CS3203 Software Engineering Project

AY23/24 Semester 2

Project Report – System Overview

Team 11

Team Members	Student No.	Email
AMADEUS ARISTO WINARTO	A0221733N	e0559265@u.nus.edu
BERNARDUS KRISHNA	A0196717N	e0389203@u.nus.edu
CHAN CHOON YONG	A0222743L	e0560275@u.nus.edu
CHEN HSIAO TING	A0222182R	e0559714@u.nus.edu
NG JUN WEI, TIMOTHY	A0217635A	e0543671@u.nus.edu
SIMON JULIAN LAUW	A0196678A	e0389164@u.nus.edu

Consultation Hours: Tuesday, 1pm–2pm

Tutor: Sumanth

Table of Content

1. System Architecture

The Simple Program Analyzer (SPA) consists of 3 main components - Source Processor (SP), Program Knowledge Base (PKB), and Query Processing Subsystem (QPS). The following Fig 1 shows the high level architecture diagram. The dashed lines indicate the interactions between the major components and external users.

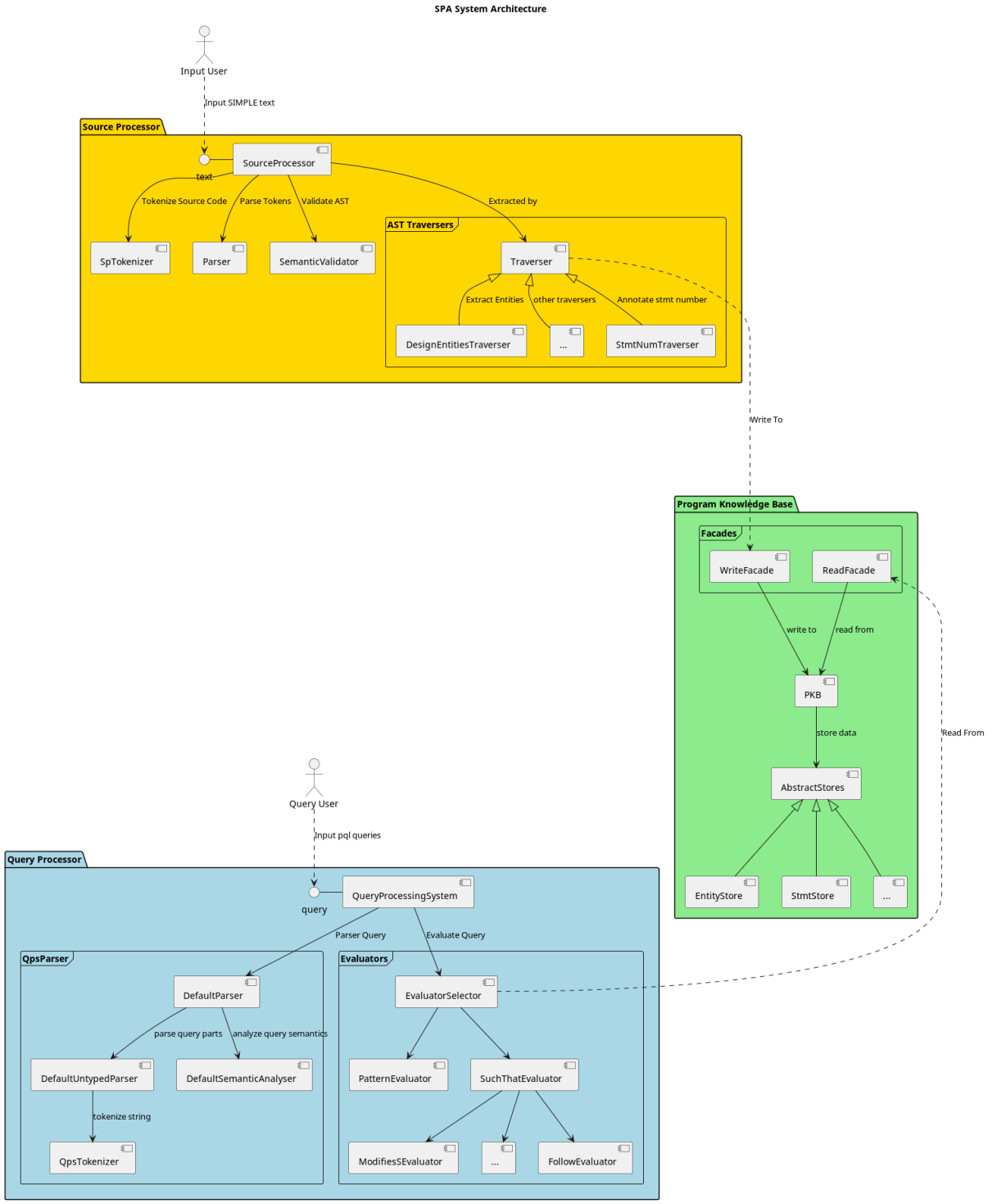


Fig 1. System Architecture Diagram of

Moreover, for higher resolution diagrams, they can be found at https://nus-cs3203.github.io/documentation/. The website has been segmented into SP, PKB, and QPS sections that contain the relevant diagrams.

2. Component Architecture and Design

2.1 SP

2.1.1. Seq Diagram of Major SP Subcomponents (Multi-Pass Approach)

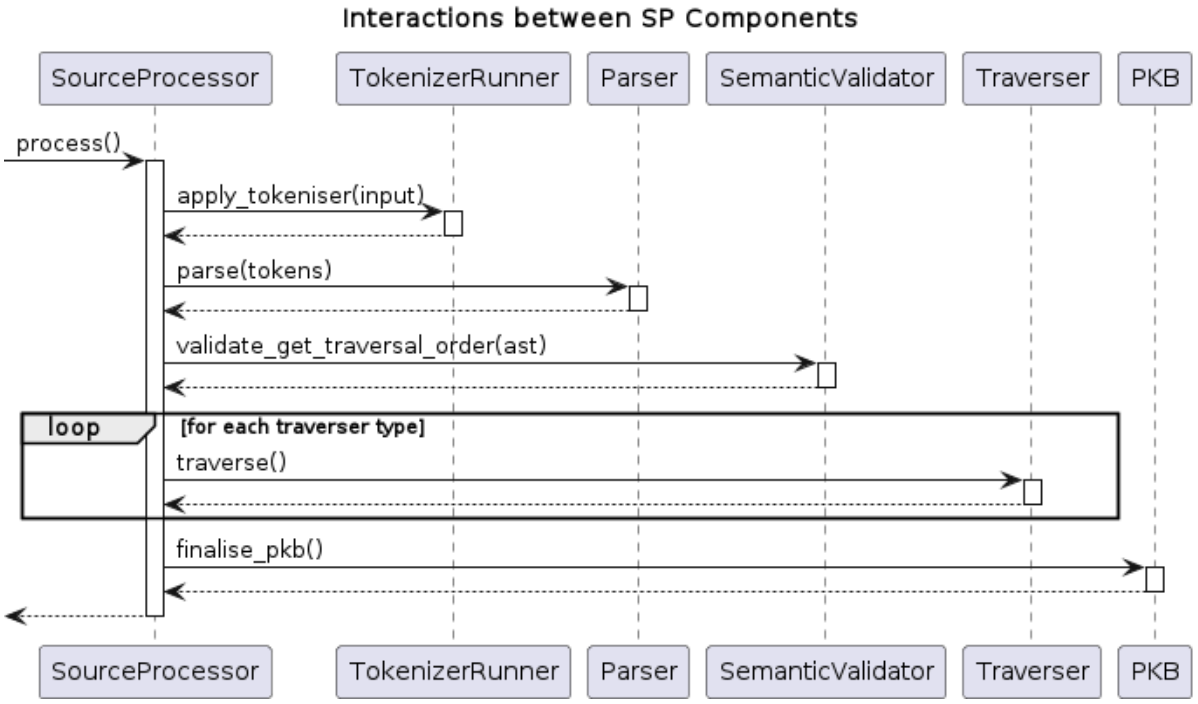


Fig 1. Sequence Diagram of Source Processor (SP)

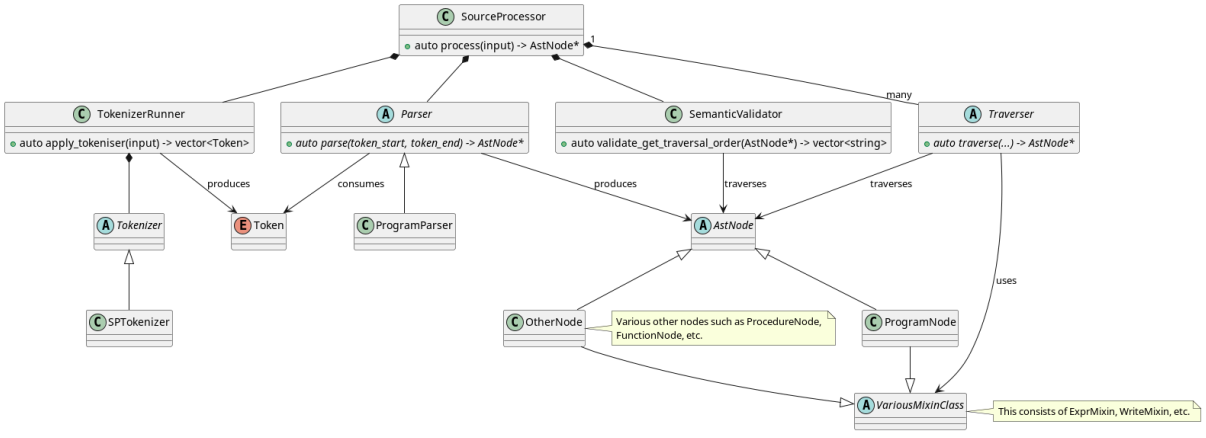


Fig 2. Class Diagram of Source Processor

2.1.3. List of APIs

SourceProcessor's public APIs are listed as follows:

SourceProcessor APIs
std::shared_ptr<AstNode> process(std::string input) Parses the input SIMPLE source program, and adds discovered Design Abstractions to PKB. Returns the root Program AstNode.

2.1.4. Design Decisions

Decision 1: How should we organize the different various Parser and Tokenizer algorithms?

Alternatives	Extensibility	Maintainability	Ease of Implementation
Single Class with giant if-else or switch-case	Low	Low	High
Composite Pattern	High	Moderate	Moderate

Decision 2: Visitor Pattern vs Mixin Pattern

Alternatives	Maintainability	Extensibility	Adherence to ISP
Visitor Pattern	Moderate	Moderate	Moderate
Mixin Pattern	Low	High	High

2.2 PKB

2.2.1. Class Diagram of PKB

Fig 4. shows the class diagram of the classes implemented in PKB.

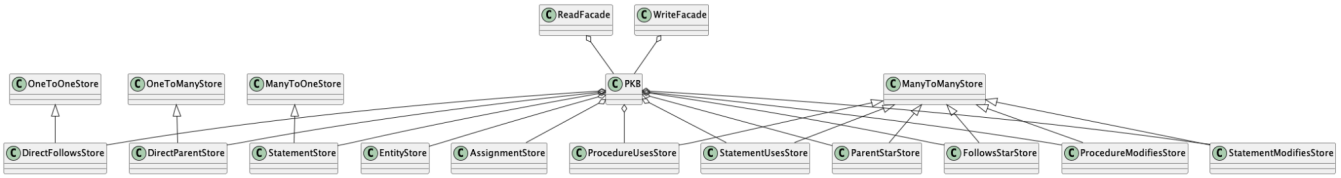


Fig 4. High Level Class Diagram for PKB

For a bigger version of this diagram, click here.

Decision 1: Facade Classes or Direct Interactions

Alternatives	Extensibility	Maintainability	Ease of Implementation	Adherence to SRP
Exposing entire PKB and all its stores methods	High. Direct access allows for quick modifications and additions	Low. Complexity and interdependencies between stores can lead to a fragile system	High. Easy to implement as only one change is needed in the respective store	Low. Every store is responsible for both updating PKB and interacting with SP or QPS
Exposing PKB APIs only through WriteFacade and ReadFacade	Moderate. Any new requirements would require changes in both the stores and the Facade classes	High. Facades hide the underlying complexity of PKB and minimise unintended access to the stores, reducing unintended side effects	Moderate. Changes in API of the stores would also require changes in Facade classes	High. Adhere SRP as WriteFacade and ReadFacade are only responsible for writing and reading respectively

Decision and Justification

We chose to implement Facade classes, option 2. Instead of having each store directly interacting with SP and QPS, we have the `WriteFacade` class which is solely responsible for all interactions between PKB and SP, and the `ReadFacade` class which is solely responsible for all interactions between PKB and QPS. This protects PKB from being exposed to external classes and prevents unauthorized or unintended modifications to the PKB's internal structures.

Decision 2: Individual Stores or Abstract Stores

There were three alternatives considered:

- Implement individual store separately (e.g. `FollowsStore` , `ParentStore`, `UsesStore`, etc do not share any parent class, and their class members do not share any parent class as well)
- Implement abstract stores and use them for encapsulation (e.g. `UsesStore` has two `ManyToMany` stores, one for `ProcedureUses` and one for `StatementUses` relation)
- Implement abstract stores and use them for inheritance (e.g. both `ProcedureUsesStore` and `StatementUsesStore` inherit from `ManyToManyStore`)

The following lists down our considerations when deciding which alternative to choose.

Alternatives	Extensibility	Maintainability	Ease of Implementation	Adherence to DRY
Implement individual store separately	Low. Every new requirement will need similar changes across all stores that have the same behaviors	Low. Every change in requirement will need changes across all stores that have the same behaviors	Low. Many repetition of codes for all similar stores	Low. Every relationship store will share very similar codes.
Implement abstract stores and use them for encapsulation	Moderate. When extending, changes are needed in the respective stores	Moderate. There is a possibility of the need to modify the encapsulating class on top of the abstract stores.	Moderate. Need to create all store classes with methods that call the abstract stores APIs, a lot of repeated code	Moderate. The encapsulating classes will still have similar codes due to similar functionalities.
Implement abstract stores and use them for inheritance	High. When extending, only the abstract stores need to be changed	High. No specific relationship stores need to be maintained, only abstract stores	Moderate. High level abstraction initially. However, very easy to implement later	High. Each relationship store does not share common code as everything is inherited from the abstract stores.

Decision and Justification

We decided to use option 3, which is to implement abstract stores and use them for inheritance. All the relationship store classes implement from an abstract store directly. These relationship store classes do not have to define their own APIs, they can use the abstract store APIs directly. This means that when extending or modifying, only the abstract store classes need to be changed. This helps to improve extensibility, maintainability and reduce lots of code repetition.

2.2.2. Sequence Diagram for Data Storage

Fig 5. shows an example of the data flow when PKB WriteFacade methods are invoked by SP.

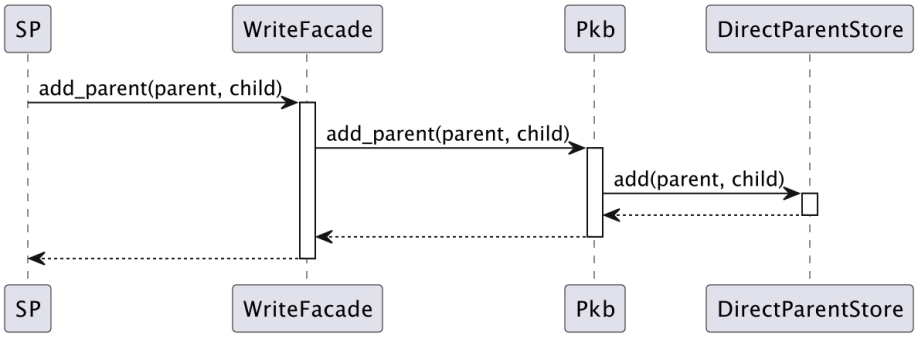


Fig 5. Sequence Diagram for adding Parent Relationship via the PKB WriteFacade

To populate the transitive relationship in Follows and Parent relationships (Follows* and Parent*), we implemented the `finalise_pkb()` method call, which SP can call after populating all the required Follows and Parent relationships. Refer to Appendix C.1 for the sequence diagram for populating the transitive relationship.

Decision 3: Calculating transitive relationship (Parent* and Follows*) after every addition or at the very end

Alternatives	Time Efficiency	Ease of Implementation
Calculate respective transitive relationship after every addition	Low. Calculating transitive relationship after every add operation can be costly	Low. There is extra logic needed depending on the order of insertion of the corresponding direct relation
Calculate all transitive relationships only once after all PKB write operations	High. Calculate transitive relationship only once, after all write operations	High. Only one API needed to populate all transitive relationships. Order of insertion does not matter as we have a guarantee that every direct relation has been inserted

Decision and Justification

We chose option 2 as it only requires one calculation at the end of all write operations. Furthermore, by doing so, we do not need to care about the order of insertion of the corresponding direct relation. By removing calculation of transitive relationships after every add operation, we can significantly improve time efficiency.

2.2.3. Sequence Diagram for Data Retrieval

Fig 6. shows an example of the data flow when PKB ReadFacade methods are invoked by QPS.

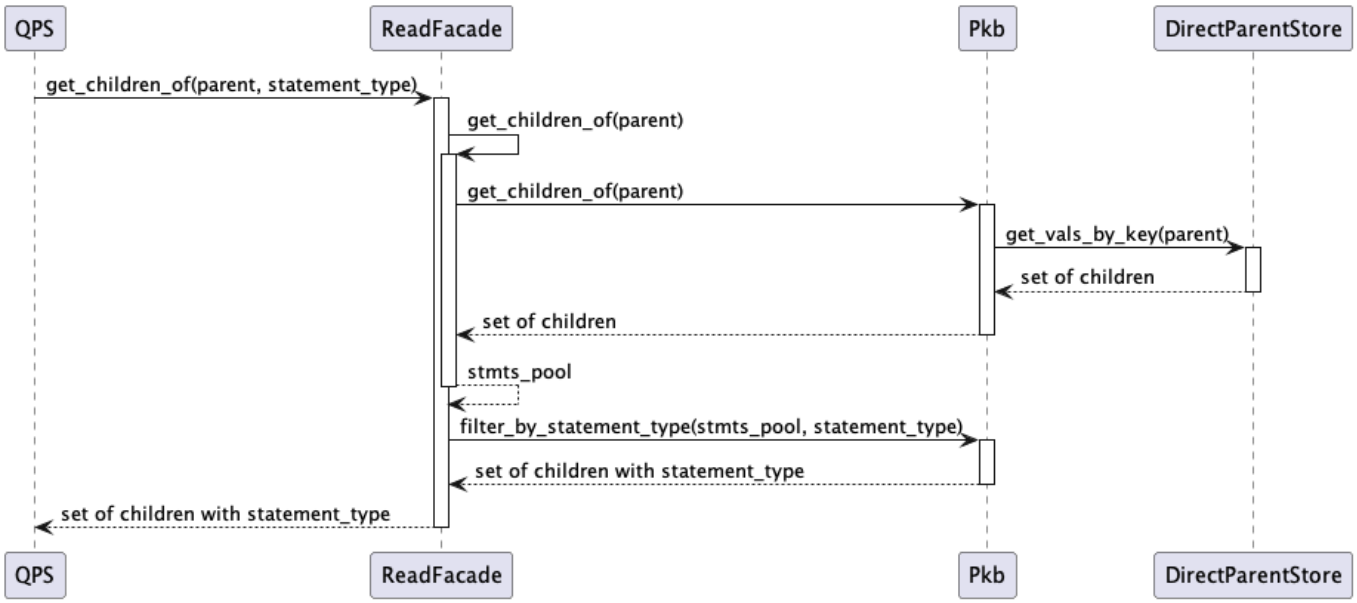


Fig 6. Sequence Diagram for Retrieval of Children by Parent and Statement Type

2.2.6. Design Principles and Patterns

Design Principles

- Single Responsibility Principle (SRP)
  Each store class is responsible for a single relationship, e.g. Parent, Follows, Modifies, Uses. Each facade class is also only responsible for either writing or reading.
- Interface Segregation Principle (ISP)
  Separate Facade classes for both reading and writing. This means that SP does not have to depend on unnecessary interfaces for reading, and QPS does not have to depend on unnecessary interfaces for writing.
- Don't repeat yourself (DRY)

Inheritance of relationship stores from abstract store superclasses help to reduce repeated code significantly. This helps to ensure DRY since we now do not have to repeat the same code logic in every store that behaves similarly.

Design Patterns

- Facade Pattern
  The Facade Pattern for PKB simplifies the interface for SP and QPS, providing a unified and high-level access point to the various stores and relationships. It enhances modularity and reduces the complexity of interactions, making the system easier to use, understand, and maintain.

2.3 QPS

2.3.1. Class Diagram of QPS

The Query Processing System (QPS) has 2 main components: the `Parser` and `Evaluator`. The `Parser` maps a given input `string` into a `Query` object, while the Evaluator processes a given `Query` object and populates a `ResultsTable`. The `ResultsTable` is used to populate the caller-provided response list.

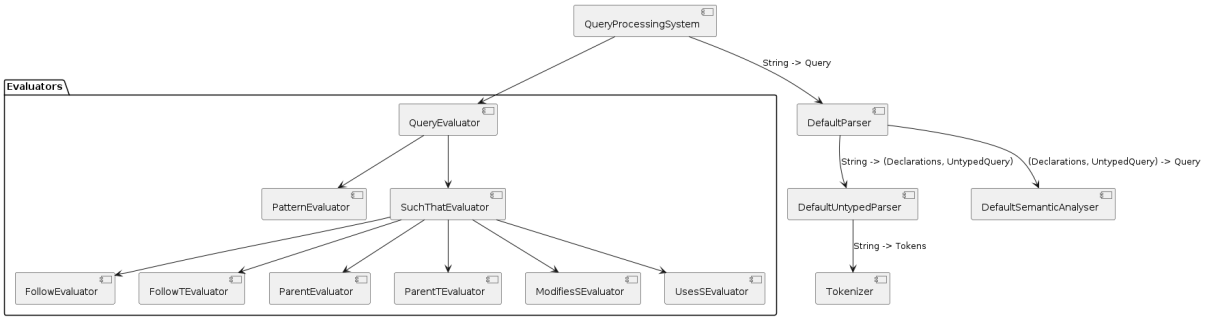


Fig 7. Class Diagram of QPS

2.3.1.1 Parsing

We split the responsibility of building the `Query` object as follows:

- `UntypedParser`: maps a given `string` into `(declarations, UntypedQuery) | SyntaxError`
  - Encodes the query grammar and convert `string` into a partly-structured object
  - SRP: Each clause has a ClauseParser, and ClausesParser takes in a ClauseParser and applies it repeatedly [Strategy Pattern]
- `SemanticValidator`: maps a given `(declarations, UntypedQuery)` object into `Query | SemanticError`
  - Checks for semantic problems
  - Simple type inference on `UntypedQuery` and produce `Query` with meaningful semantics
  - SRP: Each clause has a Validator

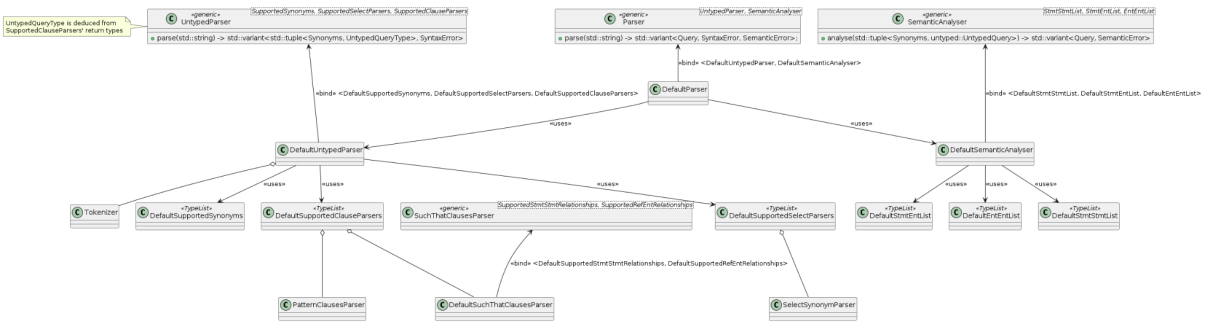


Fig 8. Class Diagram of QPS Parser

For a bigger version of this diagram, click here

Polymorphic behavior is achieved via *parametric polymorphism* instead of the more typical *subtype polymorphism*. Refer to 2.3.4. Design Decisions for our justification.

2.3.1.2 Evaluating Query Objects

A `QueryEvaluator` receives a given `Query` object and has each `Clause` evaluated separately by instantiating a specialized `ClauseEvaluator` before returning a joined result. For even more SRP, each relationship has its own evaluator.

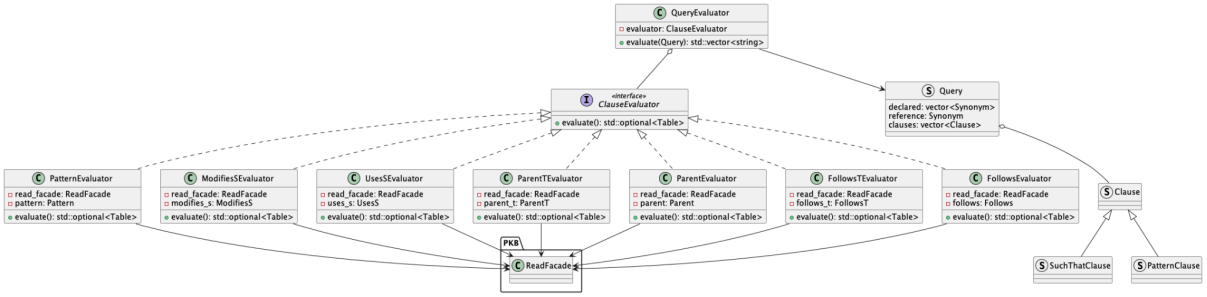


Fig 10. Class Diagram of QPS Evaluator

2.3.2. Sequence Diagram

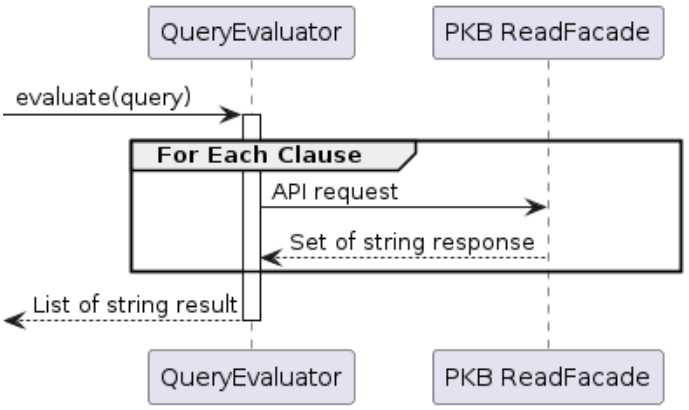


Fig. 9 High-level sequence diagram for evaluating a query

For more sequence diagrams, please refer to our website

2.3.3. List of APIs

All the public APIs of QPS components can be found in Appendix E.

2.3.4. Design Decisions

Decision 1: Generic Parser or Concrete Parser

Alternatives	Extensibility (OCP)	Testability	Ease of Implementation
Generic Parser [Strategy Pattern]	High. Behaviours can be added without changing code.	High. Different behaviour permutations can be tested at any time without changing code.	Moderate. While generic programming is hard, the provided type safety can help with correctness guarantees
Concrete Parser + Inheritance	Low. Adding behaviours requires changing code.	Low. Different permutations require changing code to test separately.	High. Working with pointers is much easier, at the cost of type-safety and compiler-based correctness guarantees

Decision and Justification

We chose option 1 to adhere strongly to OCP and have a highly configurable, extensible parser. To support new select clauses, we only need to write a new select clause parser and add it to the `SupportedSelectParsers` default `TypeList`. For thorough testing, we can easily pass in different configurations for different combinations of the parser.

Decision 2: Encapsulation of behaviours or Separation of behaviours for entities

Alternatives	Extensibility (Addition of types)	Extensibility (Addition of behaviours)
Encapsulation of behaviours via Inheritance	High. Adding new types is a matter of extending the base class, provided the type has similar behaviours.	Low. Inheritance hierarchy likely requires modification when adding new behaviours or types that require new behaviours.
Separation of behaviours via Visitor Pattern	Low. The Visitor interface needs to be modified, as well as all concrete visitors.	Low. A new Visitor interface likely need to be implemented, and all Visitables must be modified to be able to accept the new Visitor
Separation of behaviours via `std::variant`	Moderate. Adding new types is a matter of creating a new class, and adding to the variant.	High. No Visitables nor Visitor interfaces need to be modified. The new concrete visitor can be created at point of usage at any time.

Decision and Justification

We chose to separate behaviours from data for relationships such as `Follows` and `Pattern`, using `std::variant` for easier addition of types whose behaviour may differ from currently known types, e.g. a relationship with 3 members.

2.3.5. Design Principles and Patterns

Design Principles

- Single Responsibility Principle (SRP)
  - Each component of `Parser` is responsible for only 1 thing e.g. `UntypedParser` builds objects from strings and `SemanticAnalyser` assigns meaning to the objects. Each sub-component is itself responsible for only one thing.
  - Each component of `QueryEvaluator` is responsible for only 1 thing e.g. `FollowsEvaluator` evaluates only the `Follows` relation, and each subcomponent of the `FollowsEvaluator` evaluates one possible instantiation of the `Follows` relation.
- Open Close Principle (OCP)
  - Each `TypeList` in `Parser` is a point of customisation: to implement new behaviours, simply write a new class which obeys the interface and add it to the `TypeList`. No change to existing code required.
  - Existing evaluator logic does not need to be changed if new clause types are added, extension is achieved simply by creating new child classes of `ClauseEvaluator`.
- Dependency Inversion Principle (DIP)
  - Fulfilled automatically via generic programming; each function template / class template specifies what it needs, and each class that attempts to bind to the template is forced to provide what the function / class template needs.
- Interface Segregation Principle (ISP)
  - Fulfilled automatically via generic programming; no function templates / class templates are forced to depend on interface it doesn't use, since the function or class templates themselves define what they need and therefore require the templates to have.
- Don't Repeat Yourself (DRY)
  - Similar logic is extracted out as functions e.g. `parse_stmt_stmt()` is recursively called on all elements of `SupportedStmtStmtRelationships`.
  - Tokeniser is shared between SP and QPS.

Design Patterns

- Visitor Pattern
  - The `TypeList` is a union of types. As long as the visitor works on the newly created type, it can be added to the `TypeList` with no change to other code.

    - `Relationship` is a tagged union of relationships. New relationships can be added to the variant. Only functions that need to act on the new relationship (e.g. its evaluator) will need to support the new relationship.
- Chain of Responsibility Pattern
    - In `SemanticAnalyser`, each element of `StmtStmtList` will attempt to validate a given `UntypedStmtStmtRel`. If an element fails to validate it, the object is passed on to the next element on the list. If no elements are able to type-validate the object, then a `SemanticError` is returned.

3. Use of AI code generators

NIL

4. Test Strategy

4.1. Test Strategy

4.1.1. Unit Testing

Each developer who created a component would also write the unit tests cases for that specific component. We adopt a white box testing approach where we test and verify all paths of the code.

4.1.2. Integration Testing

A pair of developers from SP + PKB or QPS + PKB will develop tests for interaction between those two components, and finally, all the developers contribute to a single end-to-end test.

4.1.3. System Testing

In our project, The system testing is done as a black box testing. The three main categories of tests include

- **Invalid Syntax and Semantics Tests** - Ensure thorough catching and differentiation of all invalid syntax and semantics cases.
- **Single Clause Tests** - Ensure single clause queries are successfully working, from SP parsing, to PKB storing and retrieval, to QPS evaluation.
- **Multiple Clauses Tests** - Ensure different combinations of multiple clauses queries are successfully working, including matching of all conditions by QPS.

4.2. Test Statistics

Type of Test	Purpose	Quantity
Unit-test coverage	Cover all execution paths of individual components	1708 assertions in 84 test cases
Integration-test coverage	Verifies different components work well together	124 assertions in 5 test cases
System-test coverage	Access the entire application's adherence to requirements	349 test cases in 20 test files

For a sample of our code coverage report, refer to Appendix F.

4.3. Automation Approach

We utilized the CI (Continuous Integration) by GitHub Actions to automatically build, test, and verify code changes. A few Python scripts are also written to automatically run the system test cases.

4.4. Bug Handing Approach

Bugs are reported on Notion including description, the specific test cases, differences between expectations and results. After identifying the root problems, we create GitHub issues and assign them to the relevant developers.

After fixing the bugs, regression testing is done by ensuring it passes all the unit, integration and system testing before merging the updated code to the main branch.

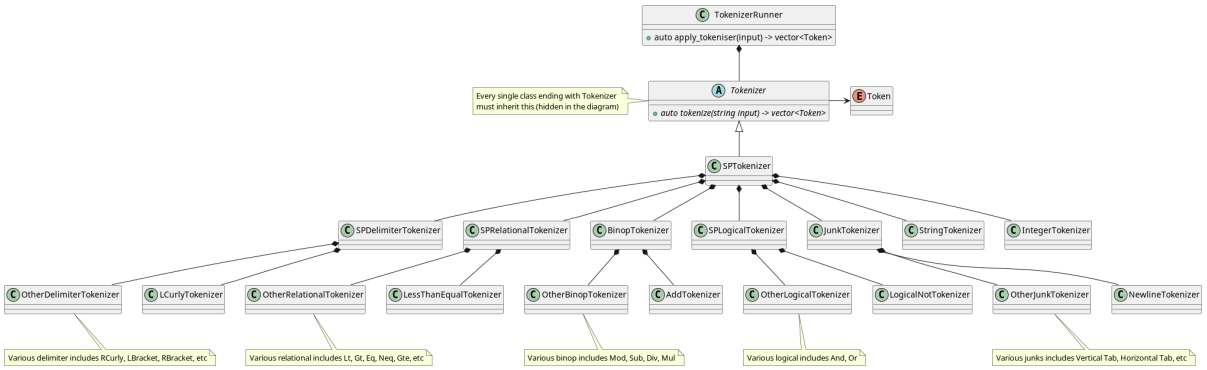
5. Plan for Milestone 2

We aim to complete all Advanced SPA features by Milestone 2. For a Gantt chart for our plan, refer to Appendix G

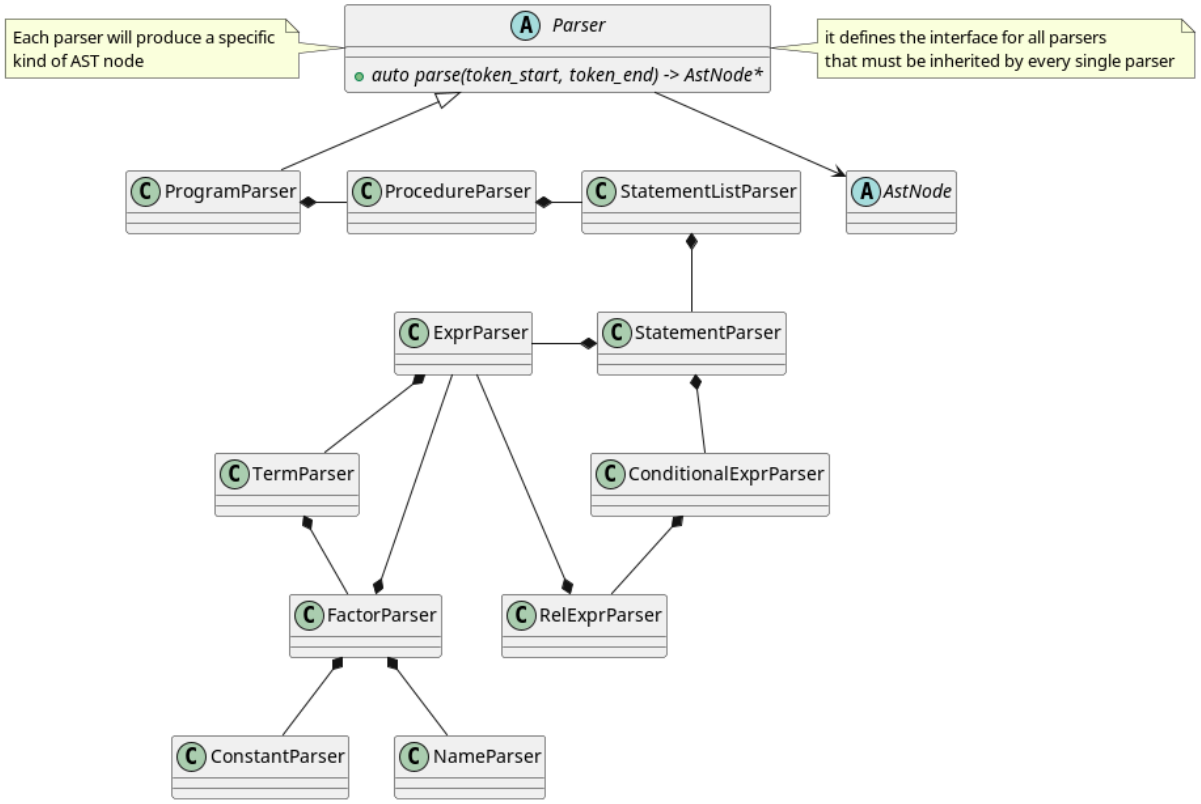
Appendix A: UML Diagrams of Each SP Component

Class Diagrams

1. Class Diagram of SP Tokenizer (Strategy Pattern)

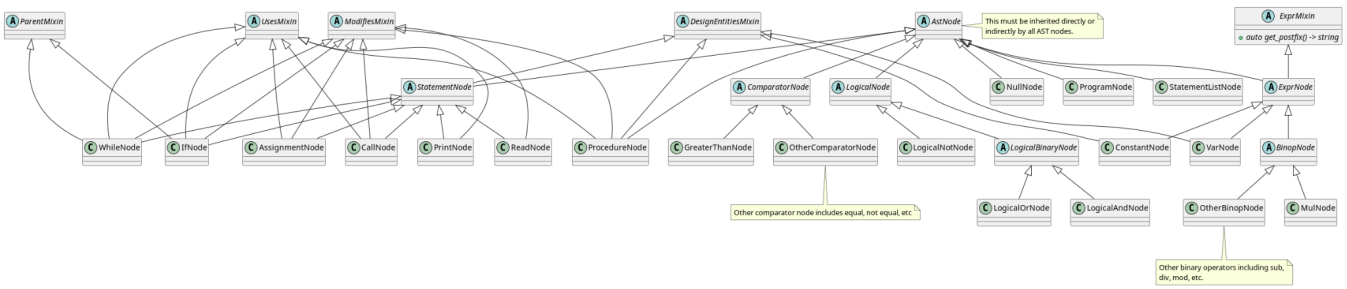


2. Class Diagram of SP Parser



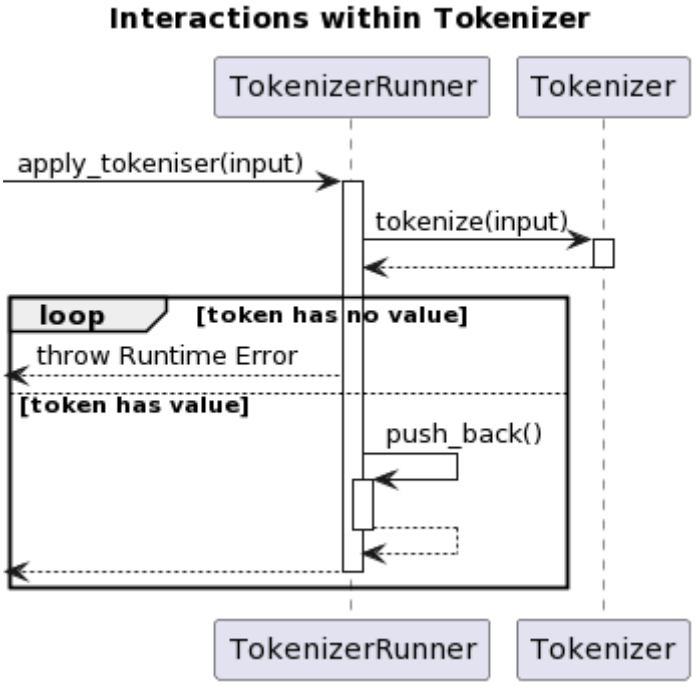
x

3. Class Diagram of SP AST Node with various Mixin Classes

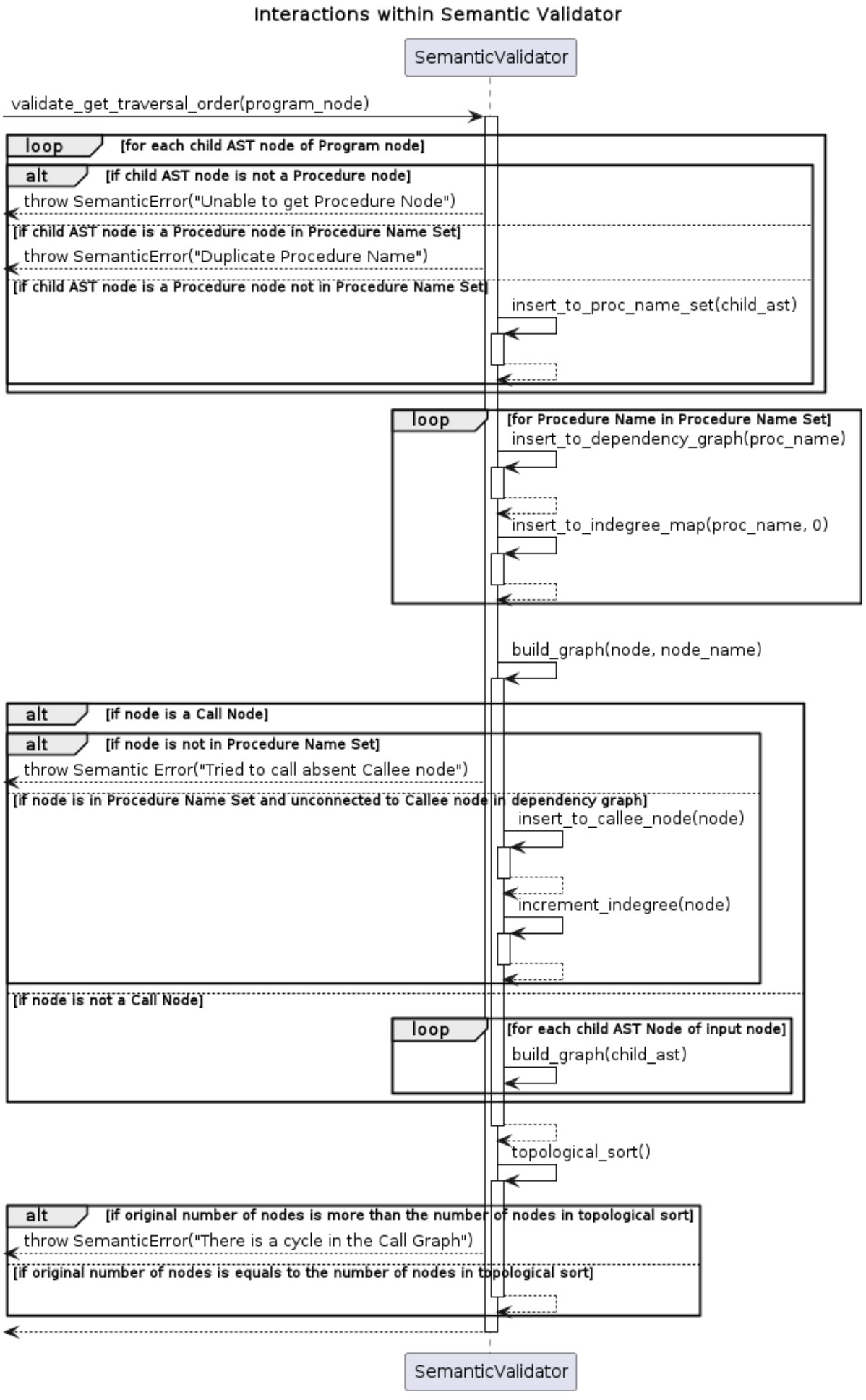


Sequence Diagrams

1. Sequence Diagram of Tokenizer, a subcomponent within SP



2. Sequence Diagram of Semantic Validator, a subcomponent within SP

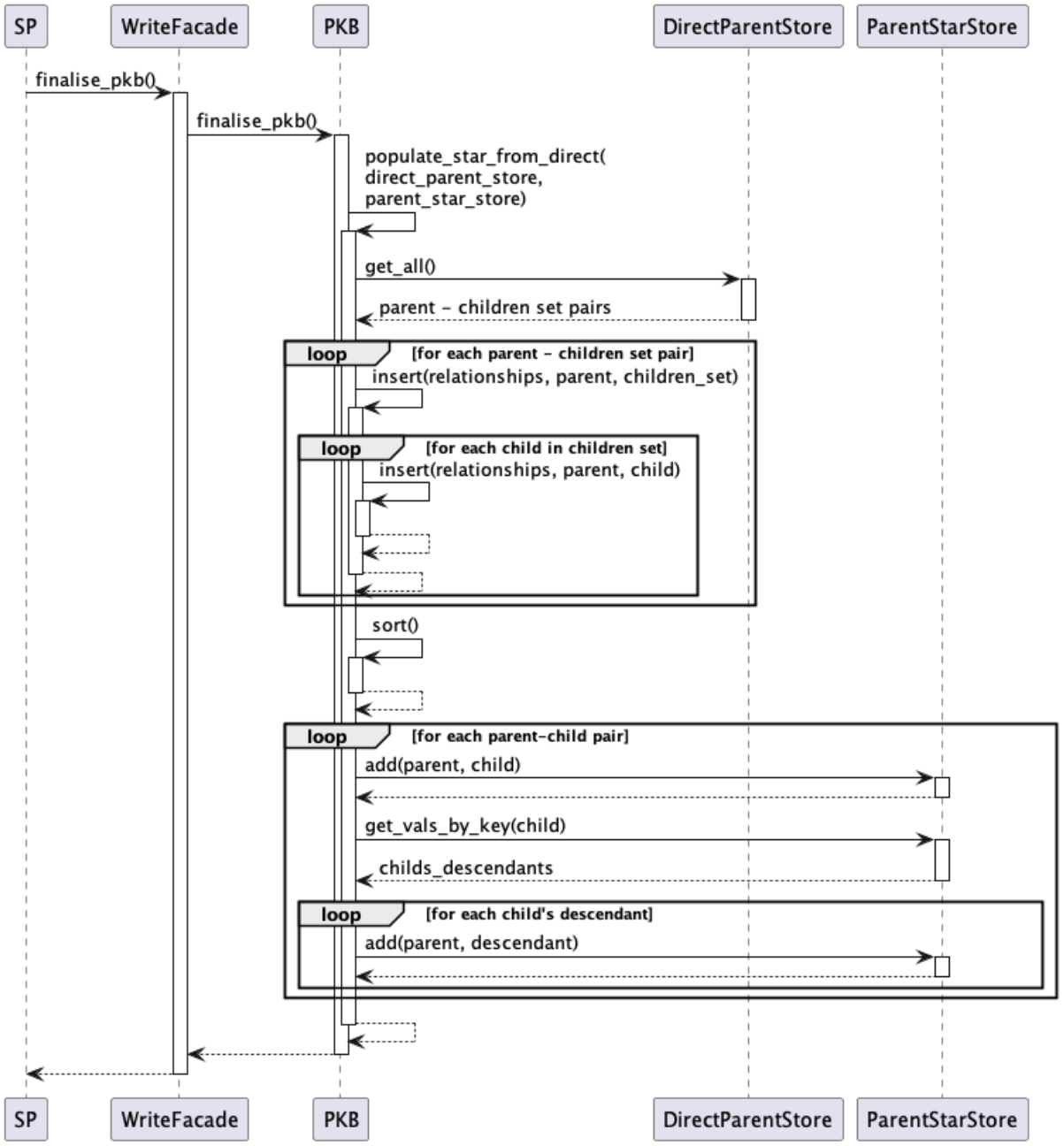

Interactions within Semantic Validator
SemanticValidator
validate_get_traversal_order(program_node)
loop
[for each child AST node of Program node]
alt
[if child AST node is not a Procedure node]
throw SemanticError("Unable to get Procedure Node")
[if child AST node is a Procedure node in Procedure Name Set]
throw SemanticError("Duplicate Procedure Name")
[if child AST node is a Procedure node not in Procedure Name Set]
insert_to_proc_name_set(child_ast)
loop
[for Procedure Name in Procedure Name Set]
insert_to_dependency_graph(proc_name)
insert_to_indegree_map(proc_name, 0)
build_graph(node, node_name)
alt
[if node is a Call Node]
alt
[if node is not in Procedure Name Set]
throw Semantic Error("Tried to call absent Callee node")
[if node is in Procedure Name Set and unconnected to Callee node in dependency graph]
insert_to_callee_node(node)
increment_indegree(node)
[if node is not a Call Node]
loop
[for each child AST Node of input node]
build_graph(child_ast)
topological_sort()
alt
[if original number of nodes is more than the number of nodes in topological sort]
throw SemanticError("There is a cycle in the Call Graph")
[if original number of nodes is equals to the number of nodes in topological sort]
SemanticValidator

Appendix B: Full SP API List

SourceProcessor APIs
std::shared_ptr<AstNode> process(STRING input) Parses the input SIMPLE source program, and adds discovered Design Abstractions to PKB. Returns the root Program ASTNode. This is done so the AST can be easily tested in unit and integration testing.
TokenizerRunner APIs
std::vector<Token> apply_tokeniser(std::string input, bool debug) Tokenises the input SIMPLE source program by matching every character/phrase to its representative token type and create TOKENS. Throws a Runtime Error when any invalid character is tokenised.
Parser APIs
std::shared_ptr<AstNode> parse(Iterator& token_start, const Iterator& token_end) Parses the input tokens into an AST in a recursive descent manner and returns a pointer to the root Program ASTNode.
Semantic Validator APIs
std::vector<std::string> validate_get_traversal_order(const std::shared_ptr<AstNode>& program_node) Validate the semantic of AST according to various SIMPLE semantic specifications, and return the topological ordering of all the procedures to be traversed.
Traverser APIs
std::shared_ptr<AstNode> traverse(std::shared_ptr<AstNode> node, std::vector<std::string> procedure_topological_sort) Traverses down the AST from the input node and uses various Traverser classes to extract Design Entities and Abstractions. Adds the extracted Design Abstractions to the PKB. Returns the root Program ASTNode.

Appendix C: Sequence Diagrams for PKB

1. Sequence Diagram for populating transitive relationship of the ParentStarStore



Appendix D: Full PKB API List

Entity-related Write Operations

void add_procedure(std::string procedure);
void add_variable(std::string variable);
void add_constant(std::string constant);

Entity-related Read Operations

std::unordered_set<std::string> get_entities();
std::unordered_set<std::string> get_procedures();
std::unordered_set<std::string> get_variables();
std::unordered_set<std::string> get_constants();

Statement-related Write Operations
void add_statement(const std::string& statement_number, StatementType type);
Statement-related Read Operations

std::unordered_set<std::string> get_all_statements();
std::unordered_set<std::string> get_assign_statements();
std::unordered_set<std::string> get_if_statements();
std::unordered_set<std::string> get_while_statements();
std::unordered_set<std::string> get_read_statements();
std::unordered_set<std::string> get_print_statements();
std::unordered_set<std::string> get_call_statements();

Modifies-related Write Operations
void add_statement_modifies_var(const std::string& statement_number, std::string variable); void add_procedure_modifies_var(std::string procedure, std::string variable);
Modifies-related Read Operations

std::unordered_set<std::string> get_vars_modified_by_statement(const std::string& s);
std::unordered_set<std::string> get_statements_that_modify_var(const std::string& variable);
std::unordered_set<std::string> get_statements_that_modify_var(const std::string& variable, const StatementType& statement_type);
bool does_statement_modify_var(const std::string& statement_number, const std::string& variable);
std::unordered_set<std::string> get_vars_modified_by_procedure(const std::string& procedure);
std::unordered_set<std::string> get_procedures_that_modify_var(const std::string& variable);
bool does_procedure_modify_var(const std::string& procedure, const std::string& variable);
std::unordered_set<std::string> get_all_statements_that_modify();
std::unordered_set<std::string> get_all_statements_that_modify(const StatementType& statement_type);
bool does_statement_modify_any_var(const std::string& statement_number);
std::unordered_set<std::tuple<std::string, std::string>> get_all_statements_and_var_modify_pairs();
std::unordered_set<std::tuple<std::string, std::string>> get_all_statements_and_var_modify_pairs(const StatementType& statement_type);
std::unordered_set<std::string> get_all_procedures_that_modify();
bool does_procedure_modify_any_var(const std::string& procedure);
std::unordered_set<std::tuple<std::string, std::string>> get_all_procedures_and_var_modify_pairs();

Uses-related Write Operations

void add_statement_uses_var(const std::string& statement_number, std::string variable);
void add_procedure_uses_var(std::string procedure, std::string variable);

Uses-related Read Operations

std::unordered_set<std::string> get_vars_used_by_statement(const std::string& s);
std::unordered_set<std::string> get_statements_that_use_var(const std::string& variable);
std::unordered_set<std::string> get_statements_that_use_var(const std::string& variable, const StatementType& statement_type);
bool does_statement_use_var(const std::string& statement_number, const std::string& variable);
std::unordered_set<std::string> get_vars_used_by_procedure(const std::string& procedure);
std::unordered_set<std::string> get_procedures_that_use_var(const std::string& variable);
bool does_procedure_use_var(const std::string& procedure, const std::string& variable);
std::unordered_set<std::string> get_all_statements_that_use();
std::unordered_set<std::string> get_all_statements_that_use(const StatementType& statement_type);
bool does_statement_use_any_var(const std::string& statement_number);
std::unordered_set<std::tuple<std::string, std::string>> get_all_statements_and_var_use_pairs();
std::unordered_set<std::tuple<std::string, std::string>> get_all_statements_and_var_use_pairs(const StatementType& statement_type);
std::unordered_set<std::string> get_all_procedures_that_use();
bool does_procedure_use_any_var(const std::string& procedure);
std::unordered_set<std::tuple<std::string, std::string>> get_all_procedures_and_var_use_pairs();

Follows-related Write Operations

void add_follows(const std::string& stmt1, const std::string& stmt2);

Follows-related Read Operations

bool has_follows_relation(const std::string& stmt1, const std::string& stmt2) const;
std::unordered_map<std::string, std::string> get_all_follows() const;
std::unordered_set<std::string> get_all_follows_keys() const;
std::unordered_set<std::string> get_all_follows_keys(const StatementType& statement_type) const;
std::unordered_set<std::string> get_all_follows_values() const;
std::unordered_set<std::string> get_all_follows_values(const StatementType& statement_type) const;
std::string get_statement_following(const std::string& s) const;
std::string get_statement_following(const std::string& s, const StatementType& statement_type) const;
std::string get_statement_followed_by(const std::string& s) const;
std::string get_statement_followed_by(const std::string& s, const StatementType& statement_type) const;
bool has_follows_star_relation(const std::string& stmt1, const std::string& stmt2) const;
std::unordered_map<std::string, std::unordered_set<std::string>> get_all_follows_star() const;
std::unordered_set<std::string> get_all_follows_star_keys() const;

std::unordered_set<std::string> get_all_follows_star_keys(const StatementType& statement_type) const;

std::unordered_set<std::string> get_all_follows_star_values() const;

std::unordered_set<std::string> get_all_follows_star_values(const StatementType& statement_type) const;

std::unordered_set<std::string> get_follows_stars_following(const std::string& stmt) const;

std::unordered_set<std::string> get_follows_stars_following(const std::string& stmt, const StatementType& statement_type) const;

std::unordered_set<std::string> get_follows_stars_by(const std::string& stmt) const;

std::unordered_set<std::string> get_follows_stars_by(const std::string& stmt, const StatementType& statement_type) const;

Parent-related Write Operations

void add_parent(const std::string& parent, const std::string& child);

Parent-related Read Operations

bool has_parent_relation(const std::string& parent, const std::string& child) const;

std::unordered_map<std::string, std::unordered_set<std::string>> get_all_parent() const;

std::unordered_set<std::string> get_all_parent_keys() const;

std::unordered_set<std::string> get_all_parent_keys(const StatementType& statement_type) const;

std::unordered_set<std::string> get_all_parent_values() const;

std::unordered_set<std::string> get_all_parent_values(const StatementType& statement_type) const;

std::unordered_set<std::string> get_children_of(const std::string& parent) const;

std::unordered_set<std::string> get_children_of(const std::string& parent, const StatementType& statement_type) const;

std::string get_parent_of(const std::string& child) const;

std::string get_parent_of(const std::string& child, const StatementType& statement_type) const;

bool has_parent_star_relation(const std::string& parent, const std::string& child) const;

std::unordered_map<std::string, std::unordered_set<std::string>> get_all_parent_star() const;

std::unordered_set<std::string> get_all_parent_star_keys() const;

std::unordered_set<std::string> get_all_parent_star_keys(const StatementType& statement_type) const;

std::unordered_set<std::string> get_all_parent_star_values() const;

std::unordered_set<std::string> get_all_parent_star_values(const StatementType& statement_type) const;

std::unordered_set<std::string> get_children_star_of(const std::string& parent) const;

std::unordered_set<std::string> get_children_star_of(const std::string& parent, const StatementType& statement_type) const;

std::unordered_set<std::string> get_parent_star_of(const std::string& child) const;

std::unordered_set<std::string> get_parent_star_of(const std::string& child, const StatementType& statement_type) const;

Pattern (assignment)-related Write Operations

std::unordered_set<std::string> get_all_assignments_rhs(const std::string& rhs);
std::unordered_set<std::string> get_all_assignments_rhs_partial(const std::string& rhs);
std::unordered_set<std::string> get_all_assignments_lhs(const std::string& lhs);
std::unordered_set<std::string> get_all_assignments_lhs_rhs(const std::string& lhs, const std::string& rhs);
std::unordered_set<std::string> get_all_assignments_lhs_rhs_partial(const std::string& lhs, const std::string& rhs);

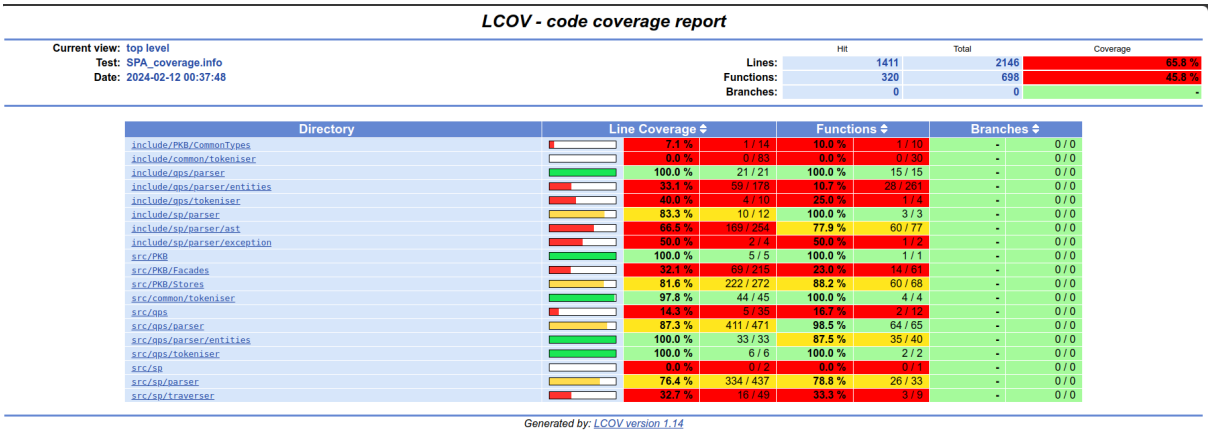
Pre-computation of Parent* and Follows* Relations
void finalise_pkb();

Appendix E: Full QPS API List

Parser APIs
std::variant<Query, SyntaxError, SemanticError> parse(std::string input) Parses the input PQL query and constructs the corresponding Query object if possible. Returns one of Query, SyntaxError, or SemanticError.

QueryEvaluator APIs
std::vector<string> evaluate(Query query) Evaluate the query object. Returns a vector of strings corresponding to the query evaluation results.

Appendix F: Sample Code Coverage Report



LCOV - code coverage report

Current view: top level
Test: SPA_coverage.info
Date: 2024-02-12 00:37:48

	Hit	Total	Coverage
Lines:	1411	2146	65.8 %
Functions:	320	698	45.8 %
Branches:	0	0	-

Directory	Line Coverage		Functions		Branches	
include/PKB/CommonTypes	7.1 %	1 / 14	10.0 %	1 / 10	-	0 / 0
include/common/tokeniser	0.0 %	0 / 83	0.0 %	0 / 30	-	0 / 0
include/qps/parser	100.0 %	21 / 21	100.0 %	15 / 15	-	0 / 0
include/qps/parser/entities	33.1 %	59 / 178	10.7 %	28 / 261	-	0 / 0
include/qps/tokeniser	40.0 %	4 / 10	25.0 %	1 / 4	-	0 / 0
include/sp/parser	83.3 %	10 / 12	100.0 %	3 / 3	-	0 / 0
include/sp/parser/ast	66.5 %	169 / 254	77.9 %	60 / 77	-	0 / 0
include/sp/parser/exception	50.0 %	2 / 4	50.0 %	1 / 2	-	0 / 0
src/PKB	100.0 %	5 / 5	100.0 %	1 / 1	-	0 / 0
src/PKB/Facades	32.1 %	69 / 215	23.0 %	14 / 61	-	0 / 0
src/PKB/Stores	81.6 %	222 / 272	88.2 %	60 / 68	-	0 / 0
src/common/tokeniser	97.8 %	44 / 45	100.0 %	4 / 4	-	0 / 0
src/qps	14.3 %	5 / 35	16.7 %	2 / 12	-	0 / 0
src/qps/parser	87.3 %	411 / 471	98.5 %	64 / 65	-	0 / 0
src/qps/parser/entities	100.0 %	33 / 33	87.5 %	35 / 40	-	0 / 0
src/qps/tokeniser	100.0 %	6 / 6	100.0 %	2 / 2	-	0 / 0
src/sp	0.0 %	0 / 2	0.0 %	0 / 1	-	0 / 0
src/sp/parser	76.4 %	334 / 437	78.8 %	26 / 33	-	0 / 0
src/sp/traverser	32.7 %	16 / 49	33.3 %	3 / 9	-	0 / 0

Generated by: LCOV version 1.14

Appendix G: Gantt Chart for Milestone 2