CS3201 Assignment 4: Extended Prototype

Due date: **Thursday, 13 November, by 2pm.**

Submission:

1. Report: Create a PDF file, named **Team<XX>.pdf**, where <XX> is your team number. The PDF file should include **the team number, as well as the full names, email addresses and phone numbers of the team members**. (We may need to contact you during testing).
2. Code: Create one folder, named **Code<XX>**, where <XX> is your team number. It should contain a README file and the Visual Studio 2010 Solution with all your code (integrated with Autotester) that we will build, run and test.
3. Test Cases: Create one folder, named "Tests<XX>", where <XX> is your team number. It should contain all your system acceptance test cases. The test cases should consist of at least **THREE** SIMPLE programs, **THREE** queries for each type of query condition (i.e., three for Follows, three for Modifies, three for pattern, etc.) and **TEN** queries for various combinations of query conditions (e.g., two for Follows + pattern, three for Modifies + pattern, etc.).
4. Place the report and the two folders in another folder named **Team<XX>** and archive it in a file named **Team<XX>.zip**. Submit the archive via IVLE: **IVLE -> CS3201 -> Workbin -> Student Submission -> Final Project.**
5. Also hand in a **hardcopy of your project report** into the box outside **COM2 #02-50 by 13 Nov., 2pm**. (The box will be set up at 10am on the same day.)
6. Each team should submit only **one** assignment. For multiple submissions received from the same team, we will mark the most recent one.

Late Assignment Policy:

1. Penalty for report submission within 24 hours after the due date: 5 points (out of 100)
2. Penalty for code submission within 24 hours after the due date: 10 points (out of 100)
3. We will not accept your report and/or code after 2pm on Friday, 14 November.

In this last assignment, teams of six students develop an extended prototype as described below. You are strongly encouraged to **carefully read all sections,** since your grade for this assignment is highly dependent on following requirements. Write a project report organizing descriptions according to the format given on Page 8.

1. General requirements for SPA prototype

You are to develop an SPA prototype, which is a fully operational mini-SPA. This prototype should allow you to enter a source program (written in a subset of SIMPLE) and some queries (written in a subset of PQL). It should parse the source program, build some of the design abstractions in PKB, evaluate the queries and display query results.

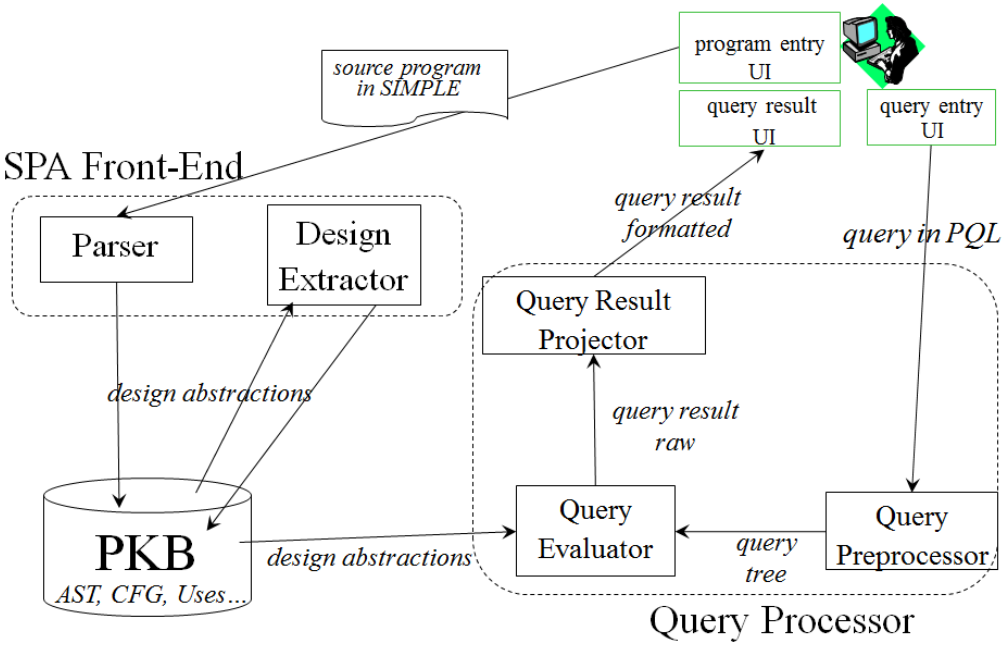
The idea of the prototype is to let you see the main SPA components in simplified form. Your solution should comply with the SPA architecture discussed in class and described in the handbook.

Even though some of the functionalities may be quite small, organize your code so that source files and directories clearly correspond to SPA architecture.

Use naming conventions to make this correspondence visible. In particular, implement SPA components and design abstractions in separate C++ source files and directories. Each of the design abstractions must be implemented in separate source files (.cpp), and its concrete APIs (public interfaces) should be defined in the corresponding header file (.h). All the communications between SPA components and design abstractions should be done via their concrete APIs. Use the same symbolic names for data types in abstract APIs and corresponding implementation classes. 'typedef' can easily assign specific C++ data type to symbolic names.

Integrate Autotester with your program. Use Autotester to reuse test cases and automate testing. This will save you a lot of time.

The above are compulsory requirements for organizing your C/C++ programs for SPA. You are not allowed to use parser generators or additional libraries, such as Bison/Flex/Boost, for parsing SIMPLE source.



2. The scope of prototype implementation

2.1 Implement the following subset of SIMPLE:

NAME : LETTER (LETTER | DIGIT)*
INTEGER: DIGIT+

procedure : 'procedure' proc_name '{' stmtLst '}'
stmtLst : stmt+
stmt : assign | while
while : 'while' var_name '{' stmtLst '}'
assign : var_name '=' expr ';'
expr : expr '+' factor | factor
factor : var_name | const_value
var_name : NAME
const_value : INTEGER

2.2 PKB contents

Design abstractions: AST (with Follows, Parent), Modifies, and Uses (for statements)
Others: You may include additional data structures that you find useful in implementing SPA.

2.3 PQL queries and Query Processor for the prototype

Queries include optional **such that** and **pattern** clauses. Each of these clauses can appear at most once in a query. For example:

assign a; while w;
Select a **such that** Follows (w, a) **pattern** a ("x", _)

such that clause can involve the following relationships:

Follows, Follows*, Parent, Parent*
Modifies, and Uses (for statements)

pattern clause can contain pattern specification for assignments of the form: a (entRef, expression-spec), where:

entRef : synonym | '_' | '"' IDENT '"'
expression-spec : '_' '"' factor '+' factor '"' '_' | '_' '"' factor '"' '_' | '_'
factor : var_name | const_value
var_name : NAME
const_value : INTEGER

Query Evaluator should produce results for any query that is correct according to PQL definition. In particular, in case a query refers to program line numbers or names of variables that do not exist in a SIMPLE source, Query Evaluator should return an empty result. You may also print informative error message for your own convenience.

Grammar definition of PQL subset for the prototype (excerpt from handbook, Appendix A):

IDENT : LETTER (LETTER | DIGIT | '#')*
synonym : IDENT
stmtRef : synonym | '_' | INTEGER
entRef : synonym | '_' | '"' IDENT '"'
NAME : LETTER (LETTER | DIGIT)*
INTEGER : DIGIT+

select-cl : declaration* '**Select**' synonym [suchthat-cl] [pattern-cl]
// [a] mean 0 or one occurrence of 'a'
declaration : design-entity synonym (',' synonym)* ';'
design-entity : 'stmt' | 'assign' | 'while' | 'variable' | 'constant' | 'prog_line'

suchthat-cl : '**such that**' relRef

relRef : ModifiesS | UsesS | Parent | ParentT | Follows | FollowsT
ModifiesS : 'Modifies' '(' stmtRef ',' entRef ')'
UsesS : 'Uses' '(' stmtRef ',' entRef ')'
Parent : 'Parent' '(' stmtRef ',' stmtRef ')'
ParentT : 'Parent*' '(' stmtRef ',' stmtRef ')'
Follows : 'Follows' '(' stmtRef ',' stmtRef ')'
FollowsT : 'Follows*' '(' stmtRef ',' stmtRef ')'

pattern-cl : **'pattern'** syn-assign (entRef, expression-spec)
// syn-assign must be declared as synonym of assignment (design entity 'assign').
entRef : synonym | '_' | '"' IDENT '"'
expression-spec : '_' '"' factor '+' factor '"' '_' | '_' '"' factor '"' '_' | '_'
factor : var_name | const_value
var_name : NAME
const_value : INTEGER

3. Suggested project phases

1) At the start: arrange one or more meetings to set up overall plan for the prototype development:
   a) Allocate responsibilities to team members
   b) Set up communication channels
   c) Agree on development process (e.g., whether or not do mini-iterations)
2) Write SPA system acceptance test cases. Your SPA prototype should pass these system acceptance tests at the end of the term. Each test case should consist of a description of purpose, SIMPLE source, PQL query (or queries) and their expected results (so that you know if your SPA produced correct results or not).
3) Develop SPA prototype
4) At the end:
   a) Run system acceptance tests
   b) Write project report
   c) Prepare presentation

4. Mini-iterations

Develop the SPA prototype in mini-iterations. At the end of each iteration, you will deliver a fully operational mini-SPA that you can test. This will allow you to phase difficulties. In addition, you will reduce the risks, as you will be guaranteed to deliver some SPA functions, even in case if you do not develop the complete prototype.

Here are suggested mini-iterations for the SPA prototype:

4.1 Iteration 1

SIMPLE: Procedure with assignment statements of the form x = y; (no expressions)
PKB: AST, VarTable, Follows, Parent
PQL: **such that**: Follows, Follows*, Parent, Parent*

4.2 Iteration 2

SIMPLE: Complete as required for the prototype
PKB: AST, VarTable, Follows, Parent
PQL: **such that**: Follows, Follows*, Parent, Parent*

4.3 Iteration 3

PKB: AST, VarTable, Follows, Parent, Modifies and Uses (for statements);
PQL: **such that**: Follows, Follows*, Parent, Parent*; Modifies and Uses (for statements); **pattern**

Suggested activities in each mini-iteration:

1) At the start: Iteration planning meeting
   a) Identify highest priority SPA features to be implemented
   b) Distribute tasks among team members and identify
   c) Plan meetings
2) Write system acceptance tests mini-SPA should pass at the end of the iteration
3) During development:
   a) Write and perform unit tests during development
   b) Perform integration testing as often as possible
4) At the end:
   a) Perform system acceptance tests
   b) Document mini-SPA
   c) Document any outstanding problems.

5. Testing your prototype

We will test your SPA at the end of the project using **AutoTester**. Plan for testing (and other quality controls such as reviews), and let quality control be an integral part of your implementation approach. The documentation and AutoTester files that need to be integrated with your submission can be found here: http://www.comp.nus.edu.sg/~cs3201/Tools-Lab/AutoTester.html. Alternatively, you may start your implementation from the sample Visual Studio 2010 solution that can be found here: http://www.comp.nus.edu.sg/~cs3201/Tools-Lab/AutomaticProjectTesting.html.

1) Plan to spend 1/3 of your effort on testing.
2) Store your test cases so that you can replay them as needed. Use **AutoTester** for regression testing (automatically replaying system tests).
3) Test your parser on a variety of SIMPLE sources.
4) Write system acceptance tests before you start development of the SPA prototype. Your SPA prototype should pass these system acceptance tests at the end of the term.
5) Test each type of query conditions (e.g., Follows, Modifies and pattern) and various combinations of query conditions (e.g., Follows + pattern and Modifies + pattern) that are allowed in the prototype.
6) Write unit tests as soon as you start writing code. Some tests may be written even before you write code, based on requirements.
7) Use UML sequence diagrams to plan for integration testing, and to monitor the progress of testing. Each arrow in sequence diagram corresponds to an interaction among SPA components and design abstractions stored in PKB that should be given attention during testing. Plan to perform integration as frequently as you find practical, at least at the end of each week.
8) After each mini-iteration, you will have a mini-SPA that can parse a subset of SIMPLE and answer some PQL queries. Prepare tests to check if mini-SPAs meet requirements in advance. Each test should consist of source code in SIMPLE, query, and expected answer.

6. Bonus points

Be sure that you implement well-tested basic functionality of the prototype before you attempt any extension for bonus points.

There are plenty of possible extensions as your prototype implements only a small subset of SIMPLE and PQL. On SPA Front-end side, you could extend SIMPLE and/or add more design abstractions. If you extend SPA Front-end, be sure that you also extend PQL and Query Processor accordingly. On Query Processor side, you could add **and** in query specifications.

Discuss any ideas for bonus points with instructors.

7. A check list of implementation issues

This section is not a sequence of development activities. It is a check list of issues you should address, for your reference only.

7.1 SPA Front-end / PKB

1) Parser for the required subset of SIMPLE. Samples of Parser's pseudocode are given in the handbook. You need extend these samples to cover the SIMPLE subset for the prototype.
   a) The predictive (recursive descent, top-down) parser given in the handbook is the simplest solution for the prototype.
   b) When your parser encounters the first error, it should print meaningful error message and terminate execution. Refer to section "Parsing SIMPLE" in "Technical Tips" in the handbook. Production parsers perform error recovery which allows them to continue parsing after finding an error. Your parser does not have to do error recovery.
   c) When your parser encounters an error in the source program, it should print meaningful error message and terminate parser execution.
2) VarTable and VarTable generation actions:
   a) Based on the abstract VarTable API, write a concrete VarTable API, i.e., a public interface for a class implementing VarTable.
   b) Choose a data representation for VarTable, justify your choice, and implement its public interface operations.
3) AST and AST generation actions
   a) Based on the abstract AST API, write a concrete AST API, i.e., a public interface for a class implementing AST.
   b) Implement AST generation actions in Parser.
   c) Choose a data representation for AST, justify your choice, and implement its public interface operations.
4) Relationships Modifies and Uses for statements:
   a) Based on the abstract Modifies API, write a concrete Modifies API, i.e., a public interface for implementing Modifies.
   b) Choose a data representation for Modifies, justify your choice, and implement its public interface operations.
   c) Repeat a) and b) for Uses.

7.2 Query Processor

1) Write down validation rules for queries. Unnoticed errors in queries will cause you lots of problems during Query Processor implementation.

2) Split Query Processor into Query Pre-Processor and Query Evaluator.
3) Query Pre-processor parses and validates query, and generates internal query representation, called query tree. Implement Query Pre-processor in a similar way as the parser for SIMPLE.
4) Query Evaluator receives the query tree and evaluates it, consulting PKB when necessary.

7.3 User Interface

Adopt the simplest solution for user interface. You can make use of AutoTester as your user interface.

Project Report Format

Report evaluation criteria:

1) Readability
2) The scope – Do you cover all important issues?
3) The level of details – Do you describe your solutions providing enough (and not too much) details? A description which is too abstract is not informative, whereas one with too much detail makes it difficult to see the main points.
4) PKB API – Complete set of documented APIs **for the whole SPA** as described in the handbook for CS3201/CS3202 (not only for the prototype).

1. Development plan

Plan the activities for each team member. *Activity* is a smaller work unit than task. For example, you may have an activity such as testing the interface to AST. Tasks consist of activities. Define activities of a size that you feel comfortable planning with.

	Iteration 1				
Team member	Activity 1	Activity 2	Activity 3	Activity 4	…
Mary	*		*		
John		*			
Suzan					
Jack					

2. Scope of the prototype implementation

Indicate whether you have met basic requirements for the prototype described in the assignment.

Describe any bonus features and how you tested them.

3. Documentation of abstract APIs

Include a complete list of APIs of all design abstractions (similar to what you wrote in Assignment 3), not only the ones you have implemented in the prototype.

4. UML diagrams

Include any UML diagrams that you found useful. For each diagram, explain how you used it (e.g., in project planning, communication, test planning or in other situations), and comment on the value the diagram added to your project.

5. Comments on design decisions

We will pay much attention to evaluating design decisions only in CS3202. For the prototype, it is up to you if and how you comment on your design.

6. Coding standards and experiences

1) State coding standards adopted by your team.
2) Comment on what you have done to enhance the correspondence between abstract APIs (specified in Assignment 4), and their concrete API counterparts (C++ classes).

7. Query processing

7.1 Validating program queries

Describe query validation rules. An example of query validation rules is: "Checking if all relationships have correct number and types of arguments". DO NOT provide procedural description (pseudocode) of how Query Pre-processor checks the rules. Very briefly (1/2 page maximum), describe how you encode validation rules, and how you perform query validation.

7.2 Design and implementation of Query Evaluator

Describe the data representation for program queries (query tree).

Describe how your Query Evaluator works.

8. Testing

Describe your approach to testing.

Comment on the testing experience gained from this project.

Be sure that you understand the role of the given AutoTester, integrate it with your SPA, and use it to automate regression testing throughout the project.

8.1 Describe your test plan

8.2 Provide examples of test cases of different categories (this is in addition to the system acceptance tests you submit with your SPA code)

1) Unit Testing
   a) Provide TWO samples of specific unit test cases for SPA Front-end, and two for Query Processor.
   b) If you used assertions, describe how and show examples.
2) Integration Testing
   a) UML sequence diagrams show communication among SPA components. Use sequence diagrams to plan integration testing and indicate which integrations you have tested.
   b) Provide TWO sample integration test cases.
3) System (Validation) Testing
   a) Provide TWO sample test cases.

Document each test case in a standard way as follows:

Test Purpose: Explain what you intend to test in this test case.

Required Test Inputs: Explain what program module (or the whole system) you test and what input must be fed to this test case.

Expected Test Results: Specify the expected results for this test case.

9. Discussion

Free format. Here are examples of issues that you might address: (Do not feel obliged to address all of them and you can discuss issues not mentioned here.)

1) What worked fine for you? What was a problem?
2) What would you do differently if you were to start the project again?
3) Comment on the experience gained in this project in respect to:
    a) Working in the team,
    b) Complexity of the SPA problem and program solution,
    c) What you have learnt in this project course.
4) Comment on the tools used for the project
    a) Were the recommended tools useful?
    b) What other tools did you use (if any), and in what ways were they useful?
    c) What were the problems you faced when using each tool?
    d) In which areas would you like to have had more tool support?
5) What management lessons have you learned?
6) Suggest how this project course can be improved.

10. Comments on the handbook

Let us know your comments on the handbook, and how it can be improved.

--- The End ---