Project Iteration 1

Submission Instructions

***Deadline*: Thu, 3 Feb, 10pm.**

Submission:

1. **Report**: Create a PDF file, named **<XX>.pdf**, where <XX> is your student number. (Project report format guidelines are provided in **Appendix A**.) The PDF file should include **your full name, email address and phone number**. (We may need to contact you during testing).
2. **Code**: Create one folder, named **Code_<XX>**, where <XX> is your student number. It should contain a README file and the Visual Studio 2019 Solution / CMake code folder with all your code (integrated with Autotester) or the that we will build, run and test.
3. **Test Cases**: Create one folder, named **Tests_<XX>**, where <XX> is your student number. It should contain all your system acceptance test cases for at least **ONE** SIMPLE program and **FIVE** queries.
4. Place the report and the two folders in another folder named **<XX>**, and archive it in a file named **<XX>.zip**. Submit the archive to **LumiNUS -> TIC2003 -> Files -> Student Submission -> Iteration 1.**
5. Detailed guidelines for LumiNUS submission are given in **Appendix B**.
6. You should submit only **one** copy of the report and code. For multiple submissions received from the same student, we will mark the most recent one.
7. You are strongly advised to **carefully read all sections** of this document to understand the requirements. When in doubt, contact the teaching team as soon as possible for clarifications.
8. This iteration accounts for 15% of the final grade. See **Section 6** for the rough marking scheme.

Late Submission Policy:

- Penalty for **report** submission within 24 hours after the deadline: 5 points (out of 100)
- Penalty for **code** submission within 24 hours after the deadline: 15 points (out of 100)
- We will **not** accept your report and/or code after **Fri, 4 Feb, 10pm**.

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 SIMPLE) and some queries (written in a subset of PQL). It should parse the source program, build some of the design abstractions in Database, evaluate the queries and display the query results.

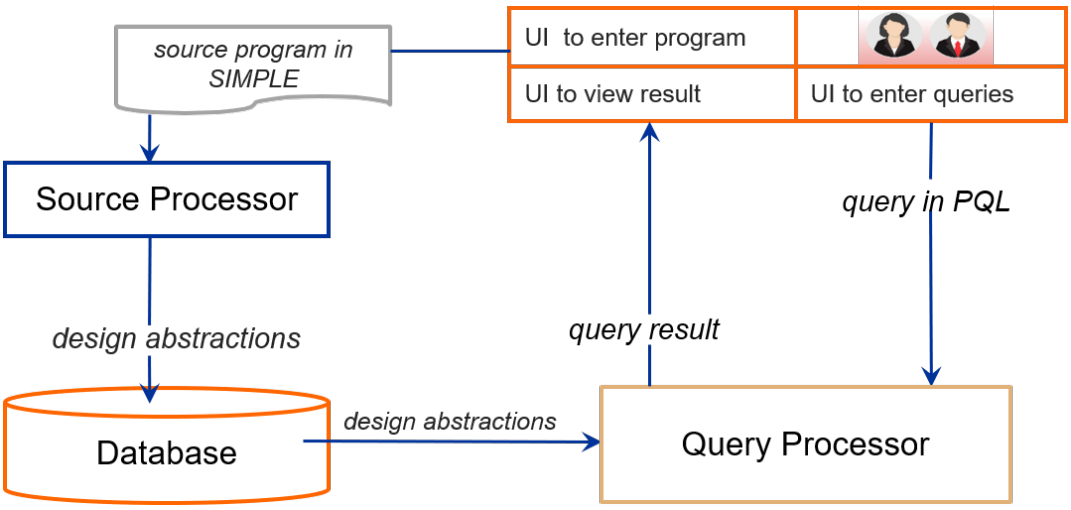
The idea of the prototype is to let you see the main SPA components in simplified form. Your solution should comply with the SPA architecture described in the course materials (i.e., lecture notes and handouts).

Even though some of the functionalities may be quite small, organize your code so that source files and directories clearly correspond to the 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 public interfaces should be defined in the corresponding header file (.h).

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 <u>not</u> allowed to use parser generators or additional libraries, such as Bison/Flex/Boost, for parsing SIMPLE source.



2. The scope of Iteration 1 (prototype) implementation

2.1 SIMPLE

Lexical tokens:

LETTER : A-Z | a-z -- capital or small letter

DIGIT : 0-9

NAME : LETTER (LETTER | DIGIT)* -- procedure names and variables are strings of letters, and digits, starting with a letter

INTEGER: DIGIT+ -- constants are sequences of digits

Grammar rules:

program: procedure

procedure: 'procedure' proc_name '{' stmtLst '}'

stmtLst: stmt+

stmt: read | print | assign

read: 'read' var_name;

print: 'print' var_name;

assign: var_name '=' factor ';'

factor: var_name | const_value

var_name, proc_name: NAME

const_value: INTEGER

2.2 Database

Program design entities: statement, read, print, assignment, variable, constant, procedure.

2.3 PQL

Queries contains only one declaration and one **Select** clause. For example:

assign a;

Select a;

Grammar definition of PQL subset for the prototype:

Lexical tokens:

LETTER: A-Z | a-z -- capital or small letter

DIGIT: 0-9

IDENT: LETTER (LETTER | DIGIT)*

NAME: LETTER (LETTER | DIGIT)*

synonym: IDENT

Grammar rules:

select-cl: declaration '**Select**' synonym

declaration: design-entity synonym ';'

design-entity: 'stmt' | 'read' | 'print' | 'assign' | 'variable' | 'constant' | 'procedure'

3. 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 in the Wiki under Tools.

- Plan to spend 1/3 of your effort on testing.
- Store your test cases so that you can replay them as needed. Use **AutoTester** for regression testing (automatically replaying system tests).
- Test your parser on a variety of SIMPLE sources.
- Test each type of query conditions (g., Follows, Modifies and pattern) and various combinations of query conditions (e.g., Follows + pattern and Modifies + pattern) that are allowed in the prototype.

4. A checklist of implementation issues

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

4.1 Source Processor

- The grammar of SIMPLE is relatively straightforward so there is no need to use any parser libraries. Simple approaches, such as Recursive Descent and regular expressions, should already work.
- You may assume that the input program is valid (i.e., there is no need to perform validation). However, it might still be useful to print diagnostic messages for debugging.

4.2 Database

- Database needs to be initialized to create the empty tables at the beginning and populated with the appropriate information using a local database and SQL statements.
- It is recommended to create a dedicated class to execute the SQL statements.

4.3 Query Processor

- It is recommended to split Query Processor into Query Parser and Query Evaluator

- Query Parser parses each query, and then generates an internal representation for the query. Implement Query Parser in a similar way as the parser for SIMPLE.
- Query Evaluator receives the internal representation of the query and evaluates it, consulting Database when necessary.
- You may assume that the input query is valid (i.e., there is no need to perform validation). However, it might still be useful to print diagnostic messages for debugging.

4.4 User Interface

There is no need to implement any User Interface since your system is evaluated using the AutoTester. Focus on developing and testing your SPA functionality.

5. Report

Follow the Project Report Format given in **Appendix A.**

There is no page limit for the report, and the ideal length highly depends on your writing style. When in doubt, please consult the teaching team for advice.

Avoid repeating material contained in the Handout. If necessary, simply make a reference to the specific section of the Handout.

6. Rough marking scheme

Iteration 1 accounts for **15%** of the final grade. The maximum mark for Iteration 1 is **100** marks: **80** from AutoTester Evaluation and **20** from Report. (Weightage is subject to change without prior notice.)

6.1 AutoTester Evaluation

All submissions are evaluated by AutoTester with a short but complicated source program and multiple queries.

The failed test cases will be analyzed to identify the number of unique errors and determine the severity of the errors. For example, if a system outputs an extra punctuation in the output, which causes multiple test cases to fail, it is still counted as one minor error since it is just an output (but not logical) issue. In contrast, if a system goes into an infinite loop for a particular type of queries due to a logical issue, it is counted as one major error.

6.2 Report

The reports are marked based on the following criteria:

- Overall quality - Make your report clear and readable.
- Content - Cover all important issues as outlined in **Appendix A**. Include any other content information you consider useful for understanding your prototype. Make it as concise as possible but complete. Use good examples to illustrate your design.
- Level of details - Provide enough (but 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.

6.3 Additional Notes

- The marking process is not mark-for-mark but ranking-based. In other words, having 1 unique minor error from AutoTester Evaluation or 1 piece of information missing from the report does not directly cause a certain number of marks to be deducted. Instead, the issues identified will be used to rank the submissions and the marks will be derived from the ranking. The teaching team will make sure that all submissions are evaluated in a consistent and fair manner.

Appendix A: Project Report Format

1. Scope of the prototype implementation

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

2. SPA design

2.1 Overview

Give an overview of your main SPA components and briefly describe the way they interact. Do not repeat information from the course materials.
Use diagrams when needed.

2.2 Design of SPA components

Describe your main SPA component in detail. Include at least the following information:

- **Source Processor:** Describe how you parse SIMPLE source code. Explain when and how your Database is populated.
- **Database:** Explain the structure of your Database, as well as when and how it is used.
- **Query Processor:** Explain how you parse the queries, the data representation for the queries, and how you evaluate the queries.

3. Testing

Describe how you have designed your test cases to cover the SPA requirements and give examples.

Appendix B: Guidelines for LumiNUS Submission

You are responsible for ensuring that your submission adheres to the guidelines.

How to submit:

Create a folder called **<XX>** with your code, documentation and supplementary information. Archive this folder in an archive called **<XX>.zip**, where <XX> is your student number.
Submit the archive via LumiNUS:

- LumiNUS -> TIC2003 -> Files -> Student Submission -> Iteration 1

What to submit:

Please include the following inside the folder **<XX>**:

1. **<XX>.pdf**: Your report, in PDF format.
2. **Test Cases**: Create one folder, named **Tests_<XX>**, where <XX> is your student number. It should contain all your system acceptance test cases for at least **ONE** SIMPLE program and **FIVE** queries.
    a. **<TEST-DESCRIPTION>_queries.txt**: Queries file that is written in the AutoTester format.
    b. **<TEST-DESCRIPTION>_source.txt**: Source file for SIMPLE program, corresponding to the queries file.
3. **Code_<XX>: Visual Studio solution folder** containing (at least):
    a. **source**: Folder containing the source code of SPA.
    b. **sln**: Visual Studio 2019 Solution file (and its additional project files and directories). The solution (and all project included) should build without errors in "Release" configuration.
    c. **txt**: Readme with basic instructions on how to compile and run the AutoTester on your SPA, the name of your solution that has to be built, the path to the executable produced. Mention also which Visual Studio version to use and which configuration to use (Release /Debug). In case no VS version and configuration are mentioned, we will build and run using VS2019 in Release configuration. Do ensure that the solution files from your submission build and produce the correct executable.
    d. **Contact.csv**: CSV file containing your full name, email and phone number.

What not to submit:

Please do not include any unnecessary files such as temporary files or binaries:

1. **.vs**: Temporary files generated by Visual Studio.
2. **Debug**: Debug binaries generated by building and linking the project.
3. **Release**: Release binaries generated by building and linking the project.

Important notes:

- We use **AutoTester** to test your submission. Ensure that your code is integrated and works correctly with AutoTester.
- We will test your implementation with a set of SIMPLE source codes and PQL queries. Failed test cases will be discussed at your first consultation after the submission.
- Please note that the source code and queries are **case-sensitive**, so check that your SPA accepts and computes results according to correct casing (see AutoTester examples). In case we cannot compile your code or run SPA at all, we will contact you so that you can fix the problem. For that, please do not forget to include your contact information (see above) and be available after submission.
- If the SPA does not return an answer for a PQL query after **5,000ms**, we assume an 'infinite loop' situation and the test case is considered failed.