CSC444 Software Engineering I Fall 2002

Assignment 2: design documentation and test plans

last updated: Sept 9, 2002

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.

Note: the set of abstractions doesn't need to be *complete* - if the module is very large, it is okay to submit a sampling of the abstractions.

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
   - 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.

Note: the test plan doesn't need to be complete – it is okay to submit a sample set of tests, and a brief summary of the remainder.

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:

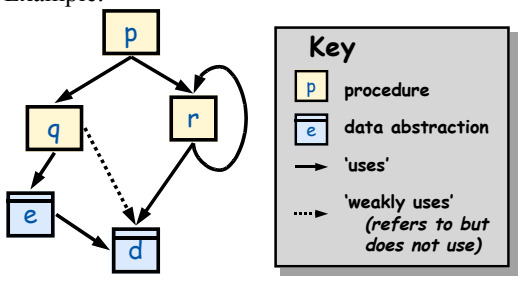
a) A name for the data abstraction
b) A short description of the data abstraction (in English)
c) A set of public operations (each defined as a procedural abstraction), classified as:
   - primitive constructors—these create new objects of the datatype
   - constructors—these take existing objects of 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.

Marking scheme

Listed below are some of the things your TA will look for when marking this assignment. Use this list to check your work before you hand it in.

Note: a module written in Java might not have procedural abstractions other than the methods of a data abstraction – in that case, the questions about procedural abstractions will just apply to the methods of the data abstractions

Note: a module might have no data abstractions. In this case points will be awarded for a good justification of why they weren't needed, including a sound argument that the design is good, even without them.

Note: a module might not have any abstractions that have side effects, or exceptions. In either case, the relevant points can be awarded for evidence that they weren't needed. If they are merely omitted with no explanation these points cannot be awarded.

Abstractions

- Are the abstractions (both procedural and data) documented consistently? E.g. are they written and laid out in a consistent style?
- Are the procedural abstractions clear (readable), precise, and well laid out. A simple test is whether another programmer could use the procedure with confidence, just from reading the procedural abstractions.
- Are the 'requires' clauses used correctly? A 'requires' clause should document *all* the conditions under which the procedure works correctly, including any assumptions about the input parameters, and any conditions on global data. Check for obvious missing requires clauses (e.g. that pointers are not null, etc).
- Are 'effects' clauses used correctly? The effects clause should say what is true after the procedure is finished (e.g. what is the value of its return value, and any modified parameters). Note also that 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 (e.g. "the return value is...").
- Are side effects used sensibly and properly documented (as 'modifies' clauses)? A side effect is any change the program makes to its environment, including modified global variables, allocated/de-allocated memory, writing/reading from i/o streams, etc. Side effects should not be used indiscriminately (e.g. not too much use of global variables), but are necessary for some tasks. They should be documented separately from the effects clause. If there are no 'modifies' clauses in any of the abstractions, there must be an explanation for why they weren't needed to get this mark.
- Are exceptions used sensibly and documented?

- Are data abstractions used where appropriate? All the modules need to manipulate various kinds of structured data. Does the team's design include (and document!) some data abstractions for some of these? (If there are no data abstractions, to get this mark the team needs to explain why they decided they didn't need any).
- Are the data abstractions used correctly? Data abstractions are not the same as data structures. A data abstraction encapsulates an object and the operations (methods) that can be performed on it. If no operations are listed, or some necessary operations are missing, then the abstraction is not used correctly. Each operation must be documented using a *procedural* abstraction.
- Are the data abstractions abstract enough? No internal information about how the data is actually represented should be included. Also, private operations (e.g. procedures used internally for basic manipulation of the data structure, but not available for use by the rest of the program) should not be included in the documentation of the data abstraction. (If there are no data abstractions, the team needs to demonstrate that they understand when they *would* use data abstractions).
- Does the set of abstractions (procedural and data), taken together, represent good design? E.g. does the division of the program into classes & methods represent a good decomposition that supports data hiding and reduces coupling between classes? Conversely, do any of the abstractions look unnecessary, or add unnecessary complexity?

Test plans

- Is there a complete dependency graph? It must include each procedure and data abstraction listed in part 1.
- Does the dependency graph correctly indicate dependencies on other modules? In particular, it should show where procedures from other modules are called, but should not include any extra detail about the internals of other modules.
- Is the dependency graph clear, and well laid out? It should be immediately apparent from the graph which procedures are top level (e.g. the main procedure) and which are bottom level (don't call any other procedures). The dependency graph should not be split over multiple pages unless it *really* cannot be fitted onto a single page.
- Is there a description of the rationale used to select test cases?
- Does the test plan clearly describe each test? Could a tester follow the plan: does it say which stubs/drivers to use for each test, what inputs to give and what results to expect?
- Is the test plan sensible? For example, does it include a mixture of both black box and white box tests? Do the set of tests seem adequate to test the main operations of each procedure?
- Does the test plan indicate what order the tests should be performed in? ...and does it say which tests are performed on individual units (procedures) and which are performed on integrated groups of procedures?
- Does the test plan include some end-to-end integration tests of the whole module?
- If the testing is not complete, does the plan give an indication of how much longer it will take, based on how long it has taken so far? There should be some rationale for the estimation - just giving a number of hours or a date is not sufficient. Is there a clear status report on testing? The status report must clearly state which tests have been performed and which are still to be done?
- If the testing is complete, does the status report give an adequate summary of the findings of the testing process? In particular, if all tests were passed, does it give a realistic assessment how thorough the testing really was?