CS3203 Software Engineering Project

AY22/23 Semester 1

Project Report – System Overview

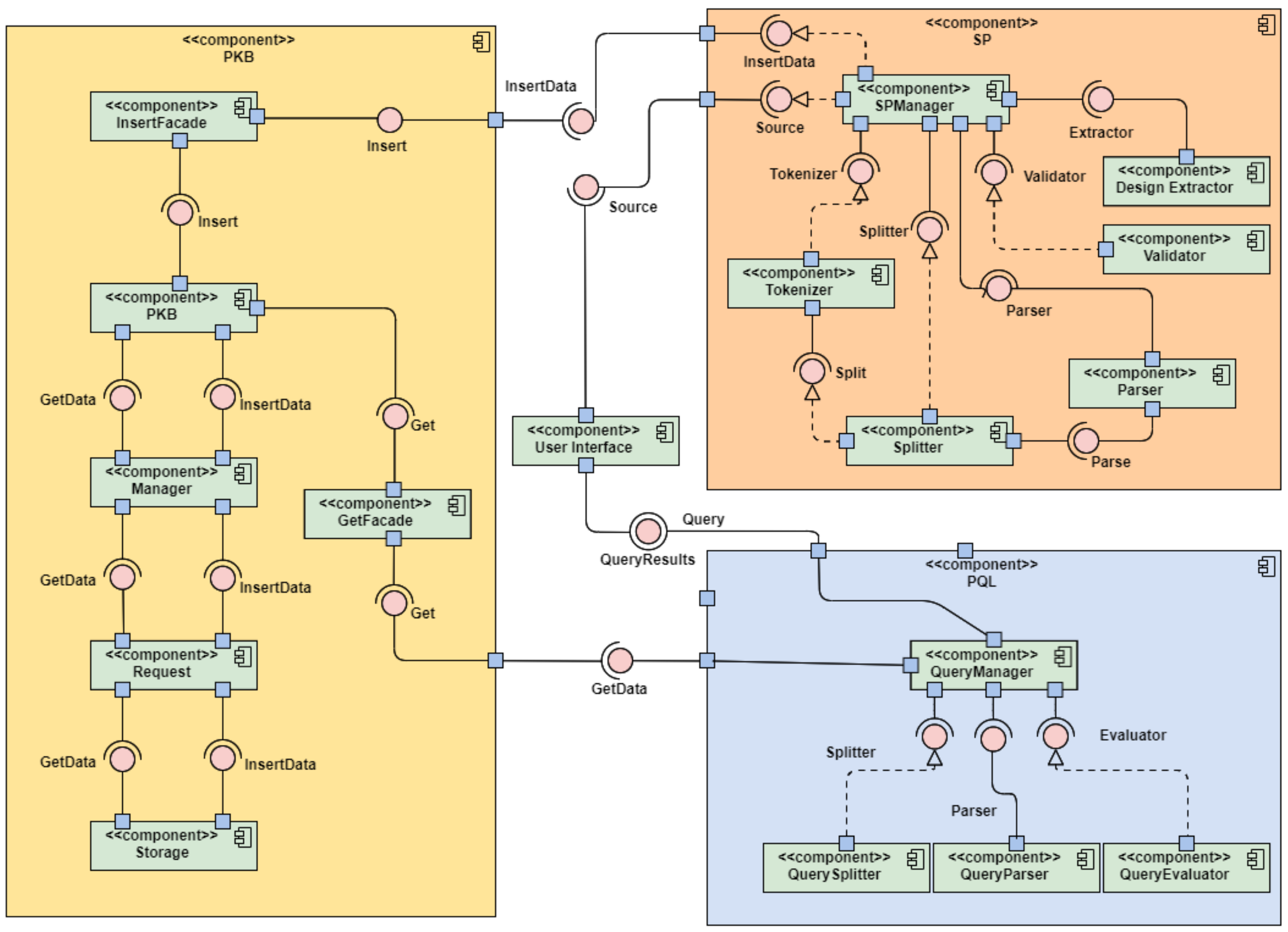
Team 19

Team Members	Student No.	Email
Daniel Tan Ren Jie	A0199712U	danielt@u.nus.edu
Ng Qian Jie Cheryl	A0189689W	ng.cheryl@u.nus.edu
Ang Kai Chao	A0206050Y	e0425973@u.nus.edu
Nicholas Yeo Yock Leng	A0199624N	e0406605@u.nus.edu
Amy Lim Zhi Ting	A0207959M	e0439265@u.nus.edu
Irfan Danial Bin Rahmat	A0218303R	e0544339@u.nus.edu

Consultation Hours: Tuesday 6pm

Tutor: Ramachandran Sandhya

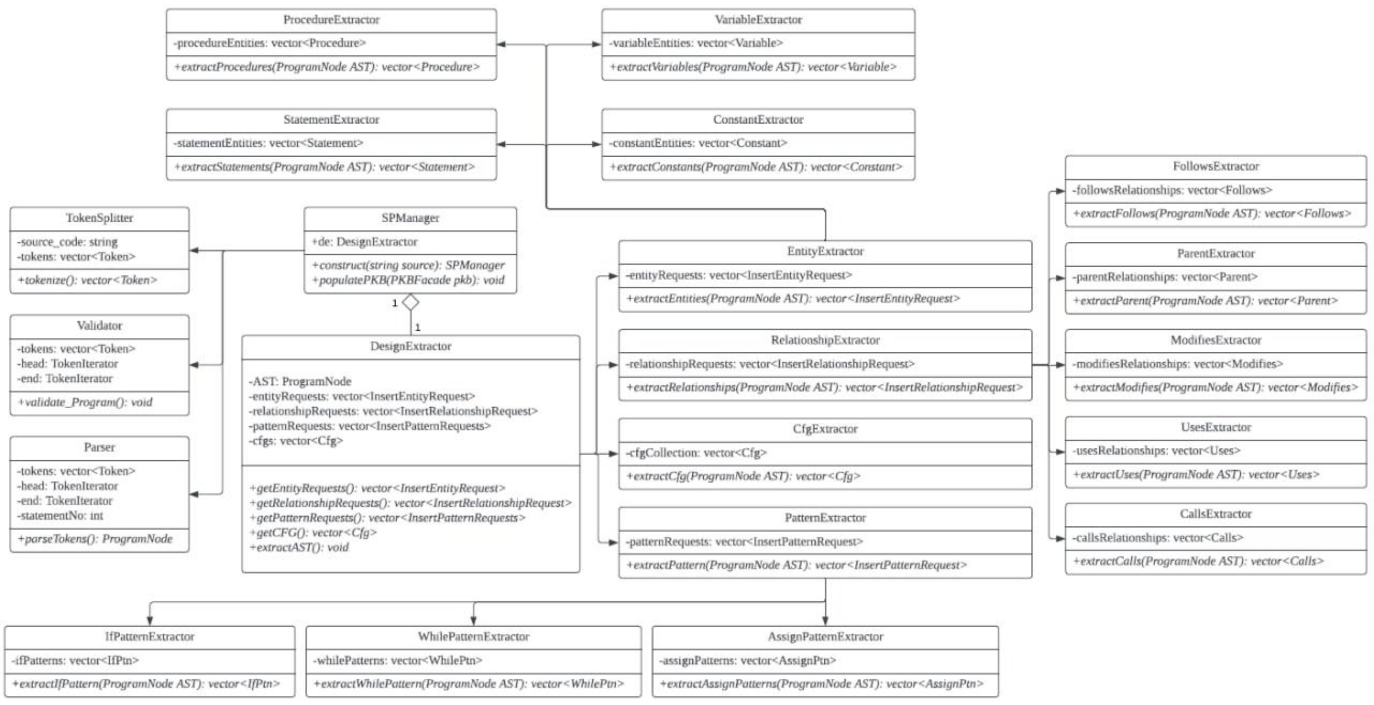
1. System Architecture



2. Component Architecture and Design

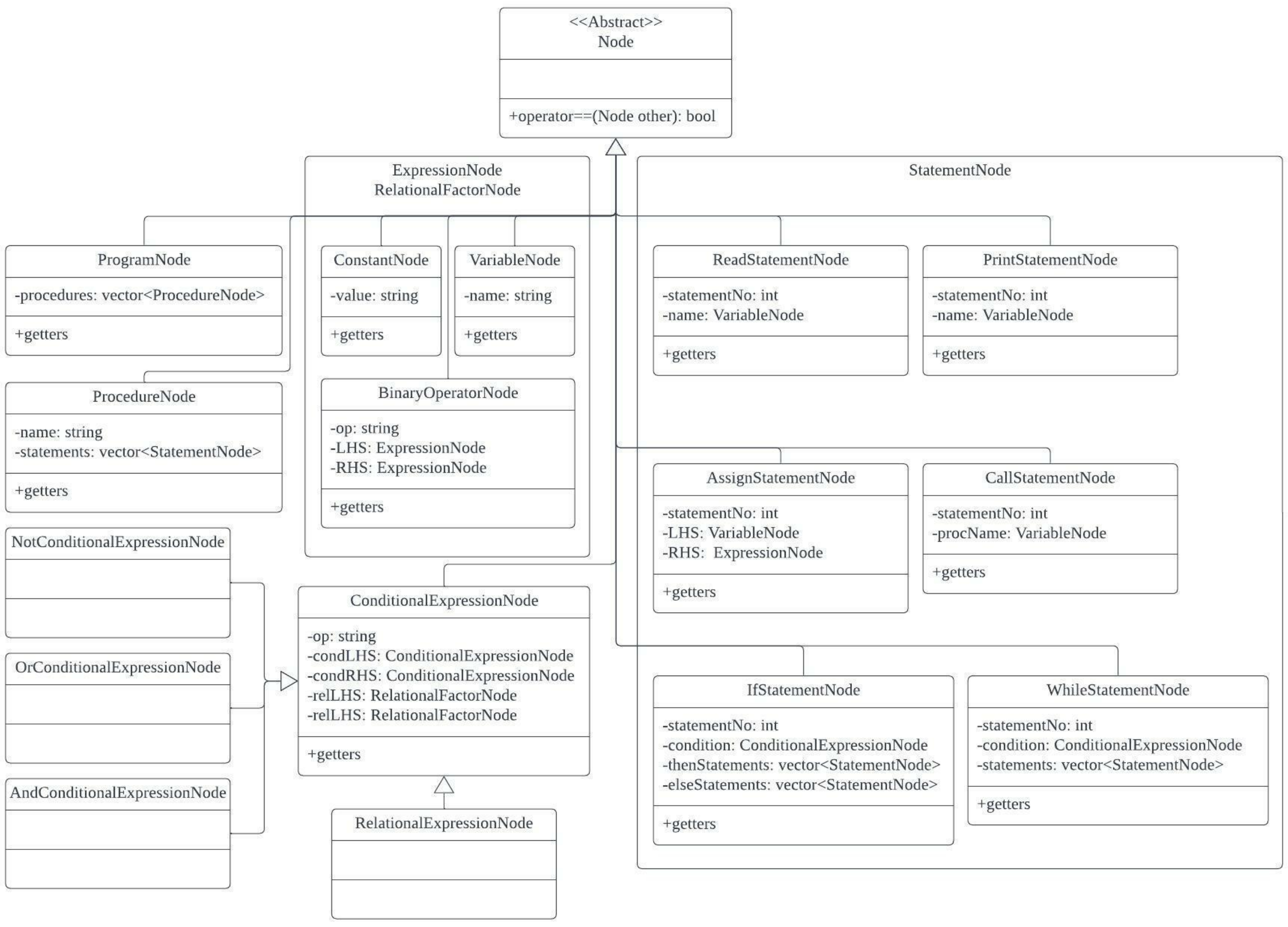
2.1 SP

Class Diagram for SPManager

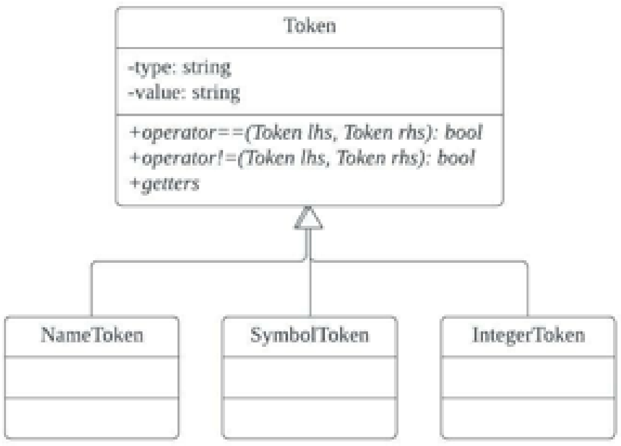


ProcedureExtractor
-procedureEntities: vector<Procedure>
+extractProcedures(ProgramNode AST): vector<Procedure>
VariableExtractor
-variableEntities: vector<Variable>
+extractVariables(ProgramNode AST): vector<Variable>
StatementExtractor
-statementEntities: vector<Statement>
+extractStatements(ProgramNode AST): vector<Statement>
ConstantExtractor
-constantEntities: vector<Constant>
+extractConstants(ProgramNode AST): vector<Constant>
FollowsExtractor
-followsRelationships: vector<Follows>
+extractFollows(ProgramNode AST): vector<Follows>
TokenSplitter
-source_code: string
-tokens: vector<Token>
+tokenize(): vector<Token>
SPManager
+de: DesignExtractor
+construct(string source): SPManager
+populatePKB(PKBFacade pkb): void
EntityExtractor
-entityRequests: vector<InsertEntityRequest>
+extractEntities(ProgramNode AST): vector<InsertEntityRequest>
ParentExtractor
-parentRelationships: vector<Parent>
+extractParent(ProgramNode AST): vector<Parent>
1
1
Validator
-tokens: vector<Token>
-head: TokenIterator
-end: TokenIterator
+validate_Program(): void
DesignExtractor
-AST: ProgramNode
-entityRequests: vector<InsertEntityRequest>
-relationshipRequests: vector<InsertRelationshipRequest>
-patternRequests: vector<InsertPatternRequests>
-cfgs: vector<Cfg>
+getEntityRequests(): vector<InsertEntityRequest>
+getRelationshipRequests(): vector<InsertRelationshipRequest>
+getPatternRequests(): vector<InsertPatternRequests>
+getCFG(): vector<Cfg>
+extractAST(): void
RelationshipExtractor
-relationshipRequests: vector<InsertRelationshipRequest>
+extractRelationships(ProgramNode AST): vector<InsertRelationshipRequest>
ModifiesExtractor
-modifiesRelationships: vector<Modifies>
+extractModifies(ProgramNode AST): vector<Modifies>
Parser
-tokens: vector<Token>
-head: TokenIterator
-end: TokenIterator
-statementNo: int
+parseTokens(): ProgramNode
CfgExtractor
-cfgCollection: vector<Cfg>
+extractCfg(ProgramNode AST): vector<Cfg>
UsesExtractor
-usesRelationships: vector<Uses>
+extractUses(ProgramNode AST): vector<Uses>
PatternExtractor
-patternRequests: vector<InsertPatternRequest>
+extractPattern(ProgramNode AST): vector<InsertPatternRequest>
CallsExtractor
-callsRelationships: vector<Calls>
+extractCalls(ProgramNode AST): vector<Calls>
IfPatternExtractor
-ifPatterns: vector<IfPtn>
+extractIfPattern(ProgramNode AST): vector<IfPtn>
WhilePatternExtractor
-whilePatterns: vector<WhilePtn>
+extractWhilePattern(ProgramNode AST): vector<WhilePtn>
AssignPatternExtractor
-assignPatterns: vector<AssignPtn>
+extractAssignPatterns(ProgramNode AST): vector<AssignPtn>

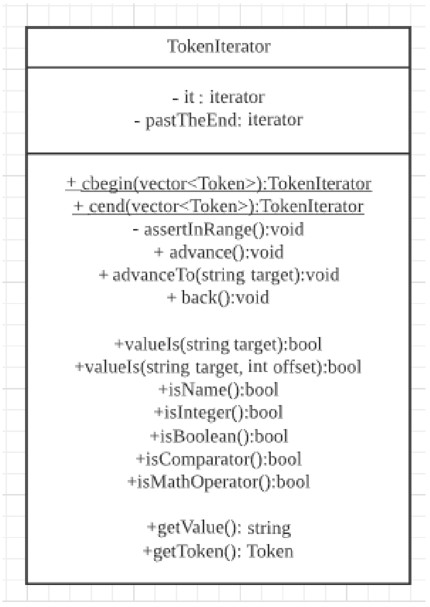
Class Diagram for AST



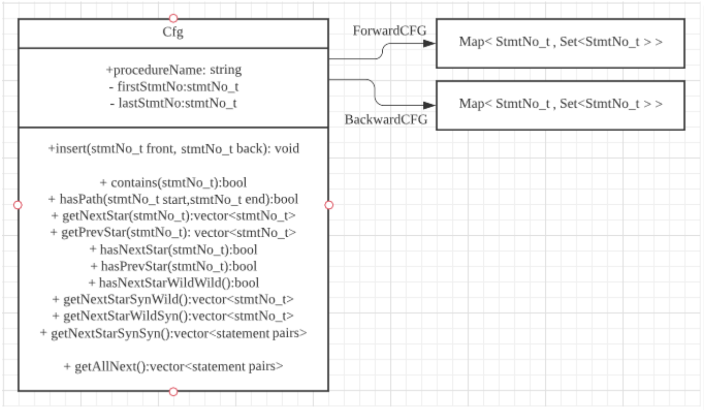
Class Diagram for Token


Token
-type: string
-value: string
+operator==(Token lhs, Token rhs): bool
+operator!=(Token lhs, Token rhs): bool
+getters
NameToken
SymbolToken
IntegerToken

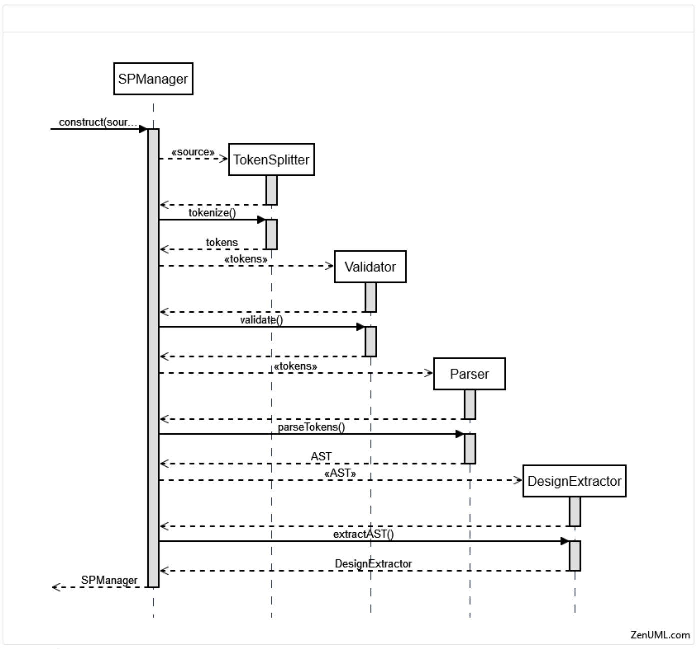
Class Diagram for TokenIterator

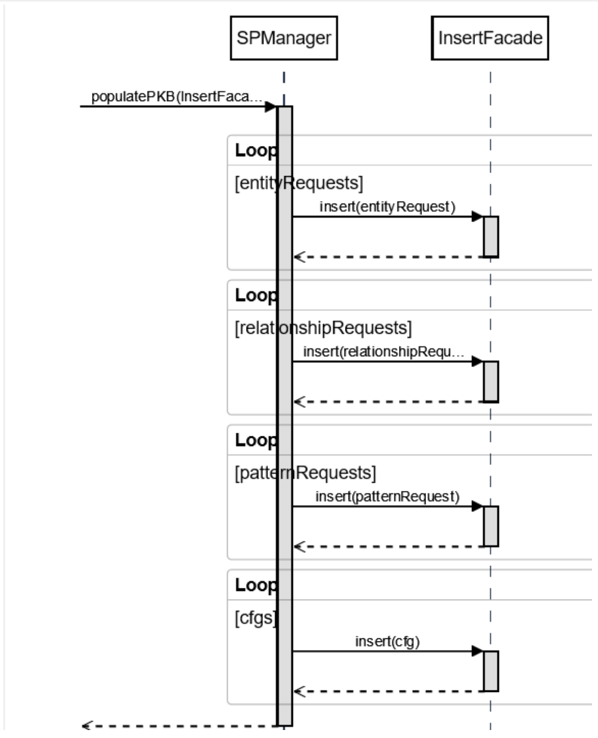


Class Diagram for CFG


Cfg
+procedureName: string
- firstStmtNo:stmtNo_t
- lastStmtNo:stmtNo_t
+insert(stmtNo_t front, stmtNo_t back): void
+ contains(stmtNo_t):bool
+ hasPath(stmtNo_t start,stmtNo_t end):bool
+ getNextStar(stmtNo_t):vector<stmtNo_t>
+ getPrevStar(stmtNo_t): vector<stmtNo_t>
+ hasNextStar(stmtNo_t):bool
+ hasPrevStar(stmtNo_t):bool
+ hasNextStarWildWild():bool
+ getNextStarSynWild():vector<stmtNo_t>
+ getNextStarWildSyn():vector<stmtNo_t>
+ getNextStarSynSyn():vector<statement pairs>
+ getAllNext():vector<statement pairs>
ForwardCFG
BackwardCFG
Map< StmtNo_t , Set<StmtNo_t > >
Map< StmtNo_t , Set<StmtNo_t > >

Sequence Diagram for SPManager





The Source Processor Manager exposes 2 external interfaces, the static construct method and the populatePKB method. The static construct method encapsulates the different steps in the pipeline of extracting information from the SIMPLE code, ie tokenizing, validation, parsing and extraction. The populate PKB method encapsulates the insertion of data into the PKB by calling the appropriate PKB API methods.

SP Design Decisions

Design Principles

1. *Single Responsibility Principle*

Source Processor is broken down into several subcomponents.
TokenSplitter handles the tokenizing of the SIMPLE source code. Validator handles the validation of statement structures and checks for any syntax errors. Parser parses the tokens into an AST. Design Extractor traverses the AST to extract out the required entities and abstractions.

2. *Separation of concerns*

The Design Extractor calls multiple different specific Extractors to traverse the AST and extract their specific data. This allows for each extractor to be developed independently and use designs which are more optimal for the data being extracted. (eg follows does not need to pass parent statement data but parent does, so it can be designed with less overhead)

3. *Open-Closed Principle*

The Design Extractor adheres to OCP since different extractors handle the extraction of their specific data. Should there be new abstractions to be extracted in the future, we can easily extend our code to include new extractors to handle those cases without modifying our existing extractors.

Use of Design Patterns

1. Iterator pattern

It is used by the TokenIterator class for bounds safe iteration over the vector of Tokens and decreasing space and time overhead encountered from copying sections of the vector. Provides a common platform for classes which use Token vectors to build upon

2. Facade pattern

The Design Extractor acts as a facade between the different Extractors and the Source Processor. Since there are many extractors traversing the AST, the Source Processor does not need to know these internal components and implementations. Hence, the Design Extractor only exposes one public API extractAST() for the Source Processor.

3. *Chain of responsibility*

It is used by the Validator class which prevents invalid SIMPLE code from reaching the Parser component and decreases the amount of checking needed to be implemented by Parser. It is also used by the internal components of Validator; earlier steps enforce guarantees required for CallValidator to run.

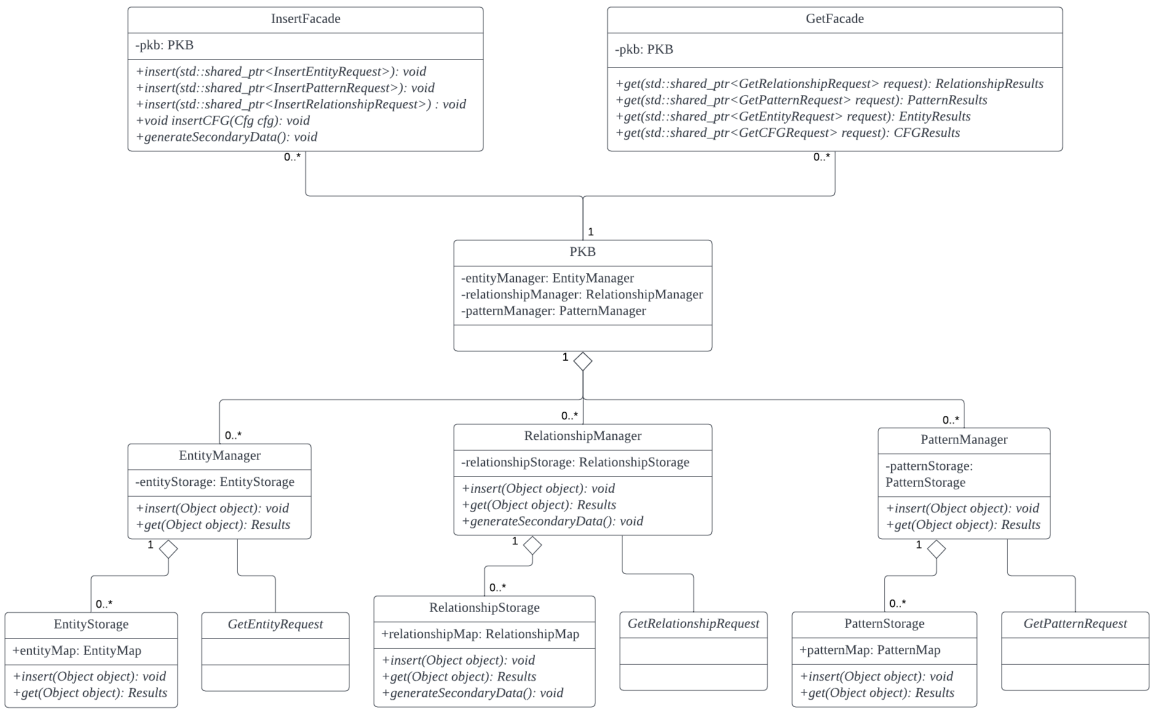
Changes since MS2

1. Redesign of CFG internal data structures
   - Addition of CFG dummy nodes (end of while loop, if-else joining, end of procedure) to aid in Affects/* extraction.
   - Storing of statement type in CFG to identify assign statements for Affects/* extraction.
2. Add one more layer of abstraction for Design Extractor
   - Design Extractor now only calls 3 extractors: Entity, Relationship, and Pattern extractors.
   - Each of these 3 extractors will then call its specific extractors to extract the relevant data.
   - E.g., ParentExtractor extracts parent data and sends them to RelationshipExtractor. RelationshipExtractor accumulates all data and creates InsertRelationshipRequests and sends them to the main DesignExtractor. SPManager obtains these requests and sends them to PKB.
   - Polymorphic requests allow us to encapsulate each different type of relationships and remove the huge amount of for-loops used at SPManager.
   - Further adheres to OCP i.e., SPManager not modified should there be new abstractions.

2.2 PKB

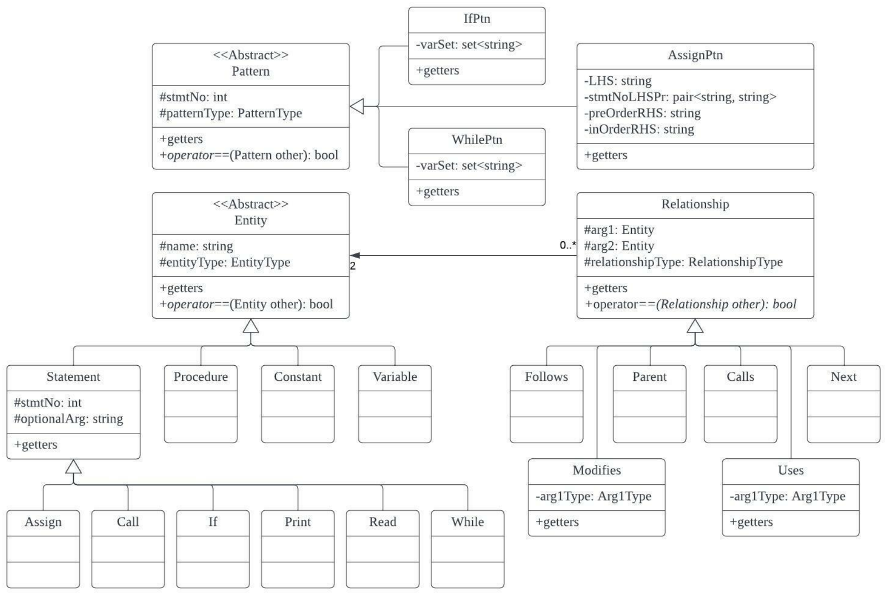
PKB Class Diagram

Class diagrams for PKB structure


InsertFacade
-pkb: PKB
+insert(std::shared_ptr<InsertEntityRequest>): void
+insert(std::shared_ptr<InsertPatternRequest>): void
+insert(std::shared_ptr<InsertRelationshipRequest>) : void
+void insertCFG(Cfg cfg): void
+generateSecondaryData(): void
GetFacade
-pkb: PKB
+get(std::shared_ptr<GetRelationshipRequest> request): RelationshipResults
+get(std::shared_ptr<GetPatternRequest> request): PatternResults
+get(std::shared_ptr<GetEntityRequest> request): EntityResults
+get(std::shared_ptr<GetCFGRequest> request): CFGResults
0..*
0..*
1
PKB
-entityManager: EntityManager
-relationshipManager: RelationshipManager
-patternManager: PatternManager
1
0..*
0..*
0..*
EntityManager
-entityStorage: EntityStorage
+insert(Object object): void
+get(Object object): Results
RelationshipManager
-relationshipStorage: RelationshipStorage
+insert(Object object): void
+get(Object object): Results
+generateSecondaryData(): void
PatternManager
-patternStorage: PatternStorage
+insert(Object object): void
+get(Object object): Results
1
1
1
0..*
0..*
0..*
EntityStorage
+entityMap: EntityMap
+insert(Object object): void
+get(Object object): Results
GetEntityRequest
RelationshipStorage
+relationshipMap: RelationshipMap
+insert(Object object): void
+get(Object object): Results
+generateSecondaryData(): void
GetRelationshipRequest
PatternStorage
+patternMap: PatternMap
+insert(Object object): void
+get(Object object): Results
GetPatternRequest

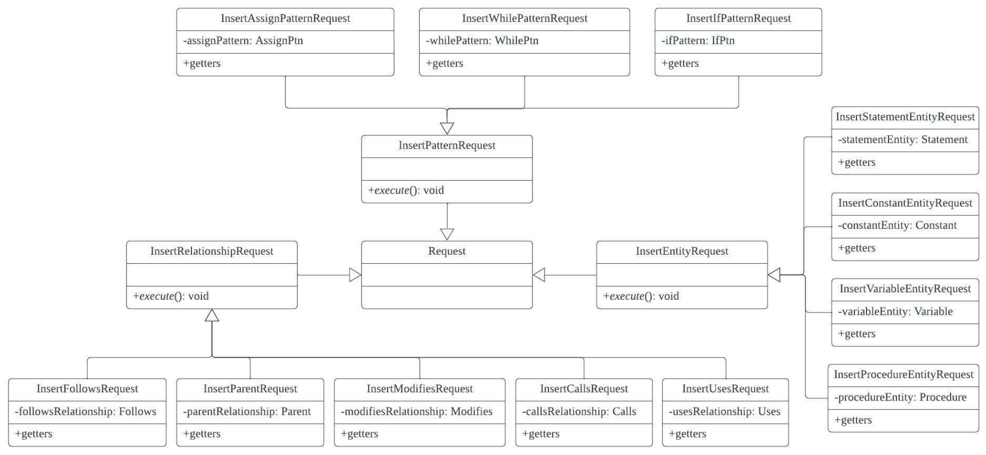
- InsertFacade and GetFacade are responsible for client interaction at facade level via APIs.
- PKB store the different Managers. The PKB object is referenced by the top-level facades.
- Manager classes are associated with Requests. They process these requests and execute them.
- Storage classes will be responsible for direct data access, and they return data to the Managers when the request is successfully executed.

Class Diagram: Entity/Relationship/Pattern Objects

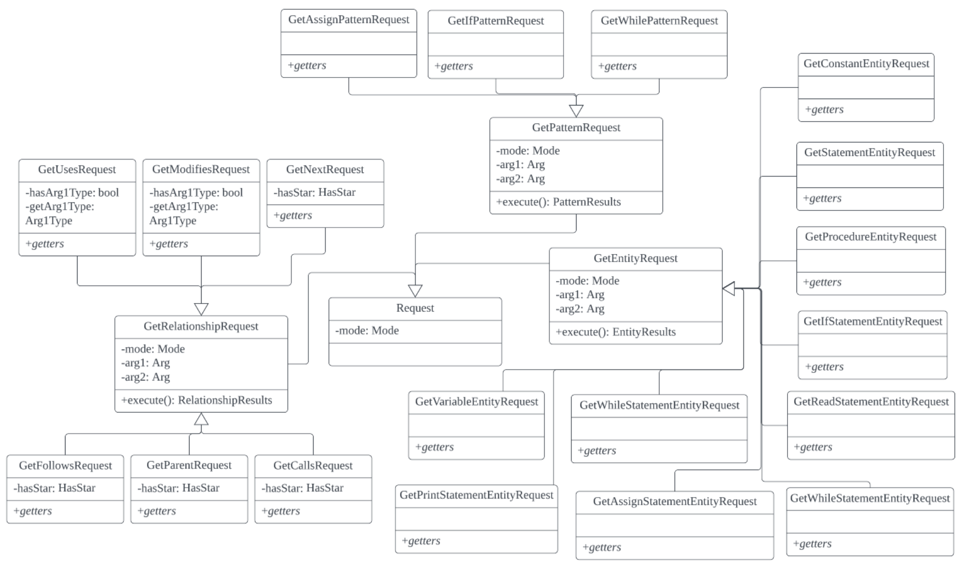


Entity class: The basis of tokens in the Source Program itself.
Relationship class: Defines meaningful behaviour and interaction between Entities.
Pattern class: Captures the pattern associated with the statement

Class diagrams for InsertRequest Objects


InsertAssignPatternRequest
-assignPattern: AssignPtn
+getters
InsertWhilePatternRequest
-whilePattern: WhilePtn
+getters
InsertIfPatternRequest
-ifPattern: IfPtn
+getters
InsertPatternRequest
+execute(): void
InsertStatementEntityRequest
-statementEntity: Statement
+getters
InsertConstantEntityRequest
-constantEntity: Constant
+getters
InsertRelationshipRequest
+execute(): void
Request
InsertEntityRequest
+execute(): void
InsertVariableEntityRequest
-variableEntity: Variable
+getters
InsertProcedureEntityRequest
-procedureEntity: Procedure
+getters
InsertFollowsRequest
-followsRelationship: Follows
+getters
InsertParentRequest
-parentRelationship: Parent
+getters
InsertModifiesRequest
-modifiesRelationship: Modifies
+getters
InsertCallsRequest
-callsRelationship: Calls
+getters
InsertUsesRequest
-usesRelationship: Uses
+getters

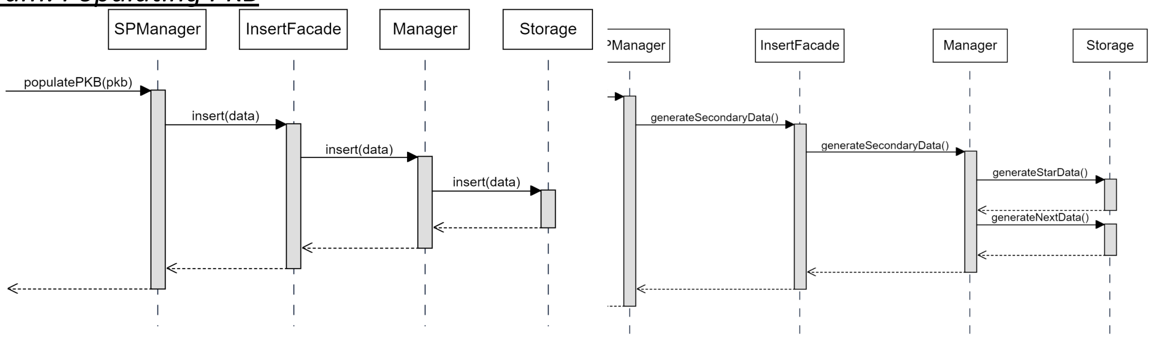
Class diagram for GetRequest Objects



GetRequest class: Each child request overrides the **execute()** method in Get<Type>Request to know which storage to obtain data from and handle the logic behind using the required parameters.

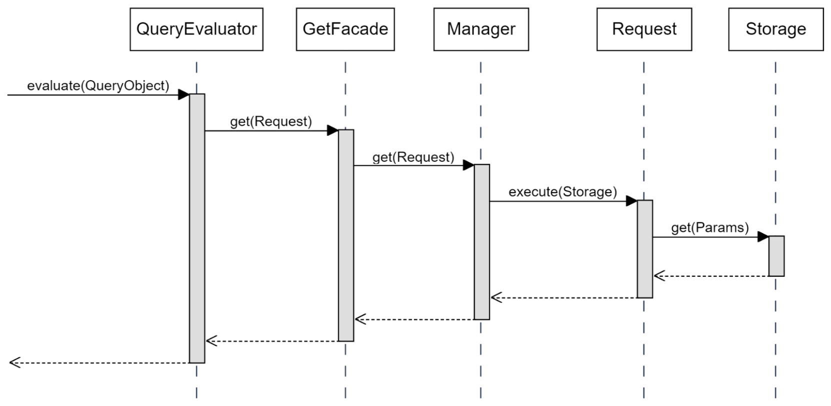
InsertRequest class: Each child request overrides the **execute()** method in Insert<Type>Request to know which storage to insert data into and handle the logic behind using the required parameters.

Sequence diagram: Populating PKB



- SPManager inserts data (Entities/Patterns/Relationships) into the PKBFacade via **insert**.
- For **each procedure** SPManager parses, a Cfg object representing it is stored in CFGStorage.
- **End of the program parsing,** SPManager calls generateSecondaryData() to generate new relationship data from the existing data (Follows*/Parent*/Calls*/Next from CFG).

Sequence diagram: Retrieving data



- QueryEvaluator forms a **Request** object based on QueryObject
- Passed on to PKB Get APIs and execute the request based on the request parameters.
Parameters include the Type: Pattern/Entity/Relationships, or Filter: Arg1, Arg2, Both or None.

PKB Design Decisions

Design principles

1. *Single Responsibility Principle*
- Facade classes bridge clients' insert and get requests independently.
- Manager classes manage the requests that PKB gets and processes them.
- Request classes execute requests, and obtain data from the Storages based on its parameters.
- Storage classes handle the direct storing and getting of data in the PKB.

2. *Separation of concerns*
- Managers that process the Request object independently.
- Request executes logic independently.
- No direct communication across Storage classes.

3. *DRY principle*
- Many APIs in Storage have overlapping methods and functions used.
- Common logic abstracted out into common methods in Util class
- e.g. In CallsStorage, common methods include insertIntoTable(), handleStar(), etc.

4. *Open-closed principle*
- Requests in Facades are polymorphic, so that on the Facade level, there will just be a single get and insert method exposed to clients.
- Helps the client side to interact with the PKB APIs easier.

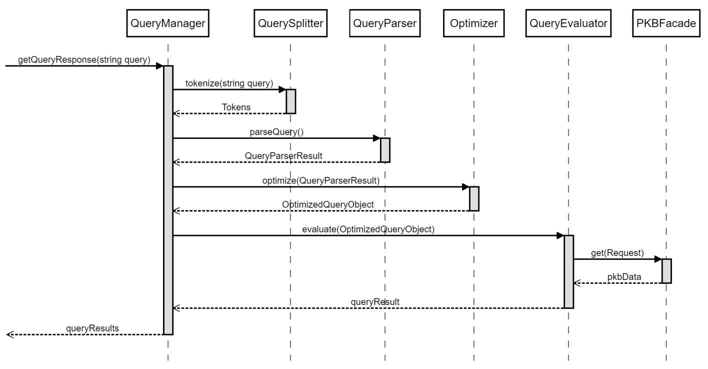
Use of design pattern

1. *Facade pattern*
- PKB serves as a middleman between SP and PQL.
- Facade pattern helps to hide the internal complex subsystems from clients.
- Helps SP and PQL interact easily with the PKB.

2. *Command pattern*
- InsertRequest and GetRequests serve as Command objects.
- InsertFacade and GetFacade become independent of how the different Request objects perform.
- InsertFacade and GetFacade do not need any information on the Requests objects, instead, these information are encapsulated within the Command objects themselves.

2.3 PQL

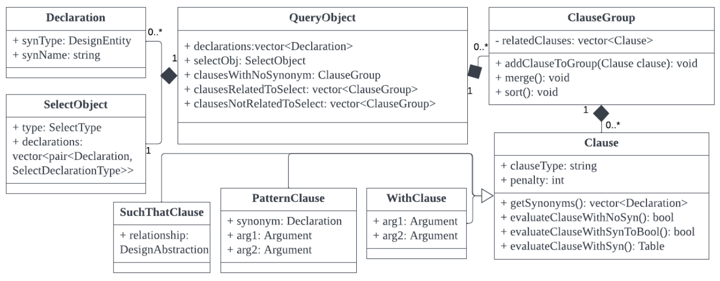
PQL Core Sequence Diagram

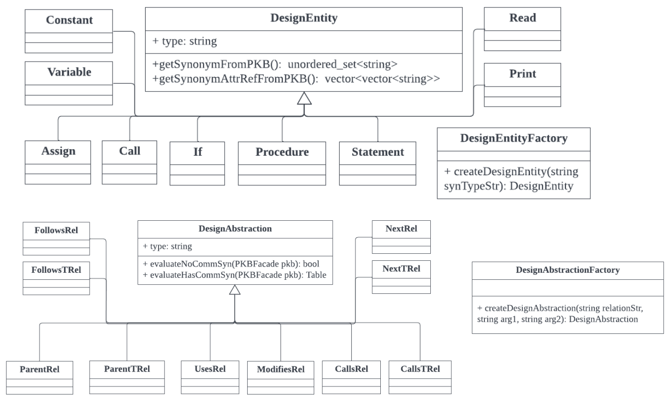


- QueryManager: Manages QuerySplitter, QueryParser, Optimizer and QueryEvaluator.
- QuerySplitter: Tokenizes the query string
- QueryParser: Parses tokens and generates a QueryParserResult.
- Optimizer: Groups and sorts clauses, and generates QueryObject.
- QueryEvaluator: Evaluates QueryObject by formings Requests and interacting with PKB's GetFacade.

PQL Class Diagrams

QueryObject, Design Entities and Design Abstractions


Declaration
+ synType: DesignEntity
+ synName: string
SelectObject
+ type: SelectType
+ declarations: vector<pair<Declaration, SelectDeclarationType>>
QueryObject
+ declarations:vector<Declaration>
+ selectObj: SelectObject
+ clausesWithNoSynonym: ClauseGroup
+ clausesRelatedToSelect: vector<ClauseGroup>
+ clausesNotRelatedToSelect: vector<ClauseGroup>
ClauseGroup
- relatedClauses: vector<Clause>
+ addClauseToGroup(Clause clause): void
+ merge(): void
+ sort(): void
Clause
+ clauseType: string
+ penalty: int
+ getSynonyms(): vector<Declaration>
+ evaluateClauseWithNoSyn(): bool
+ evaluateClauseWithSynToBool(): bool
+ evaluateClauseWithSyn(): Table
SuchThatClause
+ relationship: DesignAbstraction
PatternClause
+ synonym: Declaration
+ arg1: Argument
+ arg2: Argument
WithClause
+ arg1: Argument
+ arg2: Argument
0..*
1
0..*
1
1
0..*
1

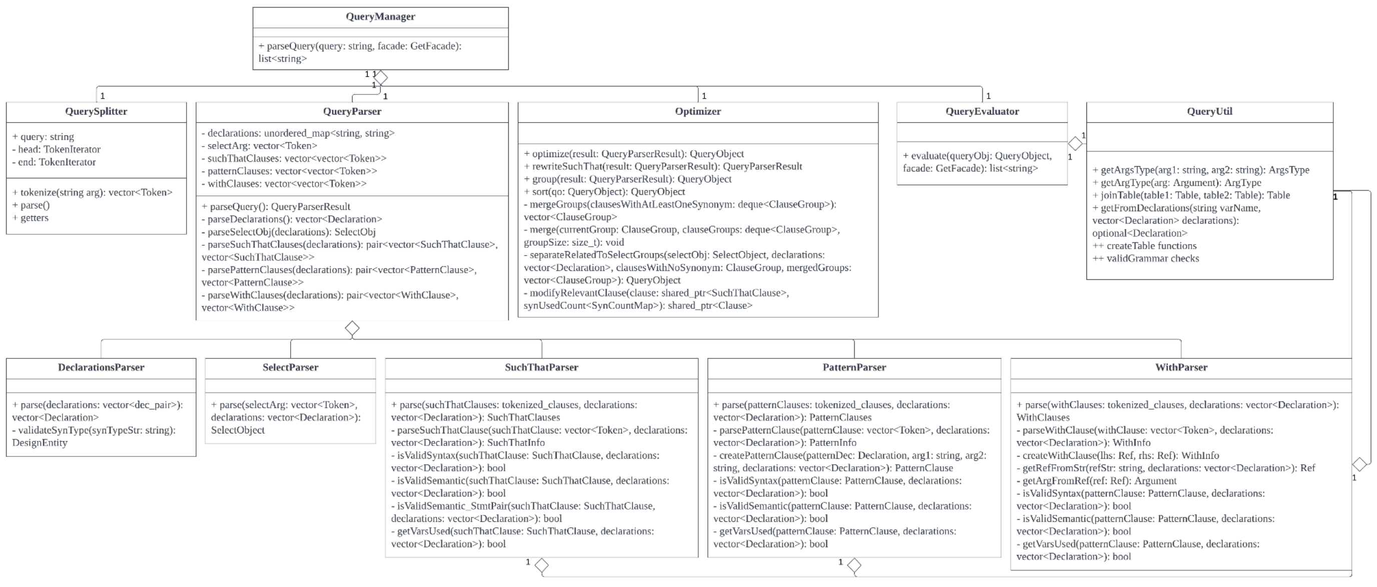

Constant
Variable
DesignEntity
+ type: string
+getSynonymFromPKB(): unordered_set<string>
+getSynonymAttrRefFromPKB(): vector<vector<string>>
Read
Print
Assign
Call
If
Procedure
Statement
DesignEntityFactory
+ createDesignEntity(string synTypeStr): DesignEntity
FollowsRel
FollowsTRel
DesignAbstraction
+ type: string
+ evaluateNoCommSyn(PKBFacade pkb): bool
+ evaluateHasCommSyn(PKBFacade pkb): Table
NextRel
NextTRel
DesignAbstractionFactory
+ createDesignAbstraction(string relationStr, string arg1, string arg2): DesignAbstraction
ParentRel
ParentTRel
UsesRel
ModifiesRel
CallsRel
CallsTRel

The QueryObject consists of Declarations, SelectObject, and ClauseGroups to be evaluated. ClauseGroup contains Clauses (eg. WithClause) related to each other by synonyms. DesignEntity represents an entity (eg. Constant) and DesignAbstraction represents a relationship (eg. Follows).

Updates from MS2 and reasons:

- *Addition of Clause parent class to represent SuchThatClause, PatternClause and WithClause* to act as polymorphic objects in evaluating the different clause types, to adhere to Open-Closed principle
- *Use of Argument class* to represent arguments within each Clause, as a variant holding alternatives

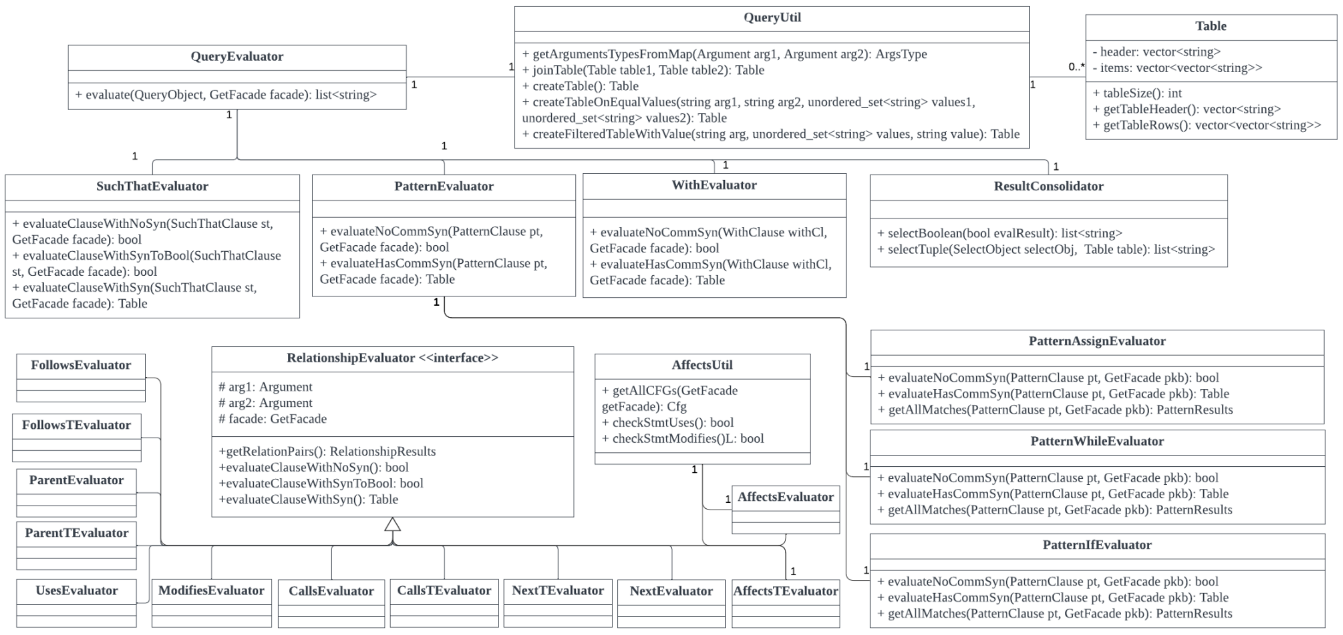
Class Diagrams for PQL Structure: QueryParser, QuerySplitter, Optimizer


QueryManager
+ parseQuery(query: string, facade: GetFacade): list<string>
1
1
1
1
1
1
1
QuerySplitter
+ query: string
- head: TokenIterator
- end: TokenIterator
+ tokenize(string arg): vector<Token>
+ parse()
+ getters
QueryParser
- declarations: unordered_map<string, string>
- selectArg: vector<Token>
- suchThatClauses: vector<vector<Token>>
- patternClauses: vector<vector<Token>>
- withClauses: vector<vector<Token>>
+ parseQuery(): QueryParserResult
- parseDeclarations(): vector<Declaration>
- parseSelectObj(declarations): SelectObj
- parseSuchThatClauses(declarations): pair<vector<SuchThatClause>, vector<SuchThatClause>>
- parsePatternClauses(declarations): pair<vector<PatternClause>, vector<PatternClause>>
- parseWithClauses(declarations): pair<vector<WithClause>, vector<WithClause>>
Optimizer
+ optimize(result: QueryParserResult): QueryObject
+ rewriteSuchThat(result: QueryParserResult): QueryParserResult
+ group(result: QueryParserResult): QueryObject
+ sort(qo: QueryObject): QueryObject
- mergeGroups(clausesWithAtLeastOneSynonym: deque<ClauseGroup>): vector<ClauseGroup>
- merge(currentGroup: ClauseGroup, clauseGroups: deque<ClauseGroup>, groupSize: size_t): void
- separateRelatedToSelectGroups(selectObj: SelectObject, declarations: vector<Declaration>, clausesWithNoSynonym: ClauseGroup, mergedGroups: vector<ClauseGroup>): QueryObject
- modifyRelevantClause(clause: shared_ptr<SuchThatClause>, synUsedCount<SynCountMap>): shared_ptr<Clause>
QueryEvaluator
+ evaluate(queryObj: QueryObject, facade: GetFacade): list<string>
1
1
QueryUtil
+ getArgsType(arg1: string, arg2: string): ArgsType
+ getArgType(arg: Argument): ArgType
+ joinTable(table1: Table, table2: Table): Table
+ getFromDeclarations(string varName, vector<Declaration> declarations): optional<Declaration>
++ createTable functions
++ validGrammar checks
1
DeclarationsParser
+ parse(declarations: vector<dec_pair>): vector<Declaration>
- validateSynType(synTypeStr: string): DesignEntity
SelectParser
+ parse(selectArg: vector<Token>, declarations: vector<Declaration>): SelectObject
SuchThatParser
+ parse(suchThatClauses: tokenized_clauses, declarations: vector<Declaration>): SuchThatClauses
- parseSuchThatClause(suchThatClause: vector<Token>, declarations: vector<Declaration>): SuchThatInfo
- isValidSyntax(suchThatClause: SuchThatClause, declarations: vector<Declaration>): bool
- isValidSemantic(suchThatClause: SuchThatClause, declarations: vector<Declaration>): bool
- isValidSemantic_StmtPair(suchThatClause: SuchThatClause, declarations: vector<Declaration>): bool
- getVarsUsed(suchThatClause: SuchThatClause, declarations: vector<Declaration>): bool
PatternParser
+ parse(patternClauses: tokenized_clauses, declarations: vector<Declaration>): PatternClauses
- parsePatternClause(patternClause: vector<Token>, declarations: vector<Declaration>): PatternInfo
- createPatternClause(patternDec: Declaration, arg1: string, arg2: string, declarations: vector<Declaration>): PatternClause
- isValidSyntax(patternClause: PatternClause, declarations: vector<Declaration>): bool
- isValidSemantic(patternClause: PatternClause, declarations: vector<Declaration>): bool
- getVarsUsed(patternClause: PatternClause, declarations: vector<Declaration>): bool
WithParser
+ parse(withClauses: tokenized_clauses, declarations: vector<Declaration>): WithClauses
- parseWithClause(withClause: vector<Token>, declarations: vector<Declaration>): WithInfo
- createWithClause(lhs: Ref, rhs: Ref): WithInfo
- getRefFromStr(refStr: string, declarations: vector<Declaration>): Ref
- getArgFromRef(ref: Ref): Argument
- isValidSyntax(patternClause: PatternClause, declarations: vector<Declaration>): bool
- isValidSemantic(patternClause: PatternClause, declarations: vector<Declaration>): bool
- getVarsUsed(patternClause: PatternClause, declarations: vector<Declaration>): bool
1
1
1

<u>Updates from MS2 and reasons:</u>

- *Addition of Optimizer* to group and sort clauses for optimization
- *Addition of WithParser* to parse With clauses

<u>Class Diagrams for Query Evaluator</u>


QueryUtil
+ getArgumentsTypesFromMap(Argument arg1, Argument arg2): ArgsType
+ joinTable(Table table1, Table table2): Table
+ createTable(): Table
+ createTableOnEqualValues(string arg1, string arg2, unordered_set<string> values1, unordered_set<string> values2): Table
+ createFilteredTableWithValue(string arg, unordered_set<string> values, string value): Table
Table
- header: vector<string>
- items: vector<vector<string>>
+ tableSize(): int
+ getTableHeader(): vector<string>
+ getTableRows(): vector<vector<string>>
QueryEvaluator
+ evaluate(QueryObject, GetFacade facade): list<string>
SuchThatEvaluator
+ evaluateClauseWithNoSyn(SuchThatClause st, GetFacade facade): bool
+ evaluateClauseWithSynToBool(SuchThatClause st, GetFacade facade): bool
+ evaluateClauseWithSyn(SuchThatClause st, GetFacade facade): Table
PatternEvaluator
+ evaluateNoCommSyn(PatternClause pt, GetFacade facade): bool
+ evaluateHasCommSyn(PatternClause pt, GetFacade facade): Table
WithEvaluator
+ evaluateNoCommSyn(WithClause withCl, GetFacade facade): bool
+ evaluateHasCommSyn(WithClause withCl, GetFacade facade): Table
ResultConsolidator
+ selectBoolean(bool evalResult): list<string>
+ selectTuple(SelectObject selectObj, Table table): list<string>
FollowsEvaluator
FollowsTEvaluator
ParentEvaluator
ParentTEvaluator
UsesEvaluator
RelationshipEvaluator <<interface>>
# arg1: Argument
# arg2: Argument
# facade: GetFacade
+getRelationPairs(): RelationshipResults
+evaluateClauseWithNoSyn(): bool
+evaluateClauseWithSynToBool: bool
+evaluateClauseWithSyn(): Table
AffectsUtil
+ getAllCFGs(GetFacade getFacade): Cfg
+ checkStmtUses(): bool
+ checkStmtModifies()L: bool
AffectsEvaluator
ModifiesEvaluator
CallsEvaluator
CallsTEvaluator
NextTEvaluator
NextEvaluator
AffectsTEvaluator
PatternAssignEvaluator
+ evaluateNoCommSyn(PatternClause pt, GetFacade pkb): bool
+ evaluateHasCommSyn(PatternClause pt, GetFacade pkb): Table
+ getAllMatches(PatternClause pt, GetFacade pkb): PatternResults
PatternWhileEvaluator
+ evaluateNoCommSyn(PatternClause pt, GetFacade pkb): bool
+ evaluateHasCommSyn(PatternClause pt, GetFacade pkb): Table
+ getAllMatches(PatternClause pt, GetFacade pkb): PatternResults
PatternIfEvaluator
+ evaluateNoCommSyn(PatternClause pt, GetFacade pkb): bool
+ evaluateHasCommSyn(PatternClause pt, GetFacade pkb): Table
+ getAllMatches(PatternClause pt, GetFacade pkb): PatternResults
1
0..*

<u>Updates from MS2 and reasons:</u>

- *Reconstruction of RelationshipEvaluator* to be constructed with PKB's GetFacade instead of being passed in as arguments to the class methods, and *refactor of methods* to adhering to template method pattern, to follow DRY principle (explained in design patterns section)

<u>PQL Design Principles</u>

1. *<u>Single Responsibility Principle</u>*
- <u>QuerySplitter</u> tokenizes and splits the query string into its relevant parts
- <u>QueryParser</u> parses and validates the query strings to form the QueryObject through individual Parsers for the declarations, select clause and each conditional clause, to construct the QueryObject.
- <u>QueryEvaluator</u> handles evaluation of query objects. SuchThatEvaluator, PatternEvaluator and WithEvaluator handles evaluation of such-that clauses, pattern clauses and with clauses respectively.
- <u>ResultConsolidator</u> class selects the results to be returned.

2. *<u>Separation of concerns</u>*
- Only the <u>QueryManager</u> interacts with the UI
- Boundaries are set in between the sub-components of PQL, the QuerySplitter, QueryParser and QueryEvaluator. Boundaries in responsibilities are set within the Parsers and Evaluators classes that handle parsing and evaluation of different clauses independent of each other.

3. *<u>DRY principle</u>*
- Function logic that is reused in the evaluators are placed in <u>QueryUtil</u>, which in turn runs overlapping source code definition validation checks through the <u>ValidatorUtil</u> class.
- <u>Table</u> class is an abstraction used to store and manipulate results from the evaluations of clauses.
- <u>Tokenizer</u> and <u>TokenizerIterator</u> are used in QuerySplitter and QueryManager significantly to iterate safely and with no concern for whitespace between, for queries.

4. *Interface Segregation principle*
RelationshipEvaluator is an interface specifically for the evaluators of such-that design abstractions, to adhere to the **Interface Segregation Principle**, so that only methods required for such-that design abstraction evaluation are exposed.

5. *Open/closed principle*
- Upon addition of the WithClause in MS2, QueryEvaluator is open to the addition of the WithEvaluator with its inherited evaluatoNoCommSyn() and evaluateHasCommSyn() methods for evaluating clauses with no common shared synonyms, and clauses which do, respectively.
- QueryParser designed with specific QueryParsers that handle specific parts of the PQL query.

PQL Design Patterns
1. *Facade pattern*
- QueryManager is a facade class for the UI to interact with, without needing to understand any implementation details or changes beneath, with one simple method getQueryResponse()
- QuerySplitter, QueryParser and QueryEvaluator are facade classes which hide complex subsystems in the pipeline, exposing only QueryEvaluator::evaluate() and QueryParser::parse() API.

2. *Factory pattern*
- DesignAbstraction provides an interface for representing relationships in SuchThatClause. This removes chunky if-elses to identify the relationship in the SuchThatEvaluator. This was done using the different DesignAbstraction **polymorphic objects** (eg. FollowsRel, NextTRel) for the different relationships, adhering to **open-closed principle**. DesignAbstractionFactory is used to create the different DesignAbstraction objects. The same is done for DesignEntity and DesignEntityFactory.

3. *Template Method pattern*
- RelationshipEvaluator uses this design pattern to adhere to the DRY principle in specific evaluators by abstracting similar behaviors exhibited.
- The evaluateClauseWithSyn() is the template method that requires querying PKB for the relevant relationships, and transforms the data into a table and returns it. It uses the helper method getRelationPairs() which is abstract, and overridden in each child evaluator to obtain the relevant relationships data.

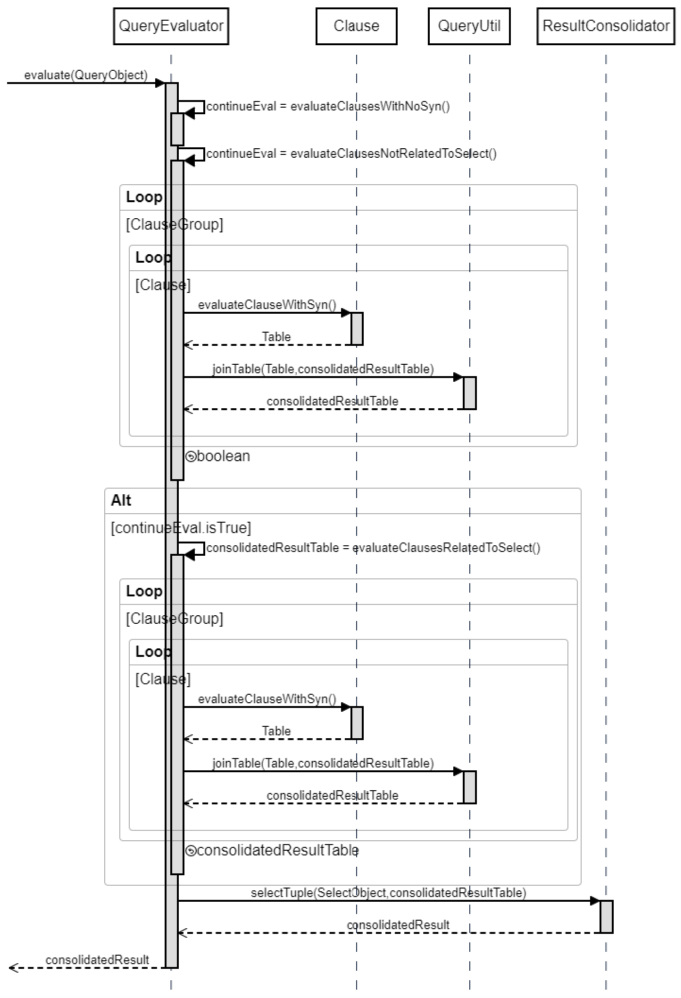
PQL Design Decision

Use of Arguments over strings in Clauses
- Argument holds clause arguments in their original class types rather than as strings
- No need for re-verifying type of argument passed to clauses before evaluation

Order of Evaluation in Query Evaluator

Query Evaluator **order of evaluation optimized** for faster query:

1. Clauses with no synonyms: Evaluated to boolean so evaluation can be terminated early.
2. Clause Groups with synonyms that are not related to Select synonyms: Clauses within a group are evaluated to produce a table each, which are joined to produce a final table. Clause Groups are evaluated to boolean so evaluation can be terminated early.
3. Clause Groups with synonyms that are related to Select synonyms: Tables from evaluation of Clause Groups are joined to produce a final table.
4. Select object (boolean/single/tuple) is evaluated last.



Optimizations

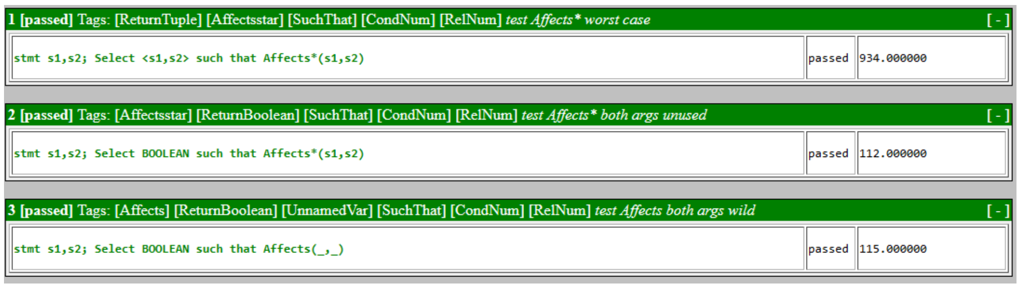
Evaluating * using the basic evaluator

Next* and Affects* clauses which contain wildcards can be evaluated using the more straightforward and faster algorithm of Next and Affects evaluators

Rewriting clauses

Mutating Next*, Affects and Affects* clauses where the synonym is unused anywhere else in the query provides a major speed boost, because we can then use the more optimized algorithms that have less requirements to check and keep track of.

Running a stress test where there are many Affects* relationships to account for, the unoptimized Affects* where both synonyms are part of evaluation takes 8x as long to run compared to the optimized version where both synonyms have been converted to wildcard before evaluation, and therefore is evaluated as Affects(_,_) instead.



1 [passed] Tags: [ReturnTuple] [Affectsstar] [SuchThat] [CondNum] [RelNum] *test Affects* worst case* [-]

`stmt s1,s2; Select <s1,s2> such that Affects*(s1,s2)`	passed	934.000000

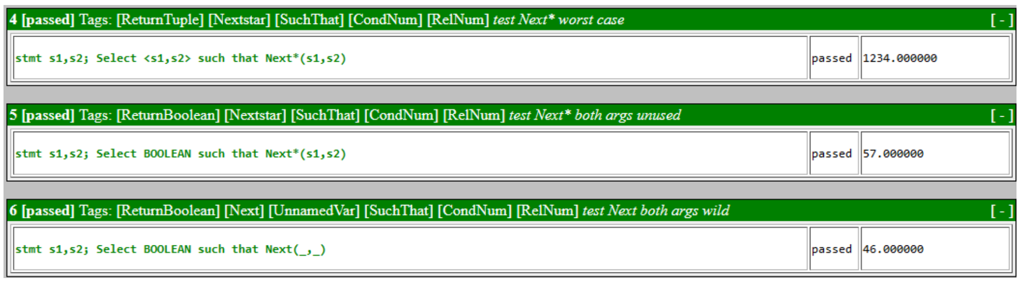
2 [passed] Tags: [Affectsstar] [ReturnBoolean] [SuchThat] [CondNum] [RelNum] *test Affects* both args unused* [-]

`stmt s1,s2; Select BOOLEAN such that Affects*(s1,s2)`	passed	112.000000

3 [passed] Tags: [Affects] [ReturnBoolean] [UnnamedVar] [SuchThat] [CondNum] [RelNum] *test Affects both args wild* [-]

`stmt s1,s2; Select BOOLEAN such that Affects(_,_)`	passed	115.000000

The same optimization changes Next*(s1,s2) to Next(_,_) , and resulted in a 20x speedup



4 [passed] Tags: [ReturnTuple] [Nextstar] [SuchThat] [CondNum] [RelNum] *test Next* worst case* [-]

`stmt s1,s2; Select <s1,s2> such that Next*(s1,s2)`	passed	1234.000000

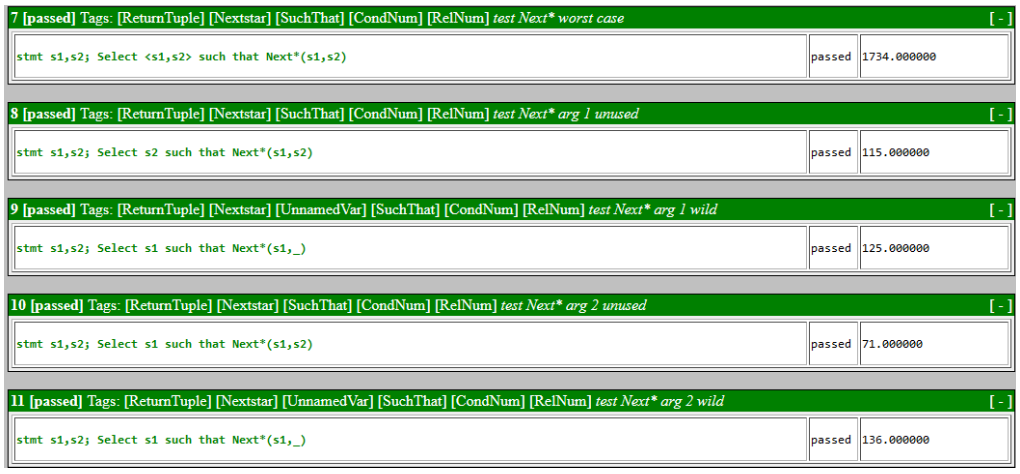
5 [passed] Tags: [ReturnBoolean] [Nextstar] [SuchThat] [CondNum] [RelNum] *test Next* both args unused* [-]

`stmt s1,s2; Select BOOLEAN such that Next*(s1,s2)`	passed	57.000000

6 [passed] Tags: [ReturnBoolean] [Next] [UnnamedVar] [SuchThat] [CondNum] [RelNum] *test Next both args wild* [-]

`stmt s1,s2; Select BOOLEAN such that Next(_,_)`	passed	46.000000

While the improvements are most significant when both synonyms are replaced with wildcards, there are also improvements to runtime when only one synonym is replaced with a wildcard.



7 [passed] Tags: [ReturnTuple] [Nextstar] [SuchThat] [CondNum] [RelNum] *test Next* worst case* [-]

`stmt s1,s2; Select <s1,s2> such that Next*(s1,s2)`	passed	1734.000000

8 [passed] Tags: [ReturnTuple] [Nextstar] [SuchThat] [CondNum] [RelNum] *test Next* arg 1 unused* [-]

`stmt s1,s2; Select s2 such that Next*(s1,s2)`	passed	115.000000

9 [passed] Tags: [ReturnTuple] [Nextstar] [UnnamedVar] [SuchThat] [CondNum] [RelNum] *test Next* arg 1 wild* [-]

`stmt s1,s2; Select s1 such that Next*(s1,_)`	passed	125.000000

10 [passed] Tags: [ReturnTuple] [Nextstar] [SuchThat] [CondNum] [RelNum] *test Next* arg 2 unused* [-]

`stmt s1,s2; Select s1 such that Next*(s1,s2)`	passed	71.000000

11 [passed] Tags: [ReturnTuple] [Nextstar] [UnnamedVar] [SuchThat] [CondNum] [RelNum] *test Next* arg 2 wild* [-]

`stmt s1,s2; Select s1 such that Next*(s1,_)`	passed	136.000000

Grouping of clauses
- Clauses are **first** divided into 3 divisions:
    1. Clauses without any synonyms
    2. Clauses with synonyms that are related directly/indirectly with Select synonyms
    3. Clauses with synonyms that have no relation to Select synonyms
- Within each of these 3 divisions, we will segregate clauses by **grouping** them into ClauseGroups if they have some direct/indirect common synonym with each other.
- This way, during evaluation, the table size will be kept smaller due to the presence of overlapping synonyms.

Sorting of clauses
- Each clause is given a **penalty** value based on either the characteristics of the clauses

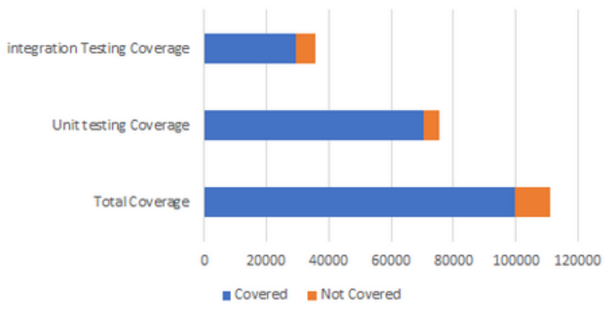
Factors	Penalty applied	Reasoning
SuchThat/With/Pattern Clauses	500	-
Number of Synonyms	500 per synonym	Clauses with fewer synonyms should be evaluated first since they are more restrictive.
Star	1000	Rel* clauses are likely to have a larger table
Runtime	2000	Clauses that are computed during query time and should be done last.
Affects/Affects* type	10000	Should be done last since they are computationally expensive.

- The Clauses will then be sorted based on this penalty value using the std::vector sort within the ClauseGroups.

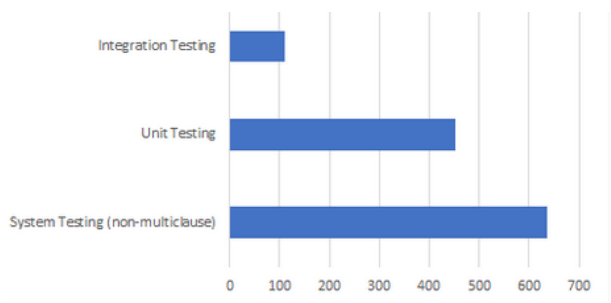
3. Test Strategy

To ensure that our SPA works in all possible situations, continuous testing is required. There are 3 types of tests : Unit testing, Integration testing and System testing. The first graph below shows the code coverage by integration and unit testing through LOC. The second shows the number of test cases for each type of testing

Test type code coverage



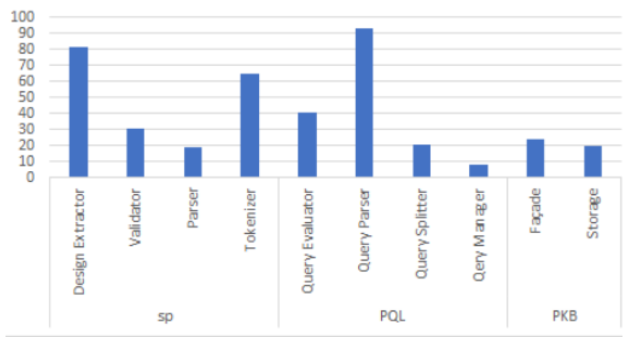
No. Tests per type



Unit Testing

Unit testing is done on each individual component of the software to ensure that the SPA performs as designed. It is usually written by the developer of the component and is done on every public function used within the component. Using this bottom-up testing approach, we can ensure that the components are behaving as expected by the developer. In addition to that, code coverage could be used to decide which functions require further testing. The graph next to this shows a rough overview of the number unit testing per component.

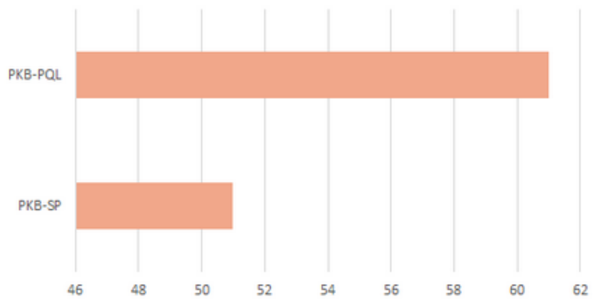
No. Test cases per Component



Integration Testing

Integration testing is done to ensure that components are able to interact with each other. These are the few components that have to talk to each other: Parser-DesignExtractor, DesignExtractor-PKB , PKB-QueryEvaluator and Query Evaluator-Query parser. When picking test cases for integration testing, there was a focus on information transfer between two components. An example would be testing whether requested information from PKB is accurate and comes in a form that QPS can easily use. This can be achieved by creating stubs and drivers that mock the component attempting to pass/receive information between. Test cases are usually written to confirm results given by the component sending the information.

No. Test cases per connection

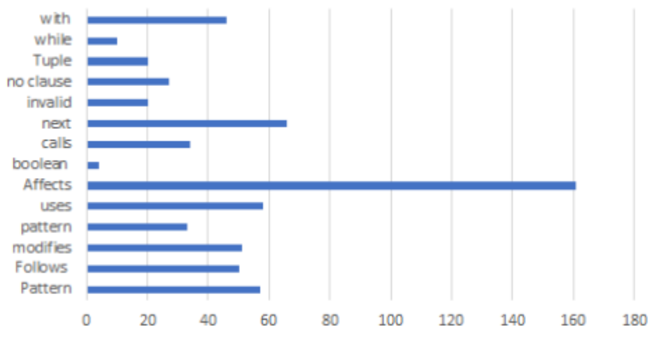


Other than passing information, integration testing also focused on covering as many syntactically possible inputs that come from a query. This is especially important for the PKB-QueryEvaluator. Test cases are formulated based on the grammar rules set in the Wiki. Additional tests were included to ensure that weird inputs such as synonyms with entity names can still be queried. However for queries that have infinitely more variations, such as multi-clauses, only a few were tested here as it is not possible to test all possible variations in integration testing. Instead, we tested them in our system testing.

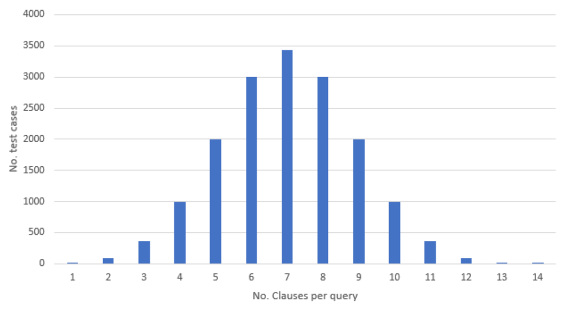
<u>SystemTesting</u>

In System testing, the objective is to cover as many possible queries as possible. To ensure correctness, most possible variations for each type of clause and its different source possibilities were covered thoroughly. This is done by referencing all valid and invalid grammar rules found in the wiki, creating test categories and permuting different overlaps of possible arguments and source codes. Source codes were also written to be versatile, often containing numerous statement types.

<u>No. of test cases per clause</u>

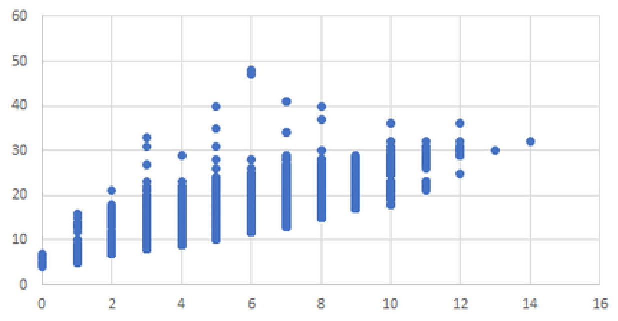


<u>No. Test cases vs No. clauses per query</u>

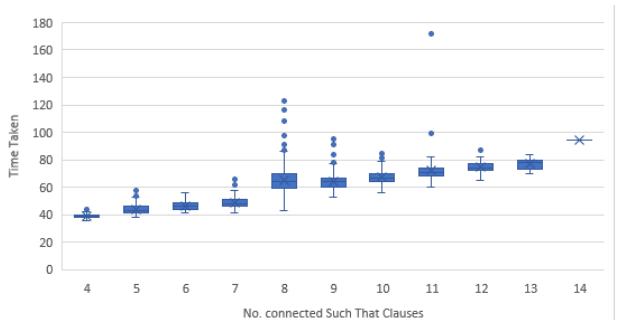


The addition of the multi-clauses functionality had to be extensively tested to ensure that queries provide the correct answer and in a reasonable time (no timeout). To ensure this, we had to test permutations as well as combinations of the clauses. Based on our testing, it was clear that the number of clauses does increase the overall time taken (bottom left figure). The bottom right figure shows that clauses that are connected by common synonyms significantly increase the overall time. Time taken ranges significantly due to the sheer number of test cases (top right) that these combinations have.

<u>Time taken(ms) per no. clause (not connected)</u>

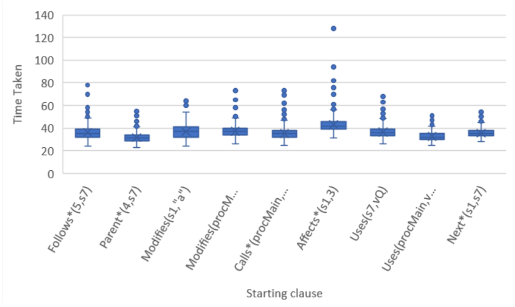


<u>Time taken(ms) per no. clause (connected)</u>

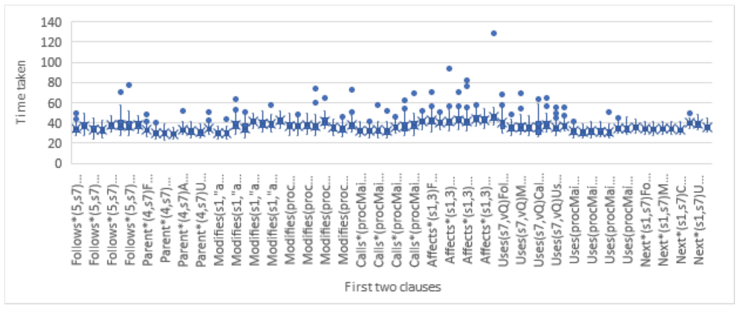


To better understand why there are variations between clauses of the same length, we did permutations of queries with 4 connected clauses. Our testing shows that using the Affects/Affects* clauses significantly impacts the query time taken. This is very useful especially when optimizing the system query evaluation.

<u>Time taken(ms) vs Starting clause</u>



<u>Time taken(ms) vs First 2 clauses</u>



Using the hypothesis that Affects/Affects* and Next/Next* clauses are likely the heaviest and most time-consuming, stress tests were performed to check that, even in the worst-case scenario, queries using these two relationships returned a response before timeout.

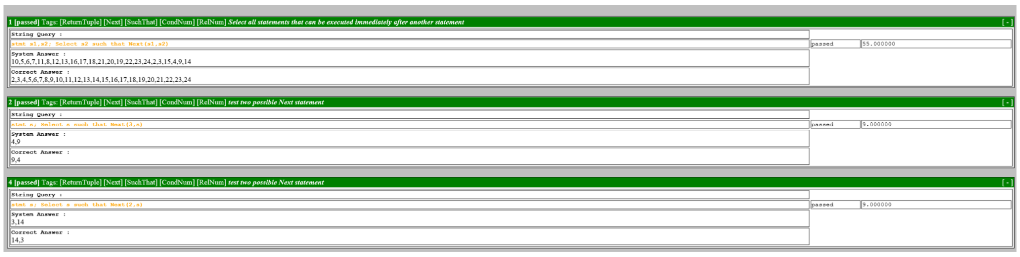
Finally, stress tests were also conducted on the Source Processor to ensure that it can parse a large source file. This was done by creating a source code with 200 procedures with 199 procedure calls.

Automation

Automation of system testing is done through a python script which runs test cases then opens the browser to display the results of the tests. This allows for comparison between different test cases without overwriting them , unlike how running them through visual studio debugger does.

To better facilitate multi-clause testing, we also created a python script to make test cases that covers all possible permutations and combinations.

When paired with our custom analysis.xls, it made it easier to analyse bugs as well as duration of the query. We changed it so queries are arranged based on time taken as well as ensured that system and correct answers are displayed.



```
1 [passed] Tags: [ReturnTuple] [Next] [SuchThat] [CondNum] [RelNum] Select all statements that can be executed immediately after another statement   [-]
String Query :
stmt s1,s2; Select s2 such that Next(s1,s2)                                   passed   55.000000
System Answer :
10,5,6,7,11,8,12,13,16,17,18,21,20,19,22,23,24,2,3,15,4,9,14
Correct Answer :
2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24

2 [passed] Tags: [ReturnTuple] [Next] [SuchThat] [CondNum] [RelNum] test two possible Next statement   [-]
String Query :
stmt s; Select s such that Next(3,s)                                          passed   9.000000
System Answer :
4,9
Correct Answer :
9,4

4 [passed] Tags: [ReturnTuple] [Next] [SuchThat] [CondNum] [RelNum] test two possible Next statement   [-]
String Query :
stmt s; Select s such that Next(2,s)                                          passed   9.000000
System Answer :
3,14
Correct Answer :
14,3
```

Other than testing, we deployed a CI/CD pipeline onto our master branch using GitHub Actions and msbuild. This ensures that any pull requests to be merged into the master branch would have to pass all the unit and integration tests beforehand.

Handling Bugs

Approach to dealing with bugs:
1. Attempt to identify the component where the bug originates
2. Narrow down the scope by adding more specific cases to the test suite
3. Check if the situation is covered by unit or integration tests. If not, add and run.
4. Ask developer of that component to fix
5. Additionally, for common bugs, we have a common "Known Bugs" file for troubleshooting

4. Reflection

Problems encountered
- Certain components that were critical to the system were lacking behind in progress, leading to a slower overall progress.
- The initial design of the query parser was not extensible, so we had to redesign it. The delay caused by this meant that other components could not be system tested while the query parser was being redesigned.
- System was unable to build in the recent PRs and was only discovered hours before submission. This caused unnecessary stress and would be detrimental to the team if left undetected. Unit and systems tests should have been checked before merging the PRs into master.

Areas to improve on next project
- Setting achievable targets. This would allow us to help each other earlier if some parts are too heavy instead of rushing at the last minute
- Seek help early. Being transparent with each other and asking for help when facing certain difficulties allow us to be more efficient and also achieve a better learning outcome.
- Plan the high-level design well and properly first. Starting to code without knowing exactly the design leads to bad coding and makes the code hard to clean and refactor in the future.
- Time management is very important. Waiting till the last minute to test the system leaves little time for debugging. Enough time should be left for system testing to cater for any unexpected incident.

What went well
- Features were implemented completely and tested comprehensively.
- Communicating about what we were working to minimize merge conflicts. Using depreciating interfaces instead of changing them directly to allow for staggered transition of redesign without breaking the system.
- Knowing and fulfilling our roles.

Other project experiences and issues
- Breaking the solution file in the first half of semester and getting very weird error messages.
- Lack of support. When faced with breaking issues with cpp, few options felt available to the team
- Messy commit history. The original master branch became very messy, and had to be entirely replaced by the next milestone. But this taught us to keep better merge discipline.
- Coding standards should be adhered to at all times. Writing poor code quality and leaving it to the end to clean up can be very tedious especially when the code is being used by other components at that stage, making refactoring a very tedious task.

Appendix

SP API (SP Manager)

SPManager construct(string source) Description: Constructs the SPManager object from the SIMPLE source code by running the different steps required to create the AST and extract designs from the AST. @param source The SIMPLE source code to be processed by SPA Returns: The SPManager object containing extracted entities and abstractions.
void populatePKB (PKBfacade& pkb) Description: Takes a reference to a pkb and populates it with the extracted designs Calls the insertion API methods provided by PKB component @param pkb a reference to the PKB which is to be populated

PKB API (InsertFacade)

void insert(InsertEntityRequest request) Description: Inserts an Entity object into the database. (Variable, Constant, Procedure, Statement)
void insert(InsertPatternRequest request) Description: Inserts a Pattern object into the database.
void insert(InsertRelationshipRequest request) Description: Inserts a Relationship (Follows, Parent, Uses, Modifies) object into the database.
void insertCFG(Cfg cfg) Description: Inserts a CFG object into the database.
void generateSecondaryData() Description: To be called by SP after all the CFGs have been made. The method will generate all the necessary secondary data from the current primary data existing in data (Includes Star and Next relationships from CFG)

PKB API (GetFacade)

PatternResults get(GetPatternRequest request) Description: Gets Patterns based on the request type and filters. @param request Contains the mode of GET, and the respective values provided. Returns: The set of filtered patterns.
EntityResults get(GetEntityRequest request) Description: Gets Entities based on the request type and filters. @param request Contains the mode of GET, and the respective values provided. Returns: The set of filtered entities.
RelationshipResults get(GetRelationshipRequest request) Description: Gets Relationships based on the request type and filters. @param request Contains the mode of GET, and the respective values provided. Returns: The set of filtered relationships.

PQL External API

list<string> QueryManager::getQueryResponse(string query)
Description: Parses string query into QueryObject which is evaluated to produce a list of results.
@param query The query input to be parsed and evaluated.
Returns: list<string> of results
Throws SyntaxError and SemanticErrors when found

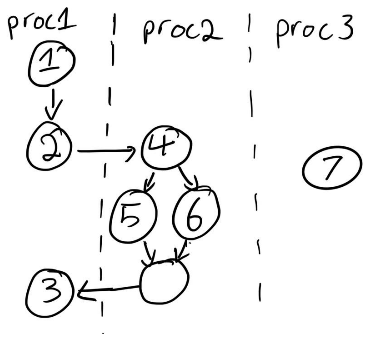
Extension Proposal

Definition

The Program Flow Graph (PFG) displays the order of execution of procedures by following the flow of Call statements into its referenced procedure. While one CFG is created for each procedure, a single PFG covers all procedures in the program.

An example of a PFG for a program

```
procedure proc1{               procedure proc2{           procedure proc3{
    a = 1;           // 1          if (b)           //4       read d;     //7
    call proc2;      // 2          then{read a;}  //5     }
    x = a;           // 3         else {b=b+1;}  //6
}                              }
```



With the PFG, a new Design Abstraction, SuperNext, an extension of Next, can be defined.

For two statements s1 and s2:
- SuperNext(s1, s2) holds if s2 can be executed immediately after s1 in some execution sequence in the Program Flow Graph.
- SuperNext*(s1, s2) holds if s2 can be executed after s1 in some execution sequence in the Program Flow Graph. SuperNext* is the transitive closure of SuperNext.

Example

Referencing the above example code,
the following relationships hold: SuperNext(1, 2), SuperNext(2, 4), SuperNext(5, 3), SuperNext*(1, 3), SuperNext*(2, 6), SuperNext*(4, 3),
while the following relationships do not hold: SuperNext(2, 3), SuperNext*(4, 1) because there is no direct edge from statement 2 to 3 in the PFG, and no path from statement 4 to 1 in the PFG.

Changes Required

The DesignExtractor needs to add a new extractor, the PFGExtractor, to extract the PFG that follows these new rules. The PKB will extend this by adding a new derived SuperNext relationship, SuperNextStorage and PFGStorage, and a GetSuperNextRequest. The PFGStorage will allow us to obtain meaningful data from the PFG itself via the PFG class methods. The QueryParser will add a new such-that clause type to handle the superNext relationship, extending the DesignAbstraction class. The method of requesting data from PKB for this new clause can follow that of existing such-that clauses. Following this, the following definitions are appended, syntactically checked by the QueryParser:

```
relRef: Follows | …  … | AffectsT | SuperNext | SuperNextT

SuperNext: 'SuperNext' '(' stmtRef ',' stmtRef ')'

SuperNextT: 'SuperNext*' '(' stmtRef ',' stmtRef ')'
```

The query including these new clauses is also only semantically valid if, for SuperNext(arg1, arg2) and SuperNext*(arg1, arg2), arg1 and arg2 follow the rule as follows: If the argument is a synonym, it must be a statement synonym, or a subtype of a statement synonym (read, print, assign, if, while, call).
To handle the new SuperNext design abstraction, the QE will add a new SuperNextEvaluator that extends the RelationshipEvaluator.

Implementation

Definitions
Let the terminals of a procedure be the statements which can be the last statements in that procedure. Let the terminals T of a statement S be the statements such that the relationships Follows(S,S1) and Next(T,S1) hold, where S1 is a statement that exists.
For a read,print, assignment or while statement, their terminals would be themself. For an if statement, it would be the combination of the terminal statements of the then and else blocks.

DataStructure
We can store the graph as a directed graph where SuperNext(S1,S2) is modelled as a directed edge pointing from node S1 to node S2. Call statements shall point to the first statement of the procedure that they call. The terminals {T} of procedures shall point to the Next statement of the Call statements which call that procedure.

This can be implemented as an adjacency list, such as a map of statement No to a group of StatementNo, where for each value in the group of values, SuperNext(key,value) holds.

We also require a map of procedure names to the statement number of the first statement in them, which shall be called procFirstStmt, and a map of the procedure to its terminals, called procTerminals.

Constraints
We can only extract the PFG of a procedure after extracting the PFG of all the procedures it calls because the called procedures will be part of a calling procedure's PFG. Thankfully, because of the no recursion and loop constraints on SIMPLE, it is possible to arrange the procedures in a partial order such that called procedures are always evaluated before their calling procedures.

Extraction Algorithm
1) Based on the information extracted in the Calls relationship, create the valid partial order where called procedures are before the procedures that call them. The first procedure will have no call statements inside it.
2) Use Depth first search to extract the PFG similarly to a CFG and generate the SuperNext() relationships.
2.0) The first statement in the procedure has no previous terminals, the current terminal list T shall be an empty list {}. Keep track of the statement number of this first statement in the map procFirstStmt.
2.1) Iterate through the statements in the procedure
2.1.1) If we encounter a Read, Print, or Assign statement S, we extract SuperNext(T,S) for each T in current terminals {T}, then make {S} the current terminal list.

2.1.2) If we encounter a While statement, we recurse into the child statement list, beginning with the while statement {S} as the terminal list. The while statement also becomes the new terminal for the statement that Follows it.
2.1.3) If we encounter an If statement, we recurse into the two child containers with the If statement as the terminal, then concatenate the terminals of the two branches as the terminal list and continue iteration.
2.1.4) If we encounter a Call statement C, we look up the statement number of the first statement of the procedure S1. Add SuperNext(C,S1) to our PFG.
2.1.4.1) Look up the terminals {T} of the called procedure. {T} are the new terminals as we continue iteration.

One possible implementation is linked here, which was produced due to a misunderstanding over the definition of Next

Possible Challenges

PFG is much larger and will take more time to traverse. In the CFG, different procedures are disconnected from each other. This allows for some lookup optimizations, such as Next*(S1,S2) being false if S1 and S2 are in different procedures. However, this assumption does not hold in the PFG and therefore the only way to test for SuperNext* relationship is by traversing the graph of the entire program.

Benefits to SPA

The PFG reflects a more accurate model of the program flow. This can in turn be used to give a more accurate definition of the current Affects and Affects* relationships.
More specifically, for any variable X and procedures P and Q, if P calls Q and there is a path in Q that modifies X and another path that does not modify X, the current definition of modifies states that the call statement modifies X. For example, referencing the above code, statement 2 calls procedure proc2 which modifies variable a, and as such Affects(1, 3) does not hold based on the current definition.
However, with the PFG, the last condition of the definition of Affects can be changed to use the PFG instead of the CFG. Using the PFG, it is possible to identify a path through proc2 without modifying variable a. More specifically, the path 2 → 4 → 6 → 3 does not modify variable a. Since this path exists, Affects(1, 3) should hold based on this new definition of Affects, which is a more accurate reflection of the reality defined by SIMPLE.