CS3203 Software Engineering Project

AY22/23 Semester 1

Project Final Report – System Overview

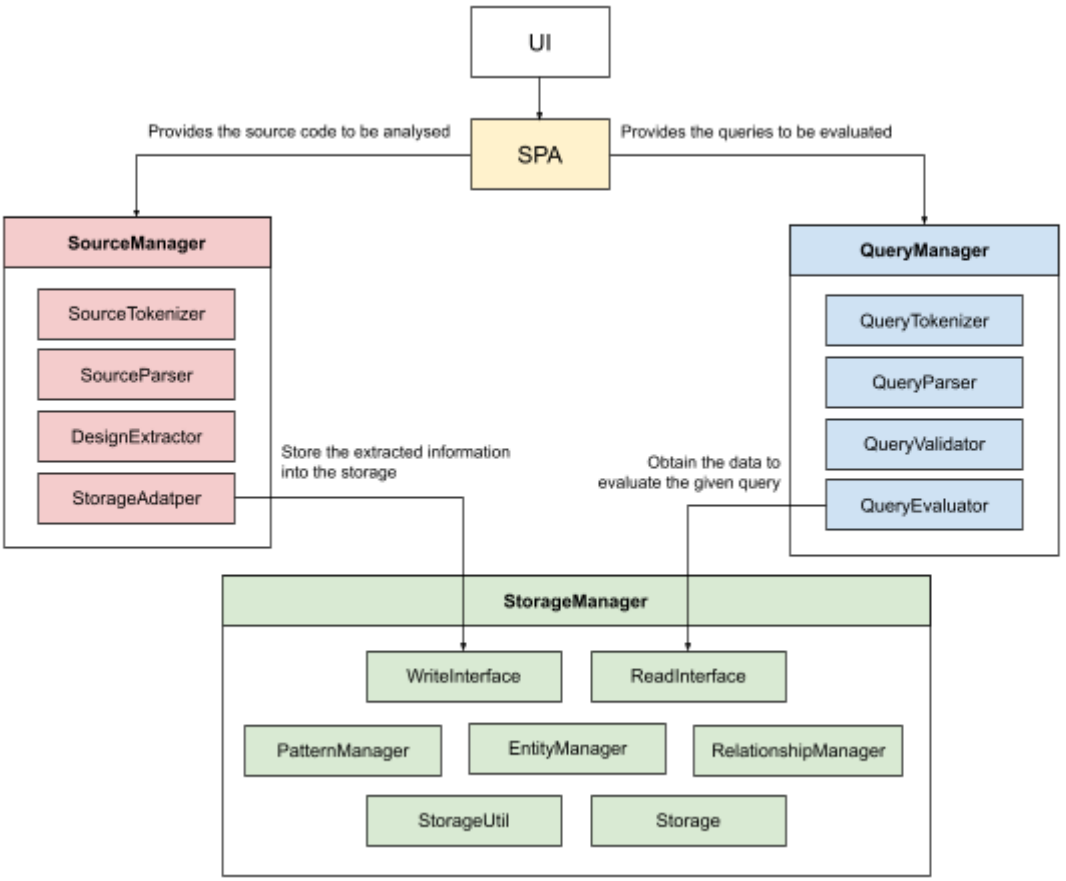
Team 26

Table of Contents

Glossary of Terms

Term	Description
SPA	Static Program Analyzer
SP	Source Processor
PKB	Program Knowledge Base
QPS	Query Processing Subsystem
API	Application Programming Interface
CI	Continuous Integration
AST	Abstract Syntax Tree
PR	Pull Requests

1. System Architecture



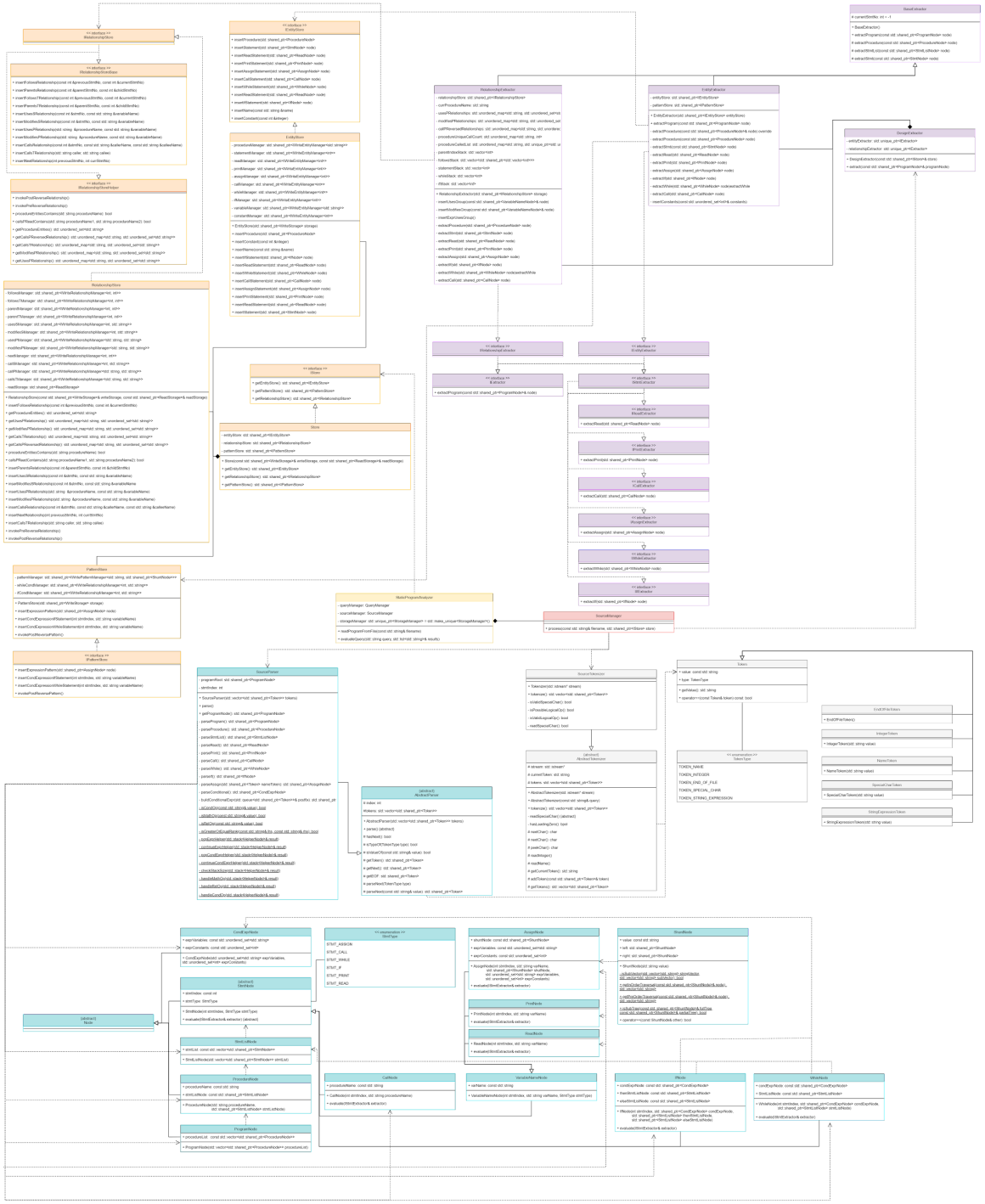
2. Component Architecture and Design

2.1 Source Processor

2.1.1 System Design

The SourceManager is the entry point of the SP containing four individual components listed below.

2.1.1.1 Class Diagram[1]



[1] Dependencies between nodes section and extractors omitted to not over complicate the diagram. | SP Class.svg

Class Diagram Explanation

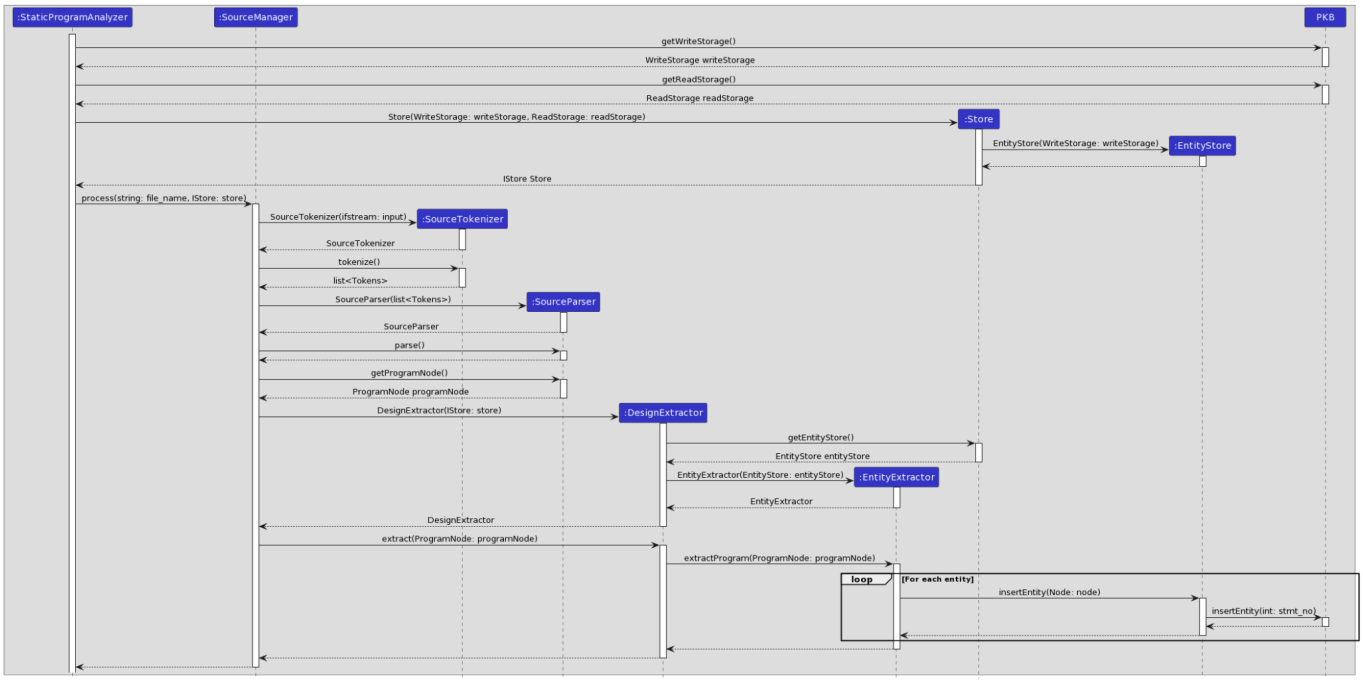
SourceTokenizer. The SourceTokenizer reads the SIMPLE programme base on the given input file path input and tokenizes the input stream into Tokens. The SourceTokenizer inherits from the AbstractTokenizer.

SourceParser. The SourceParser accepts the tokens from the SourceTokenizer and parses them into an AST. The AST's node classes are built with inheritance where each statement class (VariableNameNode (Read, Print), While, Assign, Call, If) inherits from a common StmtNode. The SourceParser inherits from the AbstractParser.

DesignExtractor. The DesignExtractor extracts information from the AST using the different implementations (Entity, Relationship) of the extractor to extract the respective information. For each extracted information the data is passed to the StorageAdapter to be saved into the data store. Each extractor inherits from the BaseExtractor and implements an extractor interface.

StorageAdapter[2] The StorageAdapter acts as an adapter for writing information extractor into the data store (the PKB in this implementation). The StorageAdapter also helps to encapsulate access to the methods in the storage interface to ensure that the respective extractors can only access the corresponding writers.

2.1.1.2 Sequence Diagram[3]



Sequence Diagram Explanation

The SPA starts by obtaining the ReadStorage and WriteStorage from the PKB. These are then passed to the SP's Store implementation to create the StorageAdapter for the SP to read and write. The SourceManager is then initialized by the SPA, in turn the SourceManager will tokenize and parse the input into an AST in which the DesignExtractor will extract the respective detail from and write to the PKB using the StorageAdapter.

---

[2] StorageAdapter is named Store in diagrams as per Class.

[3] Only the EntityExtractor is shown in this diagram. Detailed extraction method calls are omitted. |

SP Sequence Diagram.svg

2.1.2 Design Decisions[4]

Components	Complexity	Ease of Implementation	Extensibility and Maintainability
Shared Tokenizer	**Space**: O(n) **Time**: O(n) n: number of characters in input	Both the inputs for the SP and QPS are text based files, having a Common Tokenizer and Parser helps us to speed up development by only having one implementation.	Since both the parser and tokenizer are abstract classes, we can easily add additional implementations if required. The key implementation of the tokenizing and parsing is in the concrete implementation.
Shared Parser	**Space**: O(n) **Time**: O(n) n: number of tokens		
ShutingYardParser	**Space**: O(n) **Time**: O(n) n: number of tokens	The ShuntingYardParser is a common implementation for the parsing arithmetic expression. It can be implemented using the shunting yard algorithm and be shared.	A common implementation helps us such that we only need to maintain one set of code for parsing expressions. The code can easily cater to additional arithmetic rules if required and it will immediately be added to the SP and QPS due to its shared nature.
DesignExtractor	**Space**: O(k) **Time**: O(nk) n: number of procedures k: number of extractors	The DesignExtractor helps to decouple parsing and extracting, providing clarity during development.	Each extractor concrete implementation can be easily modified independently without affecting other extractors. We can also add additional extractors of existing type or new types if required.
StorageAdapter	**Space**: - **Time**: O(n) n: number of write request from DesignExtractor	The adapter provides a layer of abstraction for the DesignExtractor to consume the PKB. It acts as a layer of decoupling between extraction and storage.	The StorageAdapter is built on interfaces which can easily have additional concrete implementations.

2.1.2.1 Application of Design Principles

Single Responsibility Principle. The process of reading the input, extracting information and storing the data are divided into individual components (SourceTokenizer, SourceParser, DesignExtractor, StorageAdapter).

Open-Closed Principle. By extension of DIP, depending on abstractions rather than concrete implementations, we can extend the code by adding different implementations without modifying existing code.

[4] Some values for space and time complexity are omitted because it is irrelevant.

Interface Segregation Principle. By extension of DIP, we use small and specific interfaces that only expose the methods that are needed by the high-level module. Therefore adhering to ISP by not forcing the high-level module to depend on unnecessary methods.

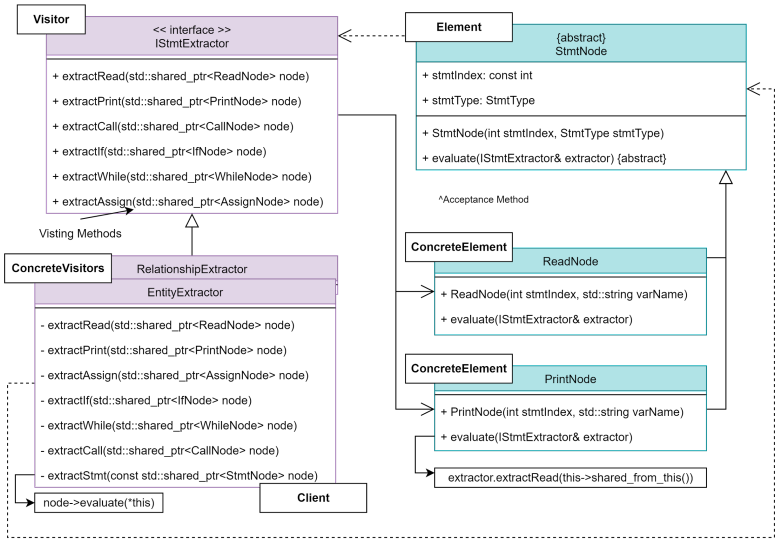
Dependency Inversion Principle. The DesignExtractor and StorageAdapter are built on abstractions rather than concrete implementation. Hence the SourceManager depends on its abstractions (i.e. IEntityExtractor, IRelationshipExtractor) rather than its concrete implementations (i.e. EntityExtractor, RelationshipExtractor).

2.1.2.2 Use of Design Patterns

Component	Possible Patterns	Considerations
DesignExtractor **Selected:** Visitor and Template Method	Template Method	Template Method was selected to build the BaseExtractor for the extraction of common nested nodes such as the ProgramNode, ProcedureNode, StatementNodes, etc.
	Visitor	Visitor was selected to extract the basic direct relationship between design entities from AST as it allows the different concrete implementation of the DesginExtractor to be accepted by the node.
	Strategy	Strategy was considered, however, it will separate the extraction by nodes rather than concrete implementation of extractors. This will make the code inextensible by extractors which was not ideal.
SourceParser, SourceTokenizer **Selected:** Template Method	Template Method	Template method was selected to create a common parser for the SP and QPS as part of the design decisions.
	Strategy	The strategy approach was considered for the Parser, however, did not work well with the Template Method and hence was dropped.

Visitor.[5]

[5] Adapted from https://refactoring.guru/design-patterns/visitor | SP Visitor.svg



The visitor interface (i.e. IStmtExtractor) declares a set of visiting methods (e.g. extractRead) that can take concrete elements of an object structure (e.g. ReadNode) as arguments. Each Concrete Visitor (i.e. EntityExtractor) implements several versions of the same behaviours, tailored for different concrete element classes. The element interface (i.e. StmtNode) declares the evaluate method for “accepting” visitors. Each Concrete Element (i.e. ReadNode) will implement the acceptance method. This is to redirect the call to the visitor’s method (i.e. extractRead) corresponding to the current element class.

Template Method. This design pattern allows us to have a default implementation which can be overridden if required, in addition call the default method in the overridden method if required. It also allows abstract steps to be defined, making it ideal for different implementations of extractors.

Facade. This design pattern allows us to hide the complexities of how the SP manages the tokenizing, parsing, extraction and storing of the input. The only method exposed is in the SourceManager component and only takes in the file and storage required to store the extracted information.

2.1.3 Changes from previous Milestone

Liskov Substitution Principle. The DesignExtractor was implemented such that all Extractors implement only one method, the extractProgram method. Hence, all extractors can be a type of IExtractor interface.

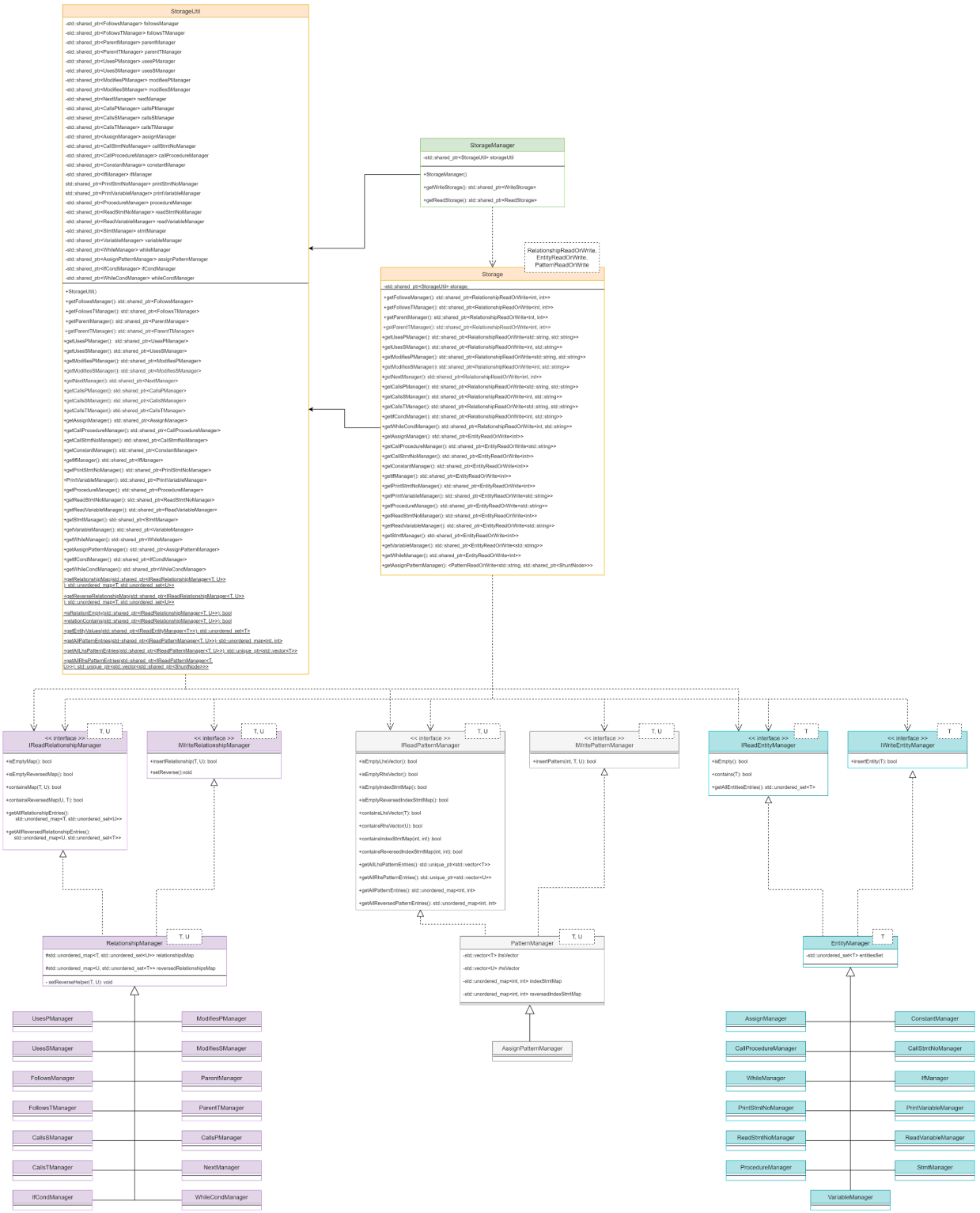
PatternExtractor. PatternExtrator was removed as it was deemed unnecessary since all patterns belong to an entity, so we combined the pattern with the EntityExtractor.

2.2 Program Knowledge Base

2.2.1 System Design

The StorageManager is the entry point of the PKB containing four individual components as listed below.

2.2.1.1 Class Diagram[6]



[6] PKB Class.svg

Class Diagram Explanation

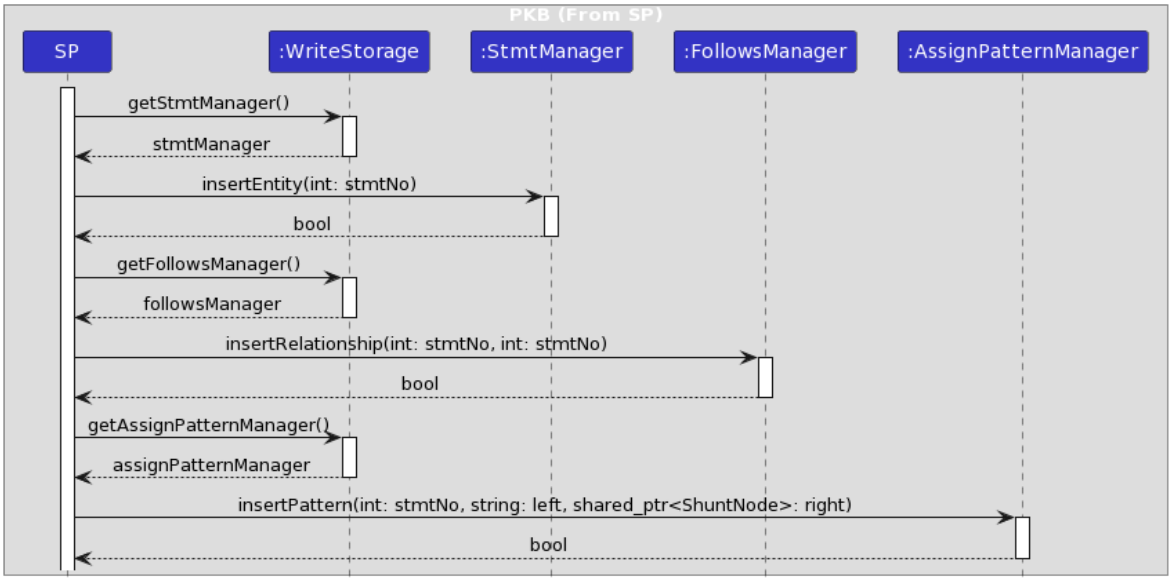
Storage. The Storage allows the SP to obtain a WriteStorage to write data to the PKB while the QPS can obtain a ReadStorage to read data from the PKB.

EntityManager. The EntityManager implements interfaces IWriteEntityManager and IReadEntityManager - write enforces an insert method and read enforces getters for all stored entries. The child entity managers (variable, constant, etc.) all inherit from EntityManager. After processing by the SP, the entity is stored as 1 piece of data: either its name or statement number. The entity is stored in an unordered set.

RelationshipManager. The RelationshipManager implements interfaces IWriteRelationshipManager and IReadRelationshipManager - write enforces an insert method and read enforces a getter for all stored entries. The child relationship managers (follows, uses, etc.) all inherit from RelationshipManager. After SP processing, the relationship is stored as 2 pieces of data: the first parameter and the second parameter. The first map is created using the first parameter as the key and an unordered set as the value, and vice versa for the second map. This supports the mapping of one key to many values. A reversed map is also created and stored.

PatternManager. The PatternManager implements both interfaces IWritePatternManager and IReadPatternManager - write enforces an insert method and read enforces a getter for all stored entries. After SP processing, the assignment statement is stored as 3 pieces of data: its statement number, the data on the left of the = operator and the data on its right. The left and right data are stored in their respective vectors, and the size of each vector is used as the index key to map to its statement number. A reversed map is also created and stored.

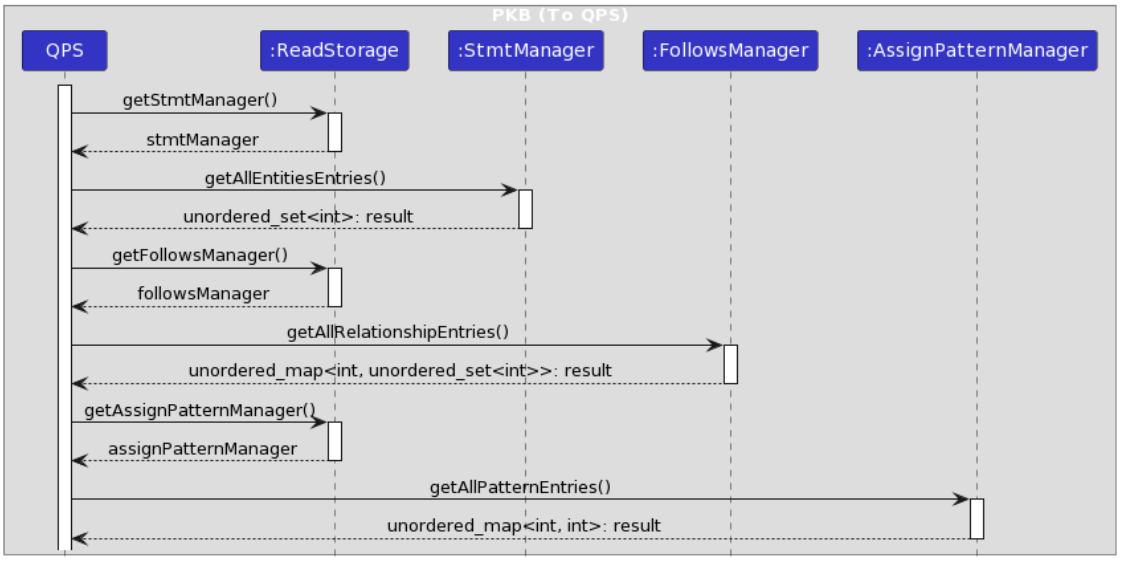
2.2.1.2 Sequence Diagram[7]



Sequence Diagram 1: SP to PKB

Sequence Diagram 1 Explanation

After initialization of the WriteStorage, the SP makes a call to an instance of the WriteStorage to obtain either the StmtManager, FollowsManager or AssignPatternManager for insertion. The manager does the insertion for either entity/relationship/pattern and returns a boolean to indicate the success or failure of insertion.



Sequence Diagram 2: PKB to QPS

Sequence Diagram 2 Explanation

After initialization of the ReadStorage, the QPS makes a call to an instance of the ReadStorage to obtain either the StmtManager, FollowsManager or AssignPatternManager for retrieving. The manager does the retrieval for either entity/relationship/pattern, and returns the respective set or map of required entries.

[7] SP to PKB Sequence Diagram.svg | PKB to QPS Sequence Diagram.svg

2.2.2 Design Decisions

Components	Complexity	Ease of Implementation	Extensibility and Maintainability
Entity Manager	**Space**: O(n) **Time**: O(1) (Average time for accessing data)	Entities are stored as an unordered set of generic type T.	Easily extendable due to the use of generics. With the use of an unordered set, it is easy to ensure that there are no repeated entries.
Relationship Manager	**Space**: O(n) **Time**: O(1) (Average time for accessing data)	Relationships are stored as an unordered map with key: generic type T and value: unordered set of generic type U. This is to support one-to-many mapping. The reverse map is stored with key and value data types reversed.	Easily extendable due to the use of generics. Slightly more difficult to maintain as we map from key T to unordered set U, and store the reversed mapping as well.
Pattern Manager	**Space**: O(n) **Time**: O(1) (Average time for accessing data)	Patterns are stored in 3 parts. The variable on the left-hand side of an Assign statement is stored in a vector with generic T. The expression on the right-hand side of an Assign statement is stored in a vector with generic U. Each Assign statement is stored in the same index for both vectors. Lastly, we make use of the indexing of the vectors and made a map of the index to the statement number of the Assign statement.	Easily extendable due to the use of generics. More difficult to maintain as we keep track of 2 vectors for the left and right data, as well as 2 maps for indexing.

2.2.2.1 Application of Design Principles

Single Responsibility Principle. Separating the base class into ReadOnly and WriteOnly components allows us to adhere to SRP as the function for reading and writing of entities can be handled by a fundamental base class.

Open-Closed Principle. A Manager class is introduced for each specific entity and relationship, which inherits from their respective base Manager class where most of the logic is written. To account for new entities/relationships in the future, we can simply make a new Manager class and inherit from the relevant base Manager class, making it open to extension with minimal modification.

Interface Segregation Principle. We decided to split the interface into smaller read-only and write-only interfaces which are specific to each base Manager class to ensure that our base Manager classes will only have methods that are of interest to them. This also helps with SRP as the reading and writing of Entities/Relationships/Patterns are separated into different interfaces.

2.2.2.2 Use of Design Patterns

<table>
<tr><th>Component</th><th>Possible Patterns</th><th>Considerations</th></tr>
<tr><td rowspan="2">StorageManager

<u>Selected</u>: Facade Pattern</td><td>Facade</td><td>The Facade pattern allows us to hide the complexities of how the PKB manages the reading and writing of Entities/Relationships/Patterns. The ReadStorage and WriteStorage allow the SP/QPS to easily leverage on relevant subsystems by calling the relevant getters for the Managers. The Facade pattern allows us to control SP/QPS by limiting access to only components and features that are needed by them.</td></tr>
<tr><td>Bridge</td><td>The Bridge pattern lets us make use of object composition, allowing us to extract one of the dimensions into a separate class hierarchy. In our case, we considered extracting the read/write code into its own class with the manager classes referencing the read/write object.

Ultimately, we decided to go with the Facade pattern due to its ease of implementation and lower complexity compared to the Bridge pattern.</td></tr>
<tr><td rowspan="2">Entity/ Relationship/ Pattern Manager

<u>Selected:</u> Factory</td><td>Abstract Factory</td><td>Abstract Factory and Factory patterns are both creational design patterns with the main difference of creating objects through composition for the Abstract Factory and through inheritance for Factory.

Factory pattern was chosen as each manager is the base class. For example, for entity manager, the base EntityManager class where all the logic is defined is inherited by the child managers which only define the specific data type they take.</td></tr>
<tr><td>Factory</td><td>Factory method was chosen specifically as it is easy to extend. Any new entities, relationships or patterns can be simply added by creating a new child manager that inherits the appropriate base class.

Abstract factory was not chosen as it was overcomplicated - no composition was needed. It was also more complicated to implement than Factory method.</td></tr>
</table>

2.2.3 Changes from Previous Milestone

Previously, the PatternManager was the only class for patterns and handled the insertion of Assign patterns. Now, PatternManager is a base class that uses generics, so that its child managers can define appropriate types for different patterns and ensure OCP. The change is reflected in the creation of an additional child pattern manager

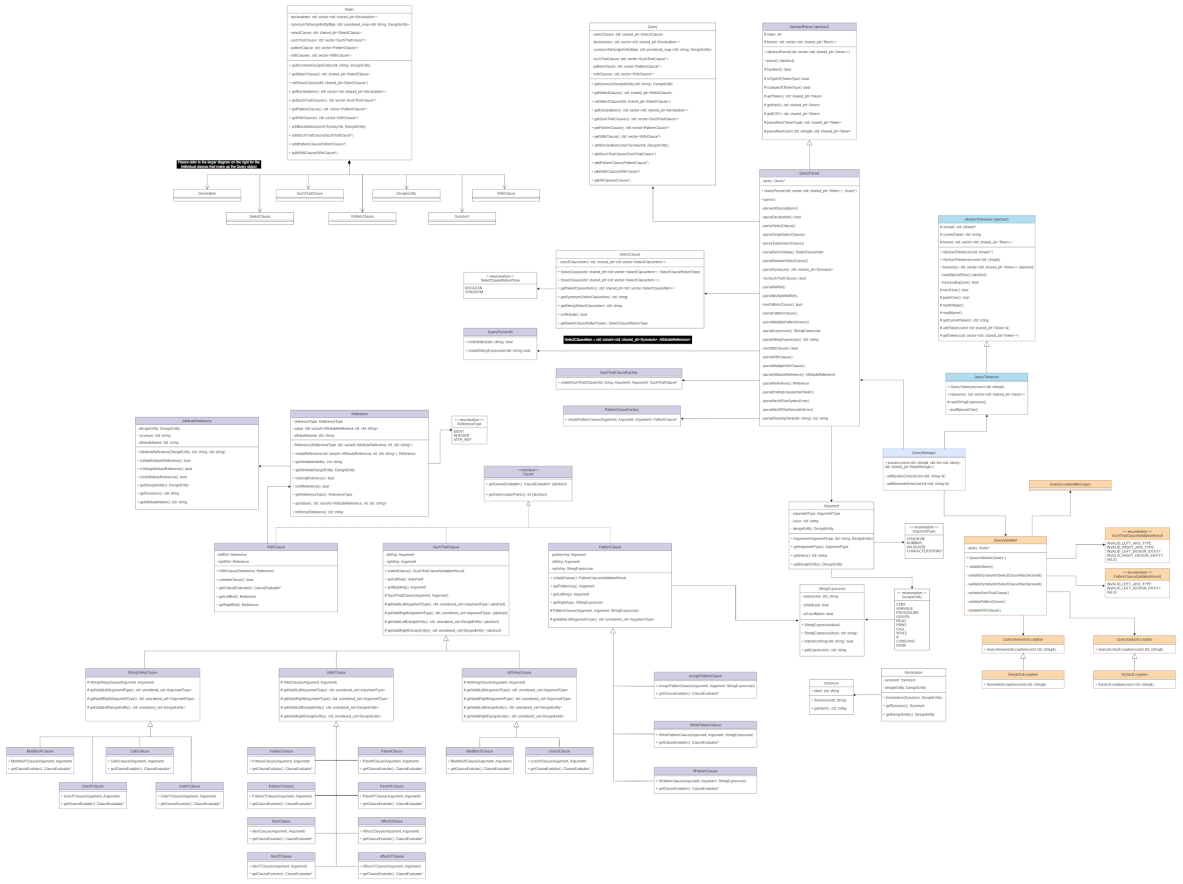
AssignPatternManager, replacing the previous sole PatternManager. Another change is that AssignPatternManager now inserts (`int: stmtNo, string: left, shared_ptr<ShuntNode>: right`), where ShuntNode is processed by the ShuntingYard in SP. This stores the expression on the right as a processed ShuntNode and reduces repeated code in the QPS.

2.3 Query Processing Subsystem

2.3.1 System Design

The QueryManager is the entry point for all QPS queries and contains four individual components. For ease of comprehension, the class diagrams are split into QPS without QueryEvaluator and only QueryEvaluator.

2.3.1.1 Class Diagram of QPS Without QueryEvaluator[8]



Class Diagram Explanation

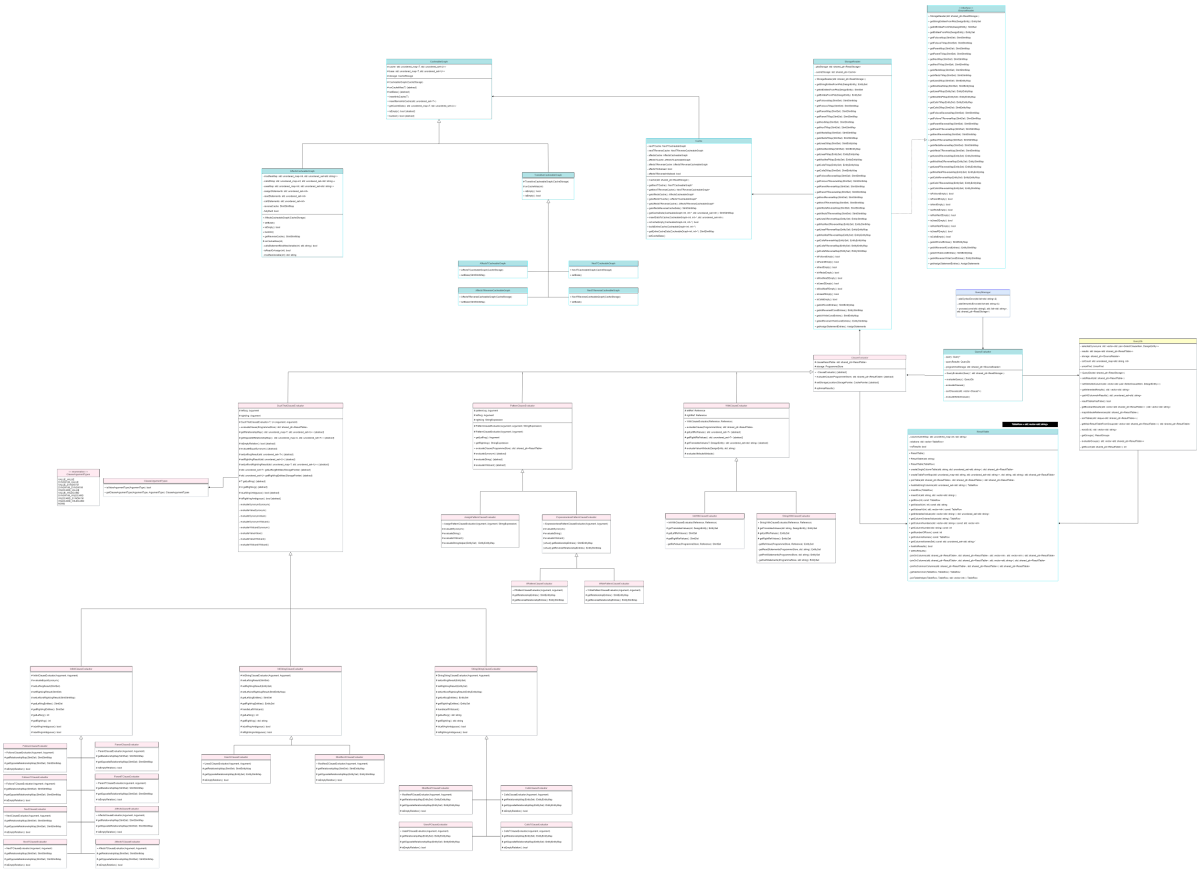
QueryTokenizer. The QueryTokenizer reads the Query provided by the user and tokenizes it into a list, it inherits from the AbstractTokenizer in the common file shared by the SP.

QueryParser. The QueryParser accepts the tokens from the QueryTokenizer and parses them into a Query object. The query object contains all the declarations, select clauses, such-that, pattern or with clauses made in the query. It inherits from the AbstractParser class in the common folder shared by the SP and catches any syntax errors then terminates the program.

QueryValidator. The QueryValidator validates every query by ensuring that the arguments in all clauses are semantically valid and abide by the PQL grammar rules. Invalid queries will throw a semantic error and terminate the program.

[8] QPS Class 1.svg

2.3.1.1 Class Diagram of QueryEvaluator[9]

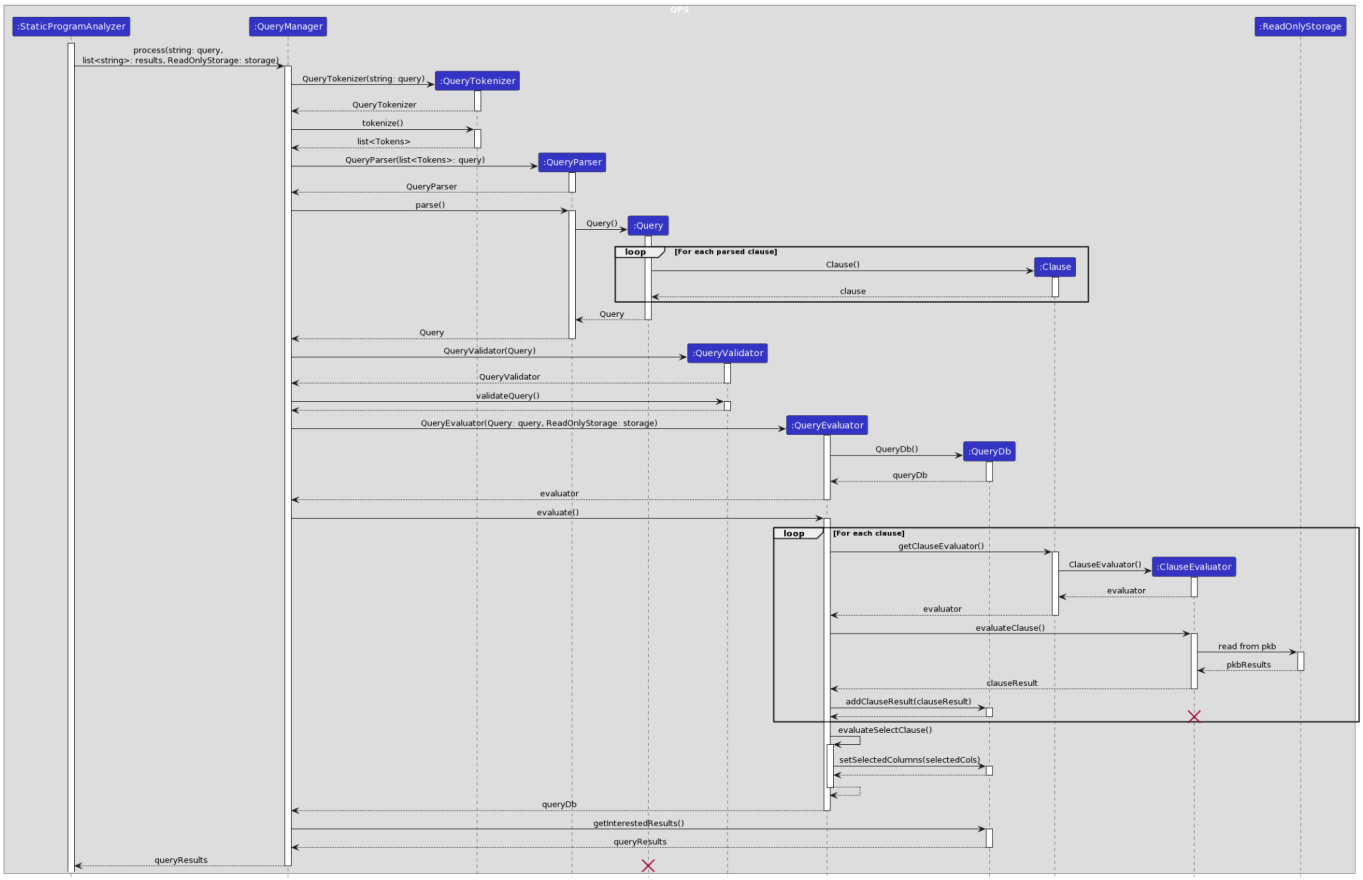


Class Diagram Explanation

QueryEvaluator. The QueryEvaluator is responsible for evaluating the query object after it has been parsed. The results of the query are stored in a QueryDb, containing a list of relational tables. The Query Evaluators are built with inheritance and polymorphism. Each evaluator for a such-that and pattern clause inherits from a SuchThatClauseEvaluator and PatternClauseEvaluator respectively, both of which inherit from a base ClauseEvaluator.

[9] QPS Class 2.svg

2.3.1.3 Sequence Diagram[10]



Sequence Diagram Explanation

When a user provides a query, it is first tokenised by QueryTokenizer then parsed by the QueryParser into a Query object. It is then passed to the QueryValidator to ensure that it is semantically correct. Evaluation is then done by the QueryEvaluator where each clause in the query is evaluated independently and the results are stored in the QueryDb. Lastly, these intermediate results are joined together and based on the selected items, the results are filtered and shown to the user.

[10] QPS Sequence Diagram.svg

2.3.2 Design Decisions

Components	Complexity	Ease of Implementation	Extensibility and Maintainability
Grouping of Results	Time to perform a union: O(α(n)) Space: O(n) n: Total number of columns specified in the query α: Inverse Ackermann function	As part of the optimisation for query evaluation, we needed to divide the clauses into groups so as to prevent any unnecessary joining. In order to keep track of which tables belongs to which groups, we decided to use a union-find data structure to keep track of the different grouping. The data structure uses path compression and weighted union to improve efficency. It is easy to implement as the data structure only contains 2 methods, find and union.	It is easy to extend and maintain as this is a general data structure class that is decoupled from the rest of the QueryEvaluator. This also allows it to be easily unit tested and reusable for other similar use cases.
Caching	Time to add a new element: O(V+E) Space: O(V+E) V: Number of nodes in the cache E: Number of edges in the cache	Some of the clauses like Next* needed to be computed on the fly, hence there is a need to cache their results during evaluation. Since evaluating an element is expensive, we wanted to prevent any unnecssary evaluations or caching. The approach we adopted was to only evaluate the interested items of a clause, rather than compute the entire result set. For example, if the user queries Next*(2,s), we only need to do a DFS on the CFG for statement 2, rather than for all statements..	We created the abstract class CacheableGraph to encapsulate the caching logic. The abstract class has a method that allows the client to add interested elements into the cache. It checks if the interested elements are present and if it is not, it will run a onCacheMiss method to evaluate and add that item into the cache. This makes it easy to create the caching for Next* and Affects/Affects* as the concrete class only needs to override the abstract onCacheMiss method.

2.3.2.1 Application of Design Principles

Single Responsibility Principle. By compartmentalising the functionality of the QPS into four main components: QueryTokenizer, QueryParser, QueryValidator and QueryEvaluator, and having the QueryManager utilise each of them, we have adhered to SRP. This has helped the QPS team in various ways such as work division, less merge conflicts, and debugging. It also decreased the amount of interconnected logic, allowing us to add new features without having to update too many classes.

Open-Closed Principle. In preparation for the new such-that clauses, we grouped the Basic SPA clauses according to their specified design abstractions, and these parent classes inherit from the SuchThatClause. By having this grandparent-parent-children hierarchy, it simplifies the addition of new clauses via inheritance.

2.3.2.2 Use of Design Patterns

Component	Possible Patterns	Considerations
Creation of Clauses **Challenge:** There are many different types of clauses in the PQL (such-that, pattern, with) and even under these parent clauses there are more child clauses to create. An ideal design pattern can help to fully implement OCP such that new clauses can be added easily without having to modify overwhelming amounts of code. **Selected:** Factory	Factory	Both creational design patterns decouple the client system from the actual implementation classes through the abstract types and factories. However, the factory method creates objects through inheritance and the abstract factory through composition. For the QPS, it was more suitable to choose the factory method because all the different types of clauses essentially only have to implement the *Clause* interface. Employing a three-level hierarchy of clauses through inheritance allows the QueryParser and QueryEvaluator to be relieved of this responsibility and creation is delegated to the factory class instead, showcasing SRP. It also highlights OCP as new clauses and their corresponding evaluators can be added easily. Using the abstract factory to contain several factory methods would have overcomplicated things. In addition, the factory method is a lot simpler to implement.
	Abstract Factory	
Such-That Clause Evaluator **Challenge**: The steps for evaluating the different such-that relations are essentially the same, apart from having to read the relationship map from different parts of the PKB. Hence, we needed to adopt a design	Template Method	Both patterns provide different ways to execute different behaviour based on the different types of the object. Strategy pattern achieves this though composition while template pattern makes use of inheritance. An advantage of using a strategy design pattern is that the evaluation algorithm can be used in more than one class. However, in this case, the evaluation algorithm for a such-that clause will only
	Strategy	

pattern that allowed us to cut down on writing any repeated code.. This also allows us to adhere to OCP as new evaluators can be easily added. **<u>Selected</u>:** Template Method		be used for evaluating such-that clauses and nothing else. The evaluation algorithms only differs based on the different subclasses in a single hierarchy. Hence, it makes more sense to use template pattern. Also, template pattern is simpler to implement as it requires less code and does not require the creation of additional files.

2.3.3 Changes from previous Milestone

<u>Dependency Inversion Principle.</u> When refactoring how the assign patterns are stored in the PKB, we realised that the AssignPatternEvaluator had to be changed. This indicated that a higher level module (AssignPatternEvaluator) depended on lower-level modules (PKB) when ideally both should depend on abstractions. Hence, we created an interface *ISourceReader* that abstracts out the interaction between the evaluator and the PKB, allowing us to change how each entity and relationship is stored in the PKB without having to change any of the evaluator classes.

3. Test Strategy

3.1 Test Approach

3.1.1 Summary

Made use of a systematic testing strategy that includes:

1. **Unit Testing**: Test a single module in isolation for correctness.
2. **Integration Testing**: Ensure different components of our systems work together properly.
3. **System testing**: Test the whole system based on requirement specifications.

Regression testing was done after milestone 1 to ensure stability and functionality after adding new design abstractions to our SPA implementation.

3.1.2 Unit Testing

Component	Common	SP	PKB	QPS
Test Case	14	14	45	97
Assertions	89	1363	208	433

Unit tests are developed along with methods, allowing us to easily find any errors in a module and ensure each module works properly. Test cases are written for all public modules. For protected methods, we wrote a Test<Class> that extends from the abstract/base class to test methods. For private methods, we ensure that test cases for public/protected methods can reach all private methods to ensure correctness.

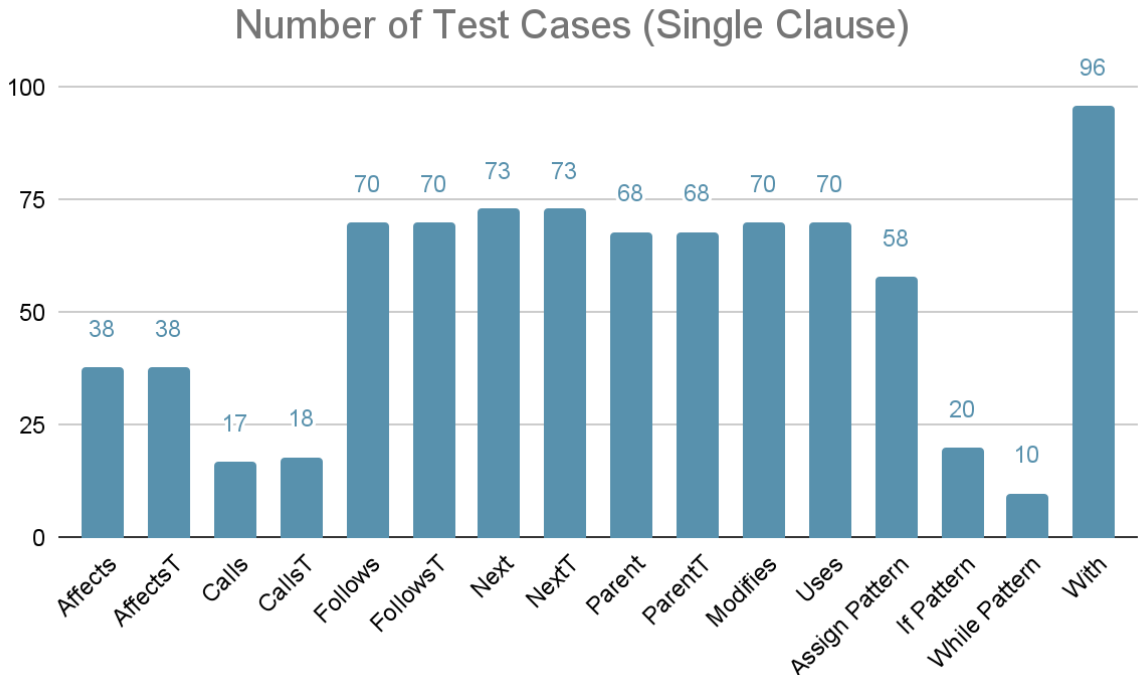
3.1.3 Integration Testing

Component	SP-PKB	PKB-QPS
Test Case	3	3
Assertions	379	94

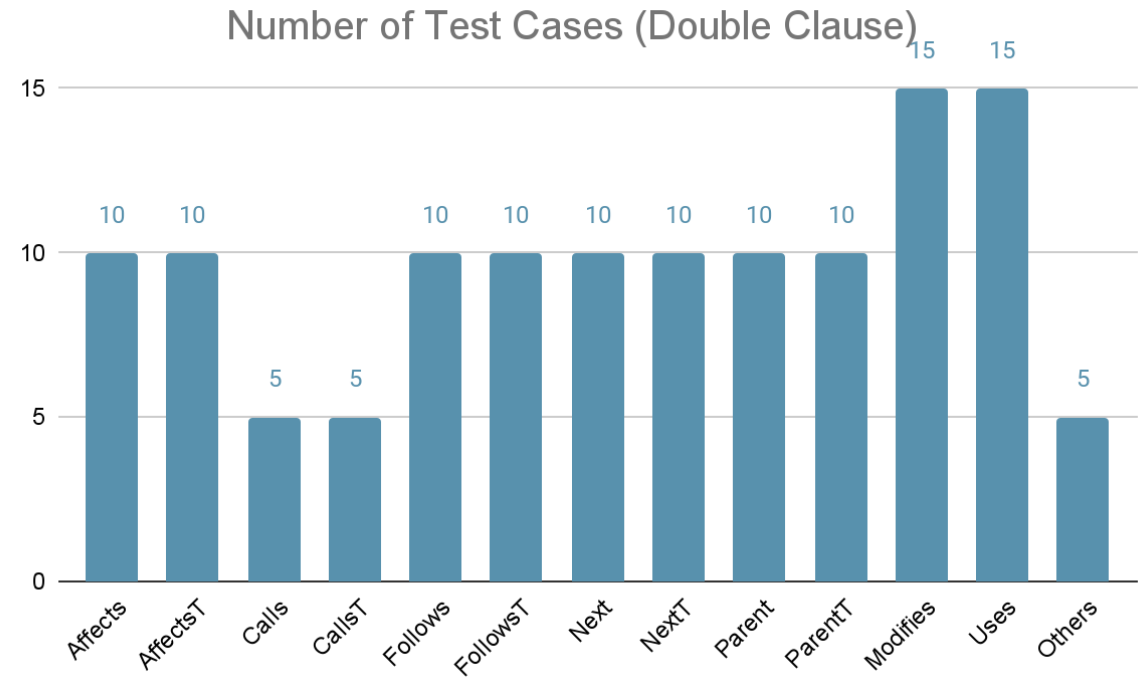
Integration Tests are added at the end of each sprint and are updated to cover the new features implemented in each subsystem. Test cases focus on the flow of data from one subsystem to another to ensure correctness.

3.1.4 System Testing[11]

Our system testing aims to be extensive and comprehensive. We came up with all possible combinations for each design abstraction. Each combination is tested at least once. A test case matrix, listing all possible entities and relationships (with all possible arguments) along with the number of clauses each query can have was created, allowing us to come up with different combinations of test cases efficiently.



We ensured that all design abstractions are accounted for. For each design abstraction, different combinations of argument types were tested at least once. Different return types were also used. A total of **857** Single Clause test cases were written.



[11] 0.6 Test Case Summary

For Double Clause System Tests, we made use of a combination of *design abstraction + with/pattern* clause for each test case. A total of **125** Double Clause test cases were written.

Number of Test Cases (Others)					
No Clause	**Multi-Clause**	**Multi-Clause (w/ Pattern)**	**Multi-Clause (w/ With)**	**Multi-Clause (w/ Pattern + With)**	**Invalid Clause**
10	**12**	**12**	**12**	**12**	**252**

For Multi-Clause Queries, we made use of a combination of design abstractions for each test case. In addition, we have also included invalid test cases to ensure that our system can detect faulty queries. In total, we have run a total of **1292** test cases.

3.2 Automation Approach

Our unit, integration and system tests run automatically during our CI workflow. This serves as a form of regression testing as every existing test suite is tested again after changes were made to the implementation. Any failing test cases will be flagged before the new features are merged into the master branch.

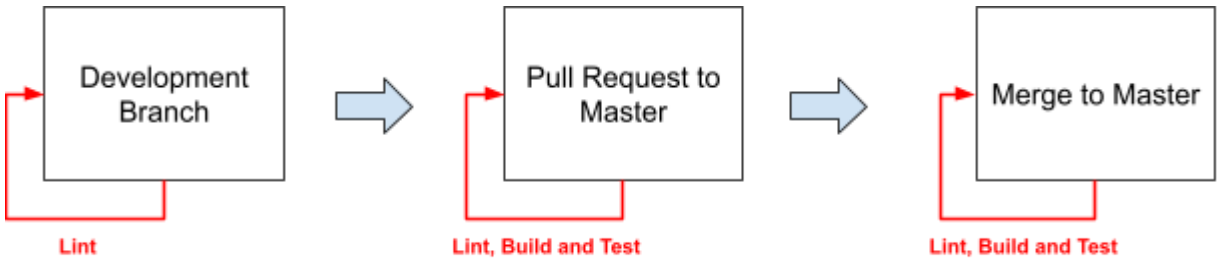
3.3 Bug Handling Approach

If an anomaly is detected through the autotester or CI workflow, the developer investigates the root cause. When uncertain, they discuss it with the team, and each component developer examines their component. Once the bug is identified, a bug ticket is created in GitHub Issues and assigned to the component developer. After resolving the bug, the developer reruns test cases and opens a PR to merge the changes.

4. Continuous Integration

The development process prioritizes code accuracy, requiring the master branch code to be both compilable and meet quality standards. This quality standard comprises two components: code quality, and code correctness.

4.1 Summary



Due to the limitations of the GitHub Actions usage guidelines provided, we have decided to run non-schedule workflows on self-hosted runners.

Trigger / Jobs	Lint	Build	Unit Test	Integration Test	System Test	Codecov
Push	All branches	Master branch only				
Pull Requests	-	All pull request				Only pull request affecting test folders
Cron (for MacOS)	-	Master branch only			-	-

4.3 Linting

The code follows the Google C++ Style Guide for linting, promoting consistent and readable code. Linting detects potential issues, aiding in debugging and ensuring the quality of the code merged into the main branch. Cpplint, implemented with Python, performs the linting process.

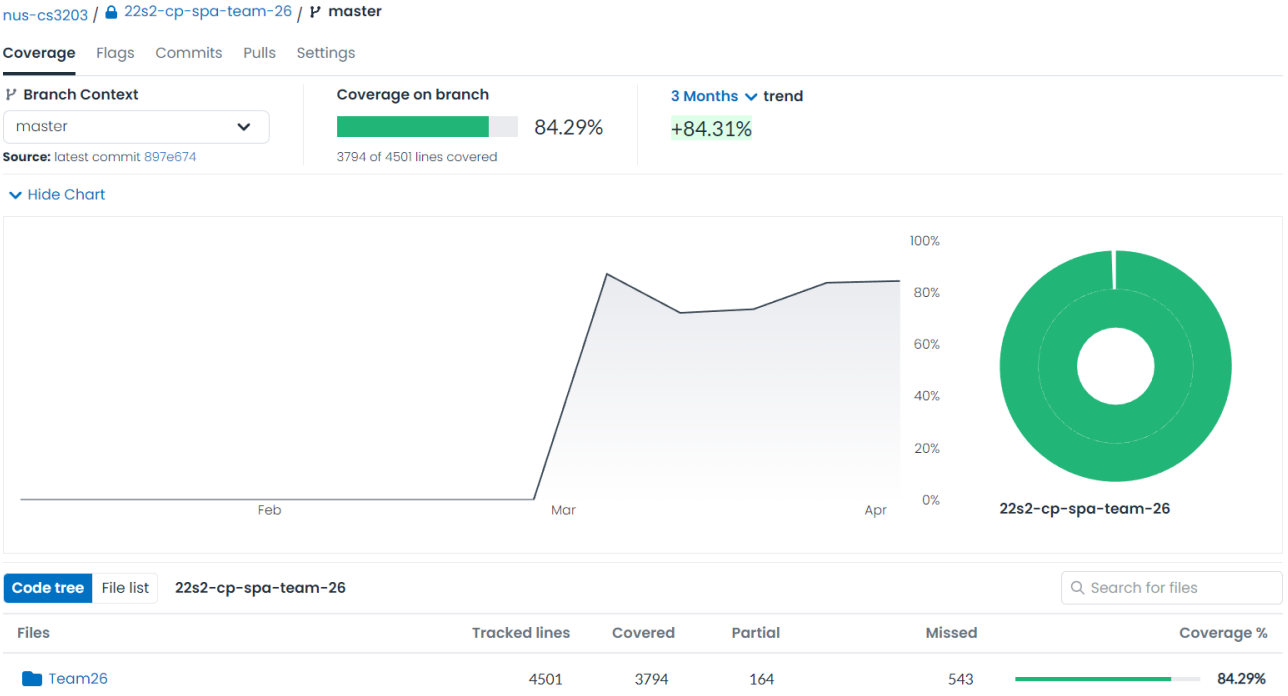
4.4 Build and Testing

We build the project and run all Catch2 Unit and Integration Tests. Additionally, we developed a *tester.js* script that runs on NodeJS, which automatically performs system testing. We store our system tests in the Tests26 folder, with the test cases being separated by clauses into different folders. The results are uploaded to Azure linked via a Google Sheet if an error is detected.

4.5 Code Review

To bolster the quality of the codebase, PR to the master branch requires mandatory code reviews and must be approved by at least one reviewer who did not author the code. This practice ensures thorough assessment and helps maintain high code quality. Furthermore, static analysis tools for C++ code such as cpplint and cppcheck were used to manually check for any code smells.

4.6 Codecov[12]



We deployed CodeCov to ensure that our coverage is maintained and tracked.

[12] https://app.codecov.io/gh/nus-cs3203/22s2-cp-spa-team-26

5. Reflections

5.1 Problems Encountered

➔ Being unable to choose the other three members of our group, we were bound to end up with people of varying skills, working styles and timetables. We had to look past these differences in order to streamline our progress through strict division of labour and effective communication.

➔ Initially, communication breakdowns hindered and delayed our progress but were eventually resolved as we learnt more about each other's working styles. Slow coding progress due to C++ language unfamiliarity and limited experience with project management took time to overcome.

5.2 Improvements

➔ Read milestone requirements and project specifications properly to better allocate time and work on components that were more tedious.

➔ Start testing earlier to leave sufficient time for bug-fixing.

5.3 What Went Well

➔ Equal division of work and setting of clear goals at the start of every sprint/milestone.

➔ Conflicts regarding scheduling and different ideas were often resolved quickly as a team.

➔ Seeking clarification promptly when we encounter unfamiliar issues/problems with our TA to prevent any blockers.

5.4 Management Lessons

➔ A good and capable leader forms the core of the group, which we were fortunate to have. Our group leader commanded our project by leading discussions, keeping track of progress and deadlines, and tying up any loose ends. Complications were quickly addressed and unattended work was re-delegated to someone else who could cope better.

➔ For such a large project with tight deadlines, we learnt that it is extremely important to practice open communication as any delay in progress will severely affect subsequent issues. During crunch periods like midterms, we would update each about whether we could meet the upcoming deadlines and rearrange the workload or timeline if necessary.

Appendix

Appendix I: SP API

This section describes the APIs available for the SP.

SourceTokenizer

`Tokenizer(istream* stream)`	Constructor for the Source Tokenizer
`vector<shared_ptr<Token>> tokenize()`	Tokenizes the source input

SourceParser

`Parser(vector<shared_ptr<Token>> tokens)`	Constructor for the Source Parser
`shared_ptr<ProgramNode> getProgramNode()`	Returns a shared pointer to the AST
`void parse()`	Parses the source input

DesignExtractor

`DesignExtractor(const shared_ptr<IStore>& store)`	Constructor for the DesignExtractor
`extract(const shared_ptr<ProgramNode>& programNode)`	Extracts patterns, entities and relationships from AST and stores extracted data in the PKB

StorageAdapter

`Store(const shared_ptr<WriteStorage>& storage)`	Constructor for the StorageAdapter
`shared_ptr<IEntityStore> getEntityStore()`	Returns a shared pointer to the EntityStore
`shared_ptr<IPatternStore> getPatternStore()`	Returns a shared pointer to the PatternStore
`shared_ptr<IRelationshipStore> getRelationshipStore()`	Returns a shared pointer to the RelationshipStore

Appendix II: PKB API

This section describes the APIs available for the PKB.

StorageManager

`shared_ptr<ReadStorage> getReadStorage()`	Get Storage instance with Read-Only functionality
`shared_ptr<WriteStorage> getWriteStorage()`	Get Storage instance with Write-Only functionality

StorageUtil

`std::unordered_map<T,std::unordered_set<U>> getRelationshipMap(std::shared_ptr<IReadRelationshipManager<T, U>> relation)`	Get map containing all relationships of a certain Relationship child manager
`std::unordered_map<T,std::unordered_set<U>> getReversedRelationshipMap(std::shared_ptr<IReadRelationshipManager<T, U>> relation)`	Get reversed mapping containing all relationships of a certain Relationship child manager
`bool isRelationEmpty(std::shared_ptr<IReadRelationshipManager<T, U>> relation)`	Check if there are no relationships for a certain Relationship child manager
`bool relationContains(std::shared_ptr<IReadRelationshipManager<T, U>> relation, T first, U second)`	Check if relationship exists for a certain Relationship child manager
`std::unordered_set<T> getEntityValues(std::shared_ptr<IReadEntityManager<T>> entityManager)`	Get set containing all entities of a certain Entity child manager
`std::unordered_map<int, int> getAllPatternEntries(std::shared_ptr<IReadPatternManager<T, U>> patternManager)`	Get map of all patterns with index as key of a certain Pattern child manager
`std::unique_ptr<std::vector<T>> getAllLhsPatternEntries(std::shared_ptr<IReadPatternManager<T, U>> patternManager)`	Get vector containing all entries of left hand side of assignments of a certain Pattern child manager
`std::unique_ptr<std::vector<std::shared_ptr<ShuntNode>>> getAllRhsPatternEntries(std::shared_ptr<IReadPatternManager<T, U>> patternManager)`	Get vector containing all entries of right hand side of assignments of a certain Pattern child manager

IReadRelationshipManager

`bool isEmptyMap()`	Check if there are no relationships in the unordered map
`bool isEmptyReversedMap()`	Check if there are no relationships in the unordered map with key, value reversed
`bool containsMap(T firstParam, U secondParam)`	Check if relationship (first, second) exists in map
`bool containsReversedMap(U secondParam, T firstParam)`	Check if relationship (second, first) exists in reversed map
`unordered_map<T, unordered_set<U>> getAllRelationshipEntries()`	Get map containing all the relationships with T as key

`unordered_map<U, unordered_set<T>> getAllRelationshipEntries()`	Get map containing all the relationships with U as key

IWriteRelationshipManager

`bool insertRelationship (T firstParam, U secondParam)`	Insert relationship into Storage

IReadEntityManager

`bool isEmpty()`	Check if there are no entities
`bool contains(T entity)`	Check if entity exists
`unordered_set<T> getAllEntitiesEntries()`	Get set containing all entities

IWriteEntityManager

`bool insertEntity(T nameOrStmtNo)`	Insert entity into Storage

IReadPatternManager

`bool isEmptyLhsVector()`	Check if lhs_vector is empty
`bool isEmptyRhsVector()`	Check if rhs_vector is empty
`bool isEmptyIndexStmtMap()`	Check if index_stmt_map is empty
`bool isEmptyReversedIndexStmtMap()`	Check if reversed_index_stmt_map is empty
`bool containsLhsVector(T left)`	Check if left hand side of the assignment entry exists in the vector
`bool containsRhsVector(U right)`	Check if right hand side of the assignment entry exists in the vector
`bool containsIndexStmtMap(int index, int stmtNo)`	Check if (index, stmt_no) exists in the map
`bool containsReversedIndexStmtMap(int stmtNo, int index)`	Check if (stmt_no, index) exists in the reversed map
`unique_ptr<vector<T>> getAllLhsPatternEntries()`	Get vector containing all entries of left hand side of assignments
`unique_ptr<vector<U>> getAllRhsPatternEntries()`	Get vector containing all entries of right hand side of assignments
`unordered_map<int, int> getAllPatternEntries()`	Get map of all patterns with index as key
`unordered_map<int, int> getAllReversedPatternEntries()`	Get map of all patterns with statement number as key

IWritePatternManager

`bool insertPattern(int stmtNo, T left, U right)`	Insert pattern into Storage

Appendix III: QPS API

This section describes the APIs available for the QPS.

QueryManager

`process(const string&, list<string>, shared_ptr<ReadStorage>)`	Processes the query input by creating a query object, tokenizes, parses, validates and evaluates the QPS query then adding to the QPS result

QueryTokenizer

`QueryTokenizer(const string&)`	Constructor for the Query Tokenizer
`void tokenize()`	Tokenizes the query input

QueryParser

`QueryParser(vector<shared_ptr<Token>>, Query*)`	Constructor for the Query Parser
`void parse()`	Parses tokens and returns Query wrapper object

QueryValidator

`QueryValidator(Query*)`	Constructor for the Query Validator
`void validateQuery()`	Validates the query input

QueryEvaluator

`QueryEvaluator(Query*, shared_ptr<ReadStorage>)`	Constructor for the Query Evaluator
`QueryDb evaluateQuery()`	Evaluates the query input

Appendix IV: Extension Proposal

1. Array Extension Proposal

We propose to support the Array data structure in SIMPLE source code and allow PQL queries for this new data structure. This is classified as a new design entity.

1.1 Definition

An array is initialized by `a = [1, 2, 3]` and is zero-indexed. Values in the array can be accessed by `arr_name[index]`, eg: `a[0]`

1.2 Constraints

➔ Only integers are supported.

➔ Arrays cannot be empty.

➔ Arrays cannot be edited, only reassigned.

➔ Variables defined as arrays cannot be used in operations like "+". For example, "array + 1" is not supported.

➔ Multidimensional arrays are not supported.

➔ Arrays cannot be reassigned to variables initialized as non-array values and vice versa.

2.2 Changes to Existing System

2.2.1 SP

➔ `AssignNode` will support additional right-side values using generics.

➔ The variables that are related to expressions in the `AssignNode` will be abstracted into an ExprNode.

➔ `extractAssign` method will have to call the specific extraction methods of the type provided by the generics.

➔ Adherence to array usage rules will be verified in the SourceParser.

➔ Adherence to reassignment rules will be verified in the DesignExtractor.

➔ Additional interfaces will need to be introduced for the typed nodes supported by `AssignNode` generics.

2.2.2 PKB

➔ Two additional managers inheriting from `EntityManager`, `ArrayNameManager` which stores the array name and `ArrayStmtNoManager` which stores assign statements related to arrays.

➔ Additional manager inheriting from `PatternManager`, `ArrayPatternManager` which handles the storing of array name and a vector of its values.

2.2.3 QPS

Support for design abstractions

➔ Support additional design entity called `array`; the user can declare `array a`, to represent all declarations of design entity `array`.

➔ Support additional attribute references.

- Users can use `assign.arrAssign#` to specifically get the assign statements related to arrays (e.g. x = [1, 2]), as well as `assign.varAssign#` to specifically get assign statements related to variables (E.g., x = 1 + 2).
- Users can use `array.index[0]` to get the value at the index specified. If the index is out of bounds, return an empty result.

2.3 Implementation

2.3.1 SP[13]

```
// AssignNode
template <typename T>
class AssigNode : public VariableNameNode, public std::enable_shared_from_this<AssigNode>  {
public:

const T rightSideValue;

AssignExprNode(int stmtIndex, std::string varName, T rightSideValue);
}

// ExprNode
class ExprNode : public Node {
public:

const std::shared_ptr<ShuntNode> shutNode;
const std::unordered_set<std::string> exprVariables;
const std::unordered_set<int> exprConstants;

ExprNode(std::shared_ptr<ShuntNode> shutNode,
std::unordered_set<std::string> exprVariables,
std::unordered_set<int> exprConstants);
}

// ArrayNode
class ArrayNode : public Node {
public:

const std::vector<int> arrayVariables;

AssignExprNode(std::vector<int> arrayVariables);
}
```

➔ We will abstract the variables holding the expression-related variables into ExprNode from the `AssignNode`.

➔ We then make `AssignNode` into a template node so that specific RHS of the types of values can be assigned.

➔ We create an `ArrayNode` that will serve as a type in the `AssignNode`.

[13] Additional interfaces and methods will be introduced but are not shown here to not further complicate the explanation.

2.3.2 PKB

- ➔ `ArrayNameManager` and `ArrayStmtNoManager` inherit from EntityManager.
- ➔ `AssignManager` will store both normal and array assign statements. Current assign statements will be added as per the status quo, but for new array declarations, statement number will be stored in both `AssignManager` and `ArrayStmtNoManager`.
- ➔ For array declaration `numArray = [1, 2, 3, 4]`, numArray will be stored in `ArrayNameManager` and each integer will be stored in `ConstantManager`. Statement number will be stored in `ArrayStmtNoManager`. Modifies relationship will store the statement number and array name by calling `insertRelationship(stmtNo,numArray)` from `ModifiesSManager`. There will be no Uses relationship for array declaration.
- ➔ e.g. `x = arr[0]`. Modifies relationship stores statement number and the left-hand side variable by calling `insertRelationship(stmtNo, x)` from the `ModifiesSManager`. Uses relationship stores the statement number and array name on the right-hand side by calling `insertRelationship(stmtNo, arr)` from the `UsesSManager`.
- ➔ To support indexing, we use a new ArrayPatternManager. For each array, the SP will call `insertPattern(int: stmtNo, string: arrayName, vector<int>: numArray)`.

2.3.3 QPS

PQL Query Example:

```
// Query 1 - returns all arrays in the source code
array arr;
Select arr

// Query 2 - returns the element that is at position index 0 of array arr
array arr;
Select arr.index[0]
```

- ➔ New design entity `ARRAY` added to DesignEntity enumeration class.
- ➔ Add ARRAY as an accepted design entity for Uses and Modifies.
- ➔ Support new attribute references such as `array.index[0]`, `assign.arrAssign#` and `assign.varAssign#`.
- ➔ For evaluation, support the new design entity and attribute references by calling the newly created relationship and entity managers in the PKB.

2.4 Challenges, Testing and Mitigation

2.4.1 Challenges

A key challenge is the choice of whether to use variant, inheritance or generics for making the `AssignNode` extensible. Generics was chosen because it allows us to easily create Node types that can serve as the right side value of an assigned statement.

However, this might make extraction logic in the `extractAssign` method used in the visitor design pattern complex as it will have to handle extraction for different types provided.

A second challenge is the use `Select array.index[#]` query will likely affect the Affects query which will require some changes on the QPS side.

2.4.2 Testing

This section details the additional testing that will be introduced or updated to accommodate the new changes.

2.4.2.1 Unit Testing

1. All subcomponents will have to add additional unit testing or update existing unit testing to include the new changes.
2. Additionally, unit testing will have to be developed for the new classes added.

2.4.2.2 Integration Testing

The new managers in the PKB created will be tested to ensure correct reading and writing of data for both pipelines, SP to PKB and PKB to SP.

2.4.2.3 System Testing

1. A new set of test cases will be introduced to test querying array values in the index, especially for the Uses and Modifies clauses.
2. The array design entity will be added to all affected clauses and tested for correctness.

2.4.3 Mitigation

To mitigate the first challenge, the extraction logic of each individual type should be abstracted to the typed nodes (i.e. `ExprNode`, `ArrayNode`). We can apply a Chain of Responsibility design pattern on top of the Visitor Design Pattern to ensure that the method remains short.

To mitigate the second challenge, changes will have to be properly considered and additional rules might be introduced to further constrain arrays.

2.5 Benefits to SPA

➔ Able to declare and use multiple constants in one statement instead of multiple statements.

➔ Source code is more realistic and brings it closer to actual programming languages. Arrays are fundamental in most object-oriented programming languages like Java and Python, both of which are widely used.

➔ Improves realism of the Static Program Analyser (SPA) developed in SIMPLE. Other static program analysers for real-life programming languages likely have support for arrays and their functionalities due to their prevalence. This allows SPA to have more similarities with properly developed static program analysers used in the industry.

Appendix V: Project Management

Software Development Life Cycle

Workflow

Our team uses Github Issues as our main workflow and project management tool. The sprints are tracked using the milestone feature on GitHub Issues. Issues are added, labelled and assigned at the start of each sprint. At every standup we review the progress of development and check for any blockers. Each issue is to be labelled with a module, status and type.

Module	Status	Type
Module/SourceProcessor	Status/Cancelled	Type/Bug-Fix
Module/ProgrammeKnowledgeBase	Status/High-Priority	Type/Documentation
Module/QueryProcessingSubsystem	Status/Icebox	Type/Enhancement
Module/Others	Status/Low-Priority	Type/Feature

Branching Strategy

We create a new branch for each new issue. However, if the issues are closely related, one branch may be used to develop or fix multiple issues. Our branch naming conventions are as follows:

- ➔ feature/*<feature-name>*: introduces a new feature to the codebase
- ➔ hotfix/*<-bug-name>*: patches a bug in your codebase
- ➔ test/*<feature-name>*: adding tests cases for features
- ➔ refactor/*<component>*: refactoring a component

Commit Naming

We practice a subset of the conventional commits to better describe each of our commits.

- ➔ fix: patches a bug in your codebase
- ➔ feat: introduces a new feature to the codebase
- ➔ test: adding tests cases related item
- ➔ chore: updating grunt tasks etc, implementation (of an existing feature)
- ➔ doc: documentation only changes
- ➔ style: changes that do not affect the meaning of the code
- ➔ refactor: code change that neither fixes a bug nor adds a feature

Appendix VI: Self-Hosted Runners

Below are the runners setup on a Windows 10 Host Machine and an AWS Lightsail Cloud VPC.

Self-Hosted Runner	**Type**[14]	**Machine**
windows-host-1	tester	Windows 10 Host
windows-host-2	tester	Windows 10 Host
windows-host-3	tester	Windows 10 Host
windows-host-4	builder	Windows 10 Host
ubuntu-docker-2	linter	Ubuntu Docker on Windows 10 Host
ubuntu-aws-2	linter, codecov	AWS Lightsail Host

[14] Tester runners are used for unit, integration and system tests.