CSC444F Software Engineering I

Tutorial Assignment 2

This assignment is handed out during the tutorial of week 4 (Week of 24/9/2001)
This assignment is due in ***three*** weeks, at **the start of your tutorial on Monday 15/10/2001.**

☞ ***Note****: An extra week is included because of Thanksgiving!*

To avoid late penalties, submit it to your TA within the first half-hour of the tutorial.

Penalties

Reports submitted up to 48 hours late: -50%.
Reports submitted more than 48 hours late will not be graded.

Grading Scheme

This assignment constitutes 10% of your grade for the course.
This report is a team assignment. Each team should submit a single report, and all members of the team will receive the same grade. *See the course orientation handout for details on team grading*. You should include a short statement about which team members wrote which parts of this assignment. If some parts were joint efforts, make this clear.

Content

The assignment is to document your phase 1 module and describe how you plan to test it. *Do not submit your program code*. You should submit the following (parts A and B have equal weight):

Part A: Design Documentation

1) A procedural abstraction definition for each procedure in your CSCI. Your procedural abstractions should demonstrate that you:
   - documented *all* the conditions under which your procedures will work correctly;
   - documented all assumptions made in writing the procedures;
   - can write postconditions in a declarative style;
   - used side effects only where appropriate and documented them clearly;
   - used exceptions where they were needed and documented them clearly.

   Note: if your program does not use any exception handling, you should explain why not.
2) A data abstraction definition for each type of data object in your CSCI. Procedures that are the methods of a data abstraction should be defined using procedural abstractions, and placed within the definition of the data abstraction. Your data abstractions should demonstrate that you:
   - can design good data abstractions, and choose appropriate methods to provide;
   - can document data abstractions clearly, including documenting each method provided by the data abstraction
   - can write specifications for your data abstractions that do not refer to implementation details.

   If you did not define any data abstractions for your CSCI you need to give a convincing reason why not.

Part B: Test plans

3) A dependency graph showing the components within your CSCI. Include on your graph components of other CSCIs **ONLY** if they are called by components of your CSCI. Your dependency graph should be, *complete* (with respect to your documented procedural and data abstractions), *correct*, and *clearly laid out*.
4) A test plan for your phase 1 CSCI. Refer to the dependency graph to indicate the order in which units are to be tested and integrated, and to indicate where stubs and drivers are required. Your plan should:
   - clearly describe each test to be carried out, including, for each test, which unit (or set of units) is being tested, the precise inputs and outputs, and the expected results were.
   - include an adequate set of tests to cover the main operations (hint: use both black and white box analysis to help you choose test cases, and explain your test selection rationale)
   - describe clearly the order that the tests should be performed in, and what integration steps are to be performed between tests
5) A status report on your testing process. For those tests that you have completed, indicate which tests passed and which failed. If your CSCI passed all the tests, explain why you think your testing strategy is sufficient. If you have not yet completed testing, give a schedule and resource estimate for the remainder of your testing process.

Background information

To help you complete this assignment, you may find the following useful:

A *procedural abstraction* has 5 elements:

a) *Name and input/output parameters* – defines how the procedure communicates with the rest of the program;
b) *Requires* – defines the conditions under which the procedure will work (i.e. its *preconditions*). A total procedure is one that works for all possible inputs, and therefore has no preconditions. For total procedures the requires clause is omitted;
c) *Effects* – defines what the procedure achieves (i.e. its *postconditions*). The effects clause should be written in a declarative style—they should ***not*** say "the procedure does this, then this, then this....", but should state properties (post-conditions) that are true at the end of the procedure;
d) *Modifies (or side-effects)* – defines any changes the procedure makes to its environment, such as changes to global variables, changes to the i/o streams, allocation/de-allocation of memory, etc. A pure function is a procedure that has no side effects. For pure functions, the *modifies* clause is omitted;
e) *Raises* – defines exceptions that the procedure may raise in response to error conditions. Even if the programming language does not explicitly provide an exception handling mechanism, it is good practice to design the procedure in such a way that error conditions (i.e. exceptions) are communicated back to the calling procedure. It is also good design practice to communicate error conditions back using a separate communication channel from correct results, so that the calling procedure can never mistake error conditions for real data.

A *data abstraction* is an encapsulated data type together with the operations available on that type. This is like an object in object-oriented programming, but is a design style that can be used in any programming language. A data abstraction has the following elements:

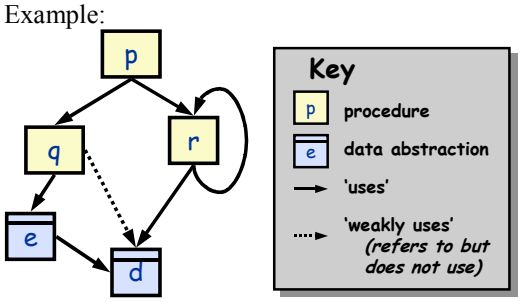
a) A name for the data abstraction
b) A short description of the data abstraction (in English)
c) A set of public operations (each defined as a procedural abstraction), classified as:
   - primitive constructors—these create new objects of the datatype
   - constructors—these take existing objects of the datatype and build new ones
   - mutators—these modify existing objects of the datatype. An immutable datatype has no mutators.
   - observers—these tell you information about existing objects of the datatype (without changing them).

Note that a data abstraction should not include private procedures that are only used internally. These should still be documented somewhere, using procedural abstractions, but do not form part of the definition of the data abstraction. Note also that the data abstraction should not include information about how the data type is actually stored (i.e. the nature of the rep type). This information may change if different implementations are used, but changing the rep type should not affect the public interface to the data abstraction. However, it is useful to document the rep type, and it is often useful to write a rep invariant for testing and debugging purposes.

A *dependency graph* is a graph showing which units within a program depend on other units. The graph is normally laid out so that that units that have no dependents are at the bottom, and units that are not depended upon by anything are at the top:

- all edges must be directed – the arrow indicates which way the dependency goes
- all nodes must be labeled with the name of the unit
- draw only one edge between any two nodes (no matter how many times the unit is called)
- recursive procedures (or data abstractions) use themselves.

Example:



A *software test plan* describes the order in which unit testing (testing a program unit on its own) and integration testing (testing that a group of units work together properly) will be carried out. Because unit testing is more thorough, it is preferable to test each unit separately, before integrating it with other units. However, this is not always possible, as it may be hard to fully test a unit in isolation from its dependents. Hence, it is normal to adopt either:

- a bottom up strategy—start at the leaf nodes in the dependency graph, test these as individual units, and then work your way up the graph integrating and testing as you go;
- a top down strategy—start at the top node and test it using stubs for the untested dependents. Then work down the graph, repeating this procedure at each level.

Often it is possible to use some combination of these strategies when determining what order to test and integrate your units. In either case, when testing a particular unit (the "unit-under-test") you may need the following:

- a test driver sets up the environment and makes a series of calls to the unit-under-test. Effectively it mimics the units that call the unit-under-test, if these units are not yet ready for integration.
- a test stub acts as a dummy for a dependent of the unit-under-test. Effectively it mimics the units that the unit-under-test calls if these are not yet ready for integration. A simple way to provide a stub is to provide a dummy function with the same name and parameters as the real dependent unit, but which merely asks the tester to enter a sensible return value to be passed back to the unit-under-test.

Finally, test cases for each unit can be chosen using:

- Black Box testing—the development of test cases purely from the specification (ie. without looking at the code). The specification is analyzed to determine a set of tests that cover all the different functions that the unit should perform.
- White Box testing—the development of test cases from an analysis of possible paths through the program code. The code is analyzed to determine a set of tests that cover all branches at each choice point. A test set is *path complete* if it exercises each path through the code at least once.

Black box testing is geared towards ensuring that a unit meets its specification, while white box testing is geared towards making sure that all of the code is covered. A good test set for a unit will combine black box and white box test cases.