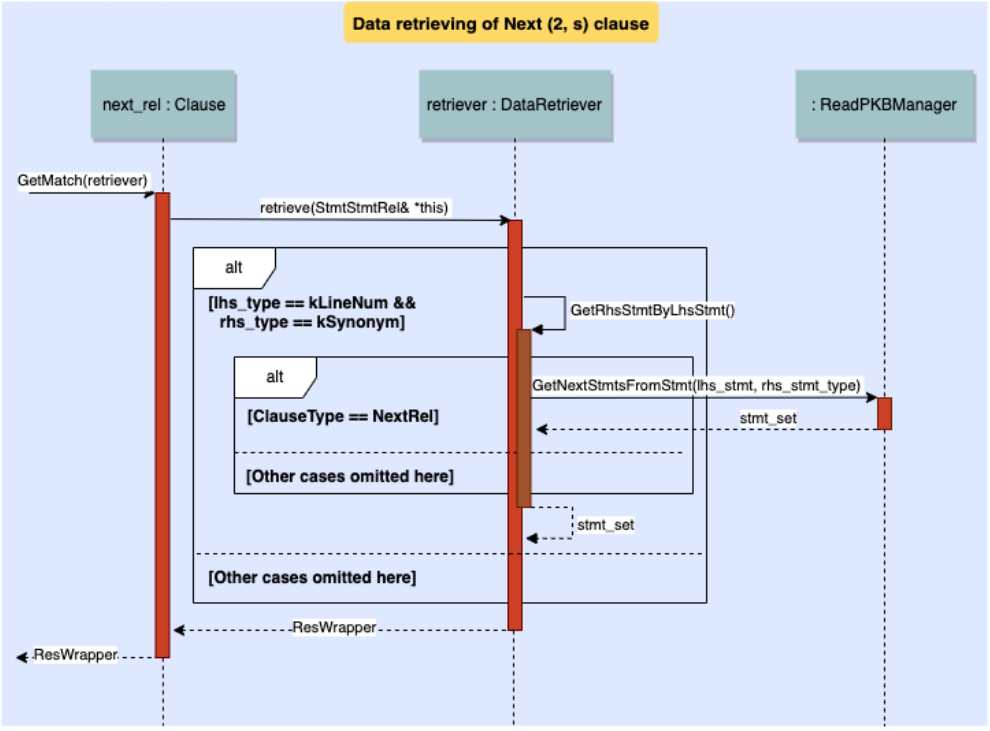
- `QueryBuilder` first feeds the query string to `QueryLexer`, where the query string will be processed into tokens and consumed by calling APIs such as `MatchComma()` and `MatchDesignEntityKeyword()`.
- Subsequently, `QueryBuilder` parses declaration statements and select statements. Within `ParseSelectStatement()`, QueryBuilder parses return values (select tuple) and the clauses iteratively.

Sequence diagram for query evaluation





Sequence diagram for data retrieving of **Next(2, s)** clause.



When the `QueryEvaluator` calls `GetMatch()` at a clause, the `Clause` object accepts a `DataRetriever` object and calls the corresponding overloaded `retrieve()` method in the `DataRetriever` by dynamic type resolution. Then the data retriever will look up the combination of clause arguments' value types to decide which helper function to call; and then inside the helper function, the data retriever will further look up the specific clause type of the clause, and call the corresponding PKB API to get matching data. Finally, it wraps the matching data and the synonym name in a `ResWrapper` and returns it to `QueryEvaluator`.

2.3.2. Design Decisions

Application of design principles

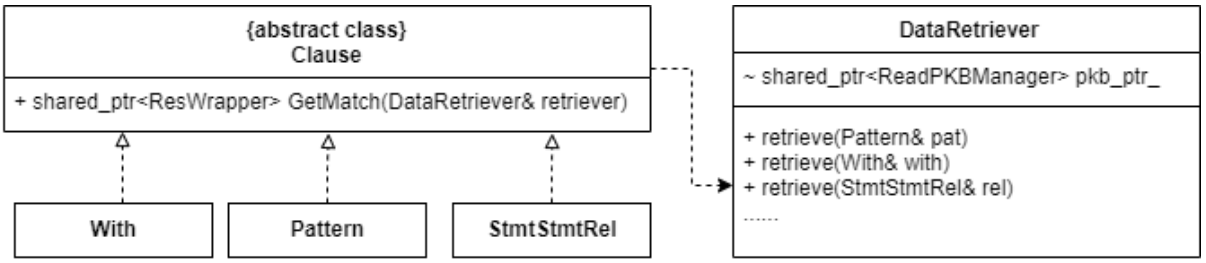
- Open Close Principle:
  - Every derived `Clause` class in QPS is responsible for one type of clause (With, Pattern or Relation). A new type of clause can be easily added as a subclass of `Clause` without affecting existing `Clause` classes.
  - Similarly, a new type of reference can be easily added as a subclass of `Ref` without affecting existing `Ref` classes.
  - The `QueryEvaluator` views all clauses as variables of type `Clause` which has abstract method `GetMatch(DataRetriever&)`, and relies on the Visitor pattern of `DataRetriever` to retrieve matching data of the clause; Thus when a new clause sub-class is added, `QueryEvaluator` doesn't need to make any changes.
- Single Responsibility Principle:
  - The query evaluation is broken into multiple phases and handled by different classes: `DataRetriever` is responsible for retrieval of matching data from

PKB and `QueryEvaluator` is responsible for combining results of clauses and `ResultExtractor` is responsible for final result extraction and formatting.
    - In query evaluation, part of the optimization is done by grouping related clauses and sorting the clause within each clause group. These two tasks are handled by `ClauseGrouper` and `ClauseSorter` separately to reduce coupling.
- Don't Repeat Yourself (DRY) Principle:
    - Similar logics are abstracted out as functions. For instance, in the `QueryBuilder`, UsesS and ModifiesS relations have the same types of left hand side and right hand side reference. Thus, the same function is shared when parsing the reference of these 2 relations. In the `DataRetriever` class, relations with the same argument value types share some processing logic (e.g. `UsesSRel` and `ModifiesSRel` are assigned to `retrieve(StmtVarRel&)` for processing).
    - Classes with similar logic are shared across different components. The QPS shares a tokenizer with SP, and shares the arithmetic expression-related class including `Expr` and `ExprSpec` with PKB and SP.

<u>Design decision on trade-offs</u>

- Use enum classes in QPS for type checking:
    - We use type enum classes and if-else statements in `DataRetriever` to determine the required type information of the query to correctly call the corresponding PKB API.
    - This breaks the Open-Close-Principle as when new clauses are added we need to add more cases to the if-else statements.
    - An alternative way to achieve the same purpose is to create a large number of classes each responsible for a clause type and a combination of value types, and then use inheritance and polymorphism to resolve to the corresponding PKB API.
    - The alternative way adds unnecessary complexity to QPS and also makes the code base filled by a large number of simple classes. As a trade-off, we decide to use the enum classes and if-else statements since we have regression tests to prevent breaking implemented features.
    - Furthermore, enum types like RefType and ClauseType are used as keys in maps which are simple and effective; In contrast, using self-defined classes as keys requires the implementation of hash functions, which is less efficient and error-prone.

<u>Uses of design patterns</u>



Note: Other sub-classes and retrieve() methods omitted for simplicity.

Visitor pattern:

- The `Clause` abstract class is the `Visitable`, it defines an abstract method to accept a `DataRetriever` and returns matching data of the clause. This helps to separate the responsibility of reading data from PKB to a separate class
- The `DataRetriver` is the `Visitor`, it has a set of overloaded `retrieve()` methods, each handles the retrieving for a specific `Clause` sub-type.
- The `Clause` class will call the retrieve method from `DataRetriever`. By passing in its own reference, the dynamic type resolution will call the correct method in `DataRetriever`.

Factory Method Pattern:

- Factory Method Pattern is used when `QueryParser` parses the synonym declaration to produce a `Ref` object
- The design pattern is also used to convert a `ResWrapper` class returned by PKB to a corresponding Table object.

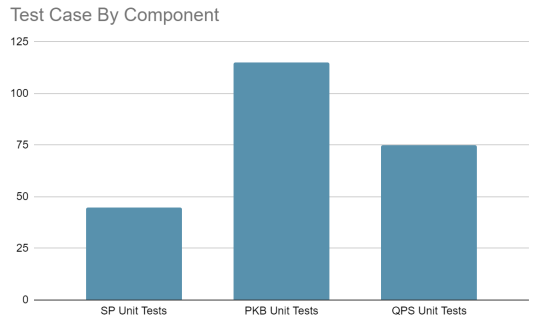
3. Testing

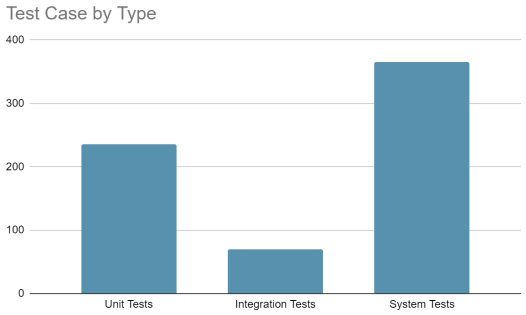
The objective was to extensively test all functional and non-functional requirements within SPA. For functional requirements, we adopted a bottom-up testing strategy which ensured that all unit and integration tests were cleared before focusing on System testing and eventually testing of non-functional requirements. Non-functional requirements included Parsing Timings, Query Timings and Memory Usage.

3.1. Automating Testing

- Power shell scripts to automate execution of system test suites.
- VS Test Explorer to automate unit/integration tests.
- GitHub Actions to ensure all test cases (unit tests, integration tests, and system tests) are cleared before a PR can be merged - also served as a form of regression testing by ensuring new features did not break existing code.

3.2. Testing Statistics





A total of 700 test cases were written - 400 system tests, 230 unit tests and 70 integration tests.

3.3. Design Strategies

Pairwise Testing (For generation of all test cases)

	PARAMETERS FOR TESTING
1	TYPES OF CLAUSES (WITH, SUCH THAT, PATTERN)
2	RESULT TYPE (SINGLE, MULTIPLE, BOOLEAN)
4	TYPES OF SYNONYMS
5	NUMBER OF CLAUSES (STRESS TESTING)
6	TYPE OF SUCH THAT CLAUSE
7	TYPE OF WITH CLAUSE
8	TYPE OF PATTERN CLAUSE

Pairwise testing was used to create the test cases to ensure the largest possible discrete combination of types of test inputs was covered. This ensured that there weren't too many test cases for the same item making testing efficient and systematic. The diagram on the left shows the most common parameters used to pairwise test but not all. There are other inputs tested as well from the semantic/syntactic validity of the PQL to the level of complexity of the SPA program.

White Box Testing (Used in Unit/Integration Testing)

In integration testing, different components/sub-components of SPA such as QPS and PKB-SPA were tested together. The creation of stubs was necessary in the design of functional test cases which required knowledge of the specific component and its interaction with other components. Not only did this help us find bugs but it also helped us narrow down the scope and causes of other bugs.

Black Box Testing (Used in System Testing)

Black Box testing was used to generate test cases that evaluate end-to-end system requirements. Test cases were generated using the cs3203 GitHub without reference to the internal code. Black Box testing proved to be the most effective at uncovering bugs.

Equivalence Partitioning

We divided the input domain into classes using Equivalence Partitioning to help us generate test cases. For instance, to test combinations of clauses some of the possible classes include:

- Select __ such that __ pattern __ and __ with __
- Select __ such that __ and __ pattern __ with __
- Select __ pattern __ such that __ with __ and __
- Select __ pattern __ such that __ such that __ with __
- Select __ such that __ with __ such that __ pattern __
- Select __ pattern __ such that __ and __ with __
- Select __ pattern __ such that __ and __ and __

3.4. Types of Tests

Stress Tests

To test the stability of the SPA, a test suite under system testing was created with valid SIMPLE source programs of up to 2000 lines per program. The goal was to monitor the parsing time and memory usage in PKB and identify the bottleneck of the system. Queries with a large number of clauses were also created to check whether the evaluation of queries was within the time limit.

Boundary Value Analysis

Using BVA we tested valid and invalid boundary values within each partition. For instance, in the evaluation of each Next* clause, a single procedure constituted a partition and there would be test cases to ensure the Next* evaluated to true between the first and last last statement of the same procedure and false for the statement after the last.

Negative Test Cases

Negative test cases were used to check whether semantically and syntactically incorrect PQL queries were gracefully handled by SPA. Negative test cases were also written to correctly identify invalid SIMPLE programs.

3.5. Bug Handling

We used GitHub Issues to track bugs and assign them to developers

4. Reflection

4.1. Problems encountered

1. Project integration: In the early stages of our project, we worked on each major component (SP, PKB, QPS) in isolation. Integration was only done close to our first demo, which resulted in multiple issues surfacing (e.g. passing uninitialized pointer from PKB to QPS). This required us to spend significant amounts of time debugging all such issues.
2. Testing: Before Milestone 1, we did not pay much attention to testing when implementing features of SPA. System testing was also done only a day before milestone submission due to problems with integration. This negatively affected our correctness in Milestone 1 and many uncaught bugs were hidden in our code.

4.2. Good practices maintained

1. Project management: It's important to keep a consistently good working style within the team. We follow the Scrum cycle tightly and have standups twice a week (1 internal standup for group discussion and 1 with the tutor to report team progress), and have reflections and summaries at the end of each sprint. These help each member to be well-informed about the project progress and problems faced, and keep us active to solve the problems and develop new features. Frequent check-ins with the team also allowed us to distribute manpower across sub-components more efficiently as the workload for each sub-component varies over time. This allowed us to be more agile and develop at a faster pace.
2. Communication between sub-components: We spent a significant amount of time designing the PKB-SP and PKB-QPS interactions and APIs needed. This paid off well as API linkage went smoothly without major conflicts.
3. Project integration: We learned from the lessons in Milestone 1 and integrated different components as early as possible after Milestone 1. The early integration exposed the bugs quickly and left enough time for bug-fixing and improvement of coding styles.
4. Testing: after Milestone 1, we added extensive unit tests, integration tests and system tests immediately after the development, and tried to cover various complex corner cases. This ensured our correctness in Milestone 2 and Milestone 3.

4.3. Release retrospection

The SPA project allowed us to design and implement a non-trivial system. Due to the well-specified requirements, we were able to apply various design patterns learnt from the module, producing a product with high code quality.

There was also complexity in the project that needed us to implement efficient algorithms, thinking about the time complexity and memory usage of our codebase.

Despite issues faced in Milestone 1 such as integration failures and low testing effort, we were able to make up for them in subsequent sprints by early use of continuous integration, and also allocating team members to focus on system testing.

We were hence able to produce software that fulfills the requirement, and more importantly, one that is maintainable.

5. Appendix

5.1 API Listing

5.1.1 List of APIs for SP

`SourceValidator`:

```
bool Validate()

Returns true if the list of tokens represents a semantically and
syntactically valid SIMPLE program, and false otherwise.
```

```
void SetTokens(shared_ptr<vector<SourceToken>>)
```

Sets vector of tokens to be used for parsing.

`SourceParser`:

shared_ptr<ProgramNode> Parse() Create AST structure representing a SIMPLE program using the list of tokens, and return the root of the Tree (i.e. ProgramNode).
void SetTokens(shared_ptr<vector<SourceToken>>) Sets vector of token to be used for parsing.

`DesignExtractor`:

void **PopulatePKB(shared_ptr<ProgramASTNode> root)** Extract all design entities and abstractions from given root node of SIMPLE program, and write them into the data store of PKB.

5.1.2 List of APIs for PKB

`ReadPKBManager` API example:

shared_ptr<unordered_set<StmtNum>> **GetChildrenFromStmt(StmtNum parent, RefType children_type)** Returns all the statement numbers that have a direct child relation to a given parent statement number and filters by child statement type.

`WritePKBManager` API example:

void **SetParent(StmtNum parent, StmtNum child)** Stores a Parent relationship containing the parent and direct child statement number.

5.1.3 List of APIs for QPS

`QueryBuilder`:

shared_ptr<Query> **GetQuery(string query_string)** Tokenize and parse the query string and return a Query object.

`QueryEvaluator`:

bool Evaluate**(Query query)** Evaluate the query by querying PKB. Return true if the result is not empty, otherwise false.
vector<string> **ExtractResult()** Extract the result of the query in the form of a vector of string.

5.2 extension proposal

5.2.1. Definition of the extension

Syntax of try-catch and throw statements

```
stmtList: stmt+
stmt: try | read | print | call | while | if | assign
throwStmt: throw
stmtListWithThrow: {stmt | throwStmt}+
try: 'try' '{' stmtListWithThrow '}' 'catch' '{' stmtList '}'
throw: 'throw' exception ';'
exception: NAME
```

Scope: Since an Exception entity would be defined as a string and not a class, we are unable to utilise polymorphism to catch various subtypes of exceptions. Hence, in this extension proposal, we will be limiting the scope to strictly one Exception being thrown and caught in the try-catch blocks respectively, where the exception thrown and caught has to be matching. Any other implementation violations found in the SIMPLE source code would be then detected in SP and denoted as syntax error in the code.

PQL Grammar

Syntax for PQL

```
New design-entity: Exception
New relRef: Throws | Catches
ThrowsS: 'Throws'  '('  stmtRef  ','  entRef   ')'
ThrowsP: 'Throws'  '('  entRef ','  entRef   ')'
CatchesS : 'Catches' '(' stmtRef  ','  entRef   ')'
CatchesP : 'Catches' '(' entRef  ','  entRef   ')'
```

A procedure or a statement can throw an exception and catch an exception.

5.2.2. Changes required to your existing system and its components

SP: Modify algorithm to set edges for CFG due to new control flow of try-catch blocks
PKB: ThrowsManager and CatchManager can be implemented with existing abstract stores created. ExceptionManager can be implemented by extending the current EntityStore class. StatementManager will have minimal modifications to store throw and catch statement numbers.
QPS: Modification needed includes adding new Clause and Ref types

5.2.3. Implementation details

SP
SourceLexer: Add keywords checking for “try”, “catch” and “throw” tokens
SourceValidator: Add syntax and semantic error checking for throw-catch and throw statements adhering to the rule defined above
SourceParser: Add *ThrowCatchStatementASTNode* and *ThrowStatementASTNode*, which subclasses *StatementASTNode*

DesignExtractor: Add new virtual methods of *ExtractThrowCatchNode* and *ExtractThrowNode* and have the concrete node extractor classes implement the new methods.

PKB

Throw/Catch Relation: A throws and catch procedure-exception relationship can be classified as a many-to-many relation. A throw and catch statement-exception relationship can be classified as a many-to-one relation. Since a ManyToManyRelationStore and OneToManyRelationStore has already been defined as an abstract generic class in PKB, the current codebase could easily extend by creating a ThrowsManager and CatchesManager within PKB component that would contain a ManyToManyRelationStore<Procedure, Exception> and ManyToOneRelationStore<StatementNum, Exception> each. This will allow us to leverage on existing logic found in the abstract store classes to support the implementation of new APIs for QPS to utilize.

Throw/Catch Statement: StatementManager can be extended to support read/write of throw and catch statements. This could then be used to support the filtering by throw and catch statement type that would be compatible with other relation managers.

Exception Entity: Current existing implementation can also be easily extended to implement the new Exception entity, where an ExceptionManager class can extend the EntityStore generic class with Exception as its generic type.

QPS

QueryLexer: add the relation keywords “Throws” and “Catches”, and entity keyword ‘Exception’
QueryParser: add the logic to parse Throws and Catches relations
Ref: add a new ExceptionRef class
Clause: add the new relation class Throws and Catches.
DataRetriever: add the functions to extract Throws and Catches results from PKB.

5.2.4. Possible challenges to implementation and testing, and mitigation plans

Unit Testing:

Try-catch statements result in a change in the control flow structure. There is thus a need to test the construct with other statements that affect CFG, such as if and while statements. Test cases need to be put in place to test such combinations thoroughly, using test methods such as pair-wise testing

System Testing:

Like if-else and while statement, try-catch statement is a container statement, so it can be involved in many different kinds of abstraction relationship (e.g. Next, Follows, Parent, Uses etc.). This requires us to add more system testing queries on try-catch code clips and on different abstractions and combined abstractions.

5.2.5. Benefits to SPA

Most modern programming languages support some form of error handling and one implementation of it would be in the form of try-catch blocks. Hence, it would be beneficial for developers using the SIMPLE language to implement try-catch blocks in their code, and for them to query the type of exceptions that can be caught.