University of Toronto

Faculty of Arts and Science
Dept of Computer Science

CSC302S – Engineering Large Software Systems

April 2009
Instructor: Steve Easterbrook

No Aids Allowed
Duration: 2 hours
Answer all questions.

Make sure your examination booklet has 14 pages (including this one). Write your answers in the space provided.

This examination counts for 35% of your final grade.

Name: ____
(Please underline last name)

Student Number: ____

Question Marks

1	_____________	/20
2	_____________	/20
3	_____________	/20
4	_____________	/20
5	_____________	/20
Total	__________	/100

1. [Short Questions; 20 marks total]

(a) [Verification Techniques – 5 marks] Many companies report that inspection is much cheaper than testing as a way of identifying and removing bugs in their software. Why is this? Can such companies give up testing altogether?

Inspection is much cheaper than testing for two reasons:

- it tends to find errors earlier, because it can be applied before the code can even be run. The earlier errors are detected, the easier it is to correct them
- it helps to avoid the effort of debugging. When a test fails, it can take a lot of effort to pinpoint the problem. When errors are found in inspection, the inspection usually identifies the problem immediately.

However, companies using inspection cannot give up testing altogether because neither inspection nor testing are 100% effective. Using both together increases the chance that important errors are caught and corrected. Furthermore, applying inspection prior to testing decreases testing costs anyway, because it reduces the length of the test-debug-retest cycle.

(b) [Software Design – 5 marks] The Law of Demeter states that an object may not call methods of an object that is returned by another method call. Why is this a good design principle? How would you detect violations of this law in your code?

The law of Demeter reduces complex coupling between objects, by reducing how much one object needs to know about others. If an object was allowed to make method calls to any other object, regardless of how weakly they are related, then implicit knowledge about the entire structure of the software has to be encoded in each object. This increases dependencies between objects, and makes the whole system very hard to modify. By restricting method calls to only those objects that are directly related, the designer must take more care in deciding which objects should be responsible for which tasks. This tends to lead to decentralized control, which is more modular, and hence more robust and easier to modify.

A simple rule of thumb to detect violations of the law of Demeter is to count the 'dots' in a method call. If there is more than one dot, then a method call is being made to a returned object. Eg. foo.bar.dostuff(x) represents a call to the method dostuff of an object returned by the method bar of object foo.
This rule of thumb does not catch all violations because it is possible to make the method calls in several steps.

(c) [White Box Testing – 5 marks] A code coverage tool reports that during testing of your Java program, your test suite covered 100% of the statements in your program. However, your project manager tells you that your unit tests still do not sufficiently cover the code. What does she mean?

Assuming the tool is correct that every statement in the program has been executed at least once, this still can miss many types of error. Stronger coverage criteria include:

- Testing all data paths through the program – for example, every possible path from the point at which a variable is defined, to each possible place that variable is used. On some of these paths, errors might occur. But ensuring each statement is executed does not guarantee all paths are covered. (this is known as All-DU paths testing)
- Testing that every clause within a complex conditional can affect the outcome. If a change in value of one part of the expression in a conditional can never affect the outcome, this almost certainly represents a programminer error. (Testing for these is known as MC-DC coverage)
- Testing the result of every method call in every state the object can be in. For example, if an object implements a stack, testing each method when the stack is empty, full, partially empty can reveal errors.

In addition to these white box tests, which are based on different ways of covering the code, one would need to do black box testing, to check that the functions described in use cases work the way they should.

(d) [Software Estimation – 5 marks] What is 3-point estimation? Why would you use it?

Three point estimation involves asking a programmer to provide three different estimates of the amount of time it will take to perform a specific task: the expected time (e), the worst case (w) and the best case (b).
These three estimates are then combined by calculating (w+4e+b)/6.

3-point estimation has been shown to provide more accurate estimates than asking for a single number, because it forces the programmer to think more carefully about the upper and lower bounds, and hence to be more realistic.

As this type of estimation works best on small tasks, it could be used when planning an upcoming release on an agile development project, for calculating the estimated time needed to build each function requested by the user.

2. **[Black Box Testing – 20 marks]** Given the following use case, what set of black box test cases would you develop to test the software? Briefly describe each type of test case, and the reason why it is needed.

Use Case Name: Reserve Hotel

Summary: The user books a hotel for a given date, and receives confirmation by email.

Precondition: User has navigated to the hotel booking page, and has planned a travel itinerary.

Basic Course of Events:

1. Client enters city, arrival and departure dates, and number of guests.
2. System searches for available hotels, and displays a list of available hotels with base price for each, sorted by average customer rating. User may choose to re-sort the list by price or by quality (star rating).
3. User selects a hotel from the list.
4. System displays available room types with price for each.
5. User selects a room type and selects “Reserve it”.
6. User provides her email address.
7. User provides name, address and phone number.
8. User provides credit card details.
9. System makes the reservation and allocates a reservation number.
10. System displays confirmation screen showing the reservation number.
11. System sends email confirmation to the user.

Postcondition: Hotel Reservation is placed and user has received email confirmation.

Alternative Paths:

A1) At steps 2, 3 and 5 – User may return to step 1 and enter different itinerary details.

A2) At step 4 - User returns to previous step and selects a different hotel.

A3) At step 5 - User selects “Cancel” and exits the system.

A4) At step 6 - User has existing account and enters a password, then skips step 7.

Exception Paths:

E1) At step 2, no suitable hotels are found. System displays message and returns to step 1.

E2) At step 9, credit card information is invalid. System allows user to try re-entering credit card information a maximum of two more times. If these both fail, system cancels the booking and returns to step 1.

E3) At step 11, email message bounces. Hotel is notified that customer’s email address is incorrect.

The following types of black box tests could be used.

(1) test the normal path through the use case. This checks normal functioning

(2) test each alternative path. This checks that each variant of the test case works.

(3) test for options embedded in the use case – for example, in step 2, test various ways of re-sorting the search results. This checks that the use case works correctly whether or not these options are used.

(4) test each of the exception paths. This checks that the software fails gracefully when a problem occurs.

(5) test what happens if the precondition is not met. For example, in this case, we could explore what happens if the user has not yet planned an itinerary. This tests whether the software will still do reasonable things when users try to use it in unexpected ways.

(6) test for empty data field. For example, each time the user is asked to enter information, test what happens if the user leaves it blank. This tests that the software correctly deals with the missing information.

[Continue on next page]

(7) test for invalid data entry. For example, test what happens with credit card numbers that are too long or too short; test for invalid dates, non-existent cities, syntactically wrong email addresses, etc. This tests that the software correctly rejects invalid data.

(8) test for data entered in different formats. For example, test what happens if the user tries to enter different date formats (month-day vs. day-month), or if the user enters spaces in the credit card number, brackets around the area code for the phone number, etc. These test that either the system prevents non-standard formats in a helpful way, or accepts the non-standard format and does the right thing with it.

(9) test for boundary conditions on input data. For example a date that is only just a date and only just not a date (e.g. Feb 29 on leap years and non-leap years;

(10) test for overfull data in data entry fields. For example, enter a very long address, a very long user name, and so on. This checks that the software doesn't crash, and might even suggest additional functionality to help users who really do have long names or long addresses.

(11) perform stress tests – for example, hotel searches that generate huge number of hits (if this is possible), very large numbers of guests, users that spend hours making their choice, etc. This tests how the system performs under unusual situations.

(12) test for internationalization. For example, does the address entry work for various types of foreign address (e.g. states vs. provinces; addresses with strange postcodes, etc)

(13) do interference testing. For example, what happens if the web browser is closed or the network connection is lost partway through a booking (especially during step 9 in this use case). This tests that the user can find out whether the transaction was completed, and is not left in some strange state.

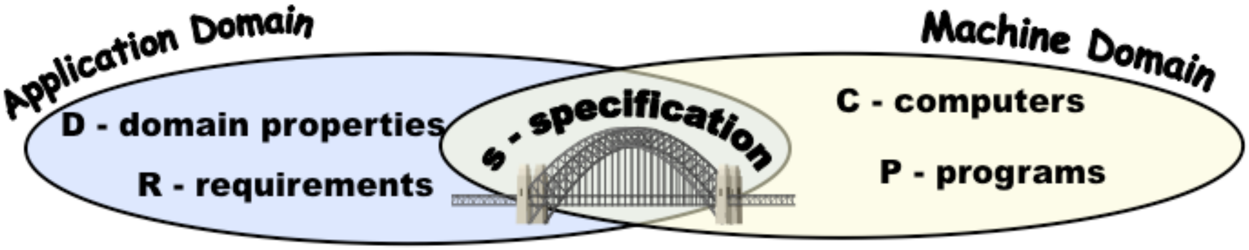
(14) test for browser compatibility. Check that the use case works on different browsers.

(15) test for browser interactions. For example, does the back button on the browser interact with the use case functionality in a sensible way?

(16) test the postcondition. Check that the postcondition is met on every case listed above that is not an exception case. This ensures that the correct result occurs in each test.

[Suggested marking approach: 2 marks for each distinct class of tests identified, and explained fully; 1 mark if it is not explained well. Up to a total of 20 marks – i.e. don't have to identify all the classes listed above; give credit for reasonable classes not given above, as long as they are black box test related to the use case

3. **[Requirements Analysis – 20 marks]** Jackson has proposed a conception of requirements analysis that distinguishes machine domain phenomena from application domain phenomena, as illustrated in the following diagram:



(a) *[5 marks]* Explain the distinction Jackson makes between Requirements, R, and Specifications, S. What additional properties should a Specification have in order to distinguish it from Requirements?

Requirements are any properties of phenomena in the application domain that a stakeholder would like to be made true by some new system. Requirements may refer to any phenomena, whether accessible to the machine or not: e.g. "cut down on errors in my accounts"
Specifications are a subset of requirements, covering only phenomena that are shared between the application domain and the machine – i.e. things that can be inputs and outputs to the software: e.g. "test whether numbers entered in the accounts are within a particular range"
To distinguish it from requirements, a specification must be expressed only in terms of inputs and outputs to the machine, i.e. phenomena that the software has access to.

(b) *[5 marks]* Give examples of R, D, S, C and P for a particular problem domain.

Many possible answers – marks given for demonstration of the key distinctions. The example given in class was:

For an aircraft
R – "Reverse thrust should be disabled unless the aircraft is moving on the runway"
D – "pulses from the wheel sensors will be received only when the aircraft is moving on the runway"
S – "reverse thrust should be disabled unless wheels sensor pulses are on"
P – the flight control software that takes commands from the pilot's controls and makes the aircraft do things, written according to specification S.
C – the flight computer (hardware) on the aircraft

(c) *[5 marks]* Using Jackson's conceptual model, explain why checking that a program meets its specification is not a sufficient test of fitness-for-purpose.

Even if a program meets its specification, the specification could still be wrong:

- The specification could fail to express the stakeholders' stated requirements;
- The stated requirements themselves may not adequately capture what the stakeholders really want/need;
- Assumptions made about the domain might be wrong, so that the reasoning that connects up the specification with the requirements is wrong.

In any of these cases, the program will not suit its purpose well, assuming that the purpose is to solve the stakeholders' problem(s).

(d) *[5 marks]* Suggest two techniques that you would use for checking whether statements of the Requirements, R, and Domain Properties, D, are valid.

Any of the following are acceptable:

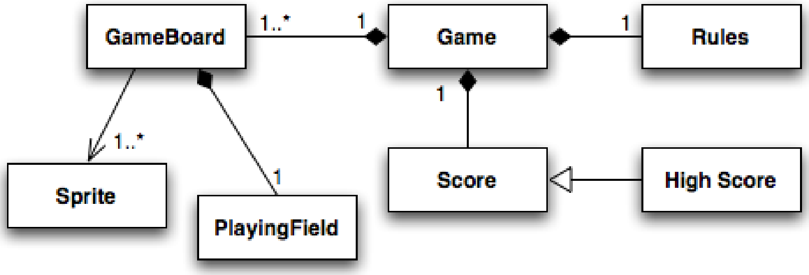
(1) Prototyping – build a quick and dirty version of the system based on your understanding of the requirements, and show it to the customer, to find out whether it really does solve the right problem, and whether the assumptions you made about the domain are correct.

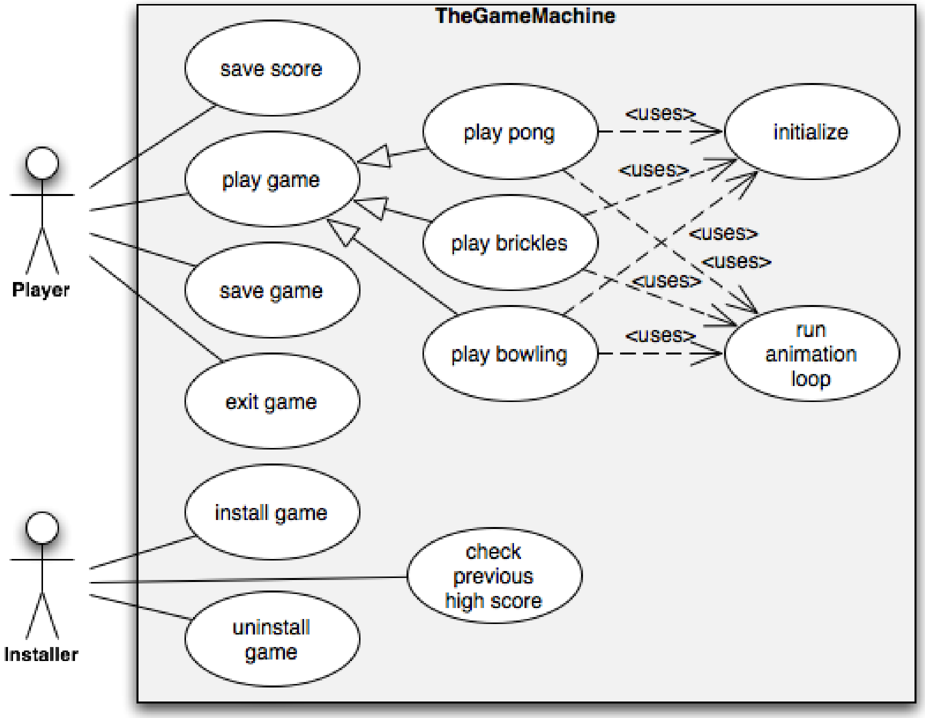
(2) Inspection – document all your statements about R and D, and then hold a formal inspection meeting, with key domain experts and stakeholders participating.

(3) Modelling – build a model (e.g. statechart, class diagram, etc) that captures your understanding of R and D, and then analyse it to check that it has the right properties, e.g. check it against scenarios or sequences of events that you know should or should not occur.

(4) Stakeholder interviews or questionnaires – ask stakeholders directly whether they agree with statements of R and/or D.

4. **[Robustness Analysis – 20 marks]** The following UML diagrams show an initial domain model and a use case diagram for a proposed computer game system, called TheGameMachine. Complete a robustness analysis for this system. Your analysis should include (at least) two robustness diagrams: (1) a first draft showing all the boundary ⊢◯, control ↻ and entity ◯ classes needed to implement each use case, and (2) a refined robustness diagram that combines some of these classes in sensible ways.





[Continue on next page]

[Question 4 continued]

5. **[Process Improvement – 20 marks]** The Capability Maturity Model (CMM) assesses the software development capability of a company as being at one of five levels: initial, repeatable, defined, managed, and optimizing.

(a) *[10 marks]* Describe each of the levels of CMM, and explain their rationale. What are the advantages and disadvantages of this approach to assessing software companies?

1) Initial level – the organization has no understanding of its software development processes. If projects succeed, that's purely down to either luck, or good people making the best of it. Very little project management used, e.g. no cost estimation and planning. This level is used to describe organizations that don't really know how they build software.
2) Repeatable level – the organization does apply a repeatable process to its software development projects, but this is dependent on individuals remembering what was done previously, and repeating practices that they believe worked. This level describes organizations that understand their own best practices, but don't manage them in any coherent way – it's up to individuals to take responsibility.
3) Defined level – The process used to develop software is defined in a qualitative (descriptive) way, and this defined process is used across the organization on all (or most) projects. The rationale for this level is that it's worthwhile for an organization to document its processes (although most would argue this is only a stepping stone the the next levels – that just documenting the processes on its own doesn't give much benefit).
4) Managed level – The process is defined in quantitative terms, and measurements are used to manage projects according to the defined process. The rationale for this level is that defined process models should greatly help managers in identifying things to measure about their projects, in order to keep them on track.
5) Optimizing – the organization regularly seeks to optimize the process by looking for improvements and continually feeding these back into the process definitions. The rationale for this level is that organizations need to institutionalize and support the learning process, so that rather than blindly following a defined process, everyone in the organization is looking for ways to make it better.

The advantages of this approach are that it helps everyone in the organization understand what process they are supposed to be following, and emphasizes that continuous process improvement is needed to ensure the software remains high quality.
The disadvantages are that it can be time-consuming and expensive to document and measure the processes, and once this is done, people tend to blindly follow what the documents say, rather than thinking things through. In the end, the defined processes can stifle innovation and individual responsibility.

(b) *[10 marks]* How well do the ideas of the CMM apply to smaller, more agile teams, and to individual developers? How do they compare with other approaches to improving software quality used in agile software development?

[scratch paper]

[scratch paper]

[scratch paper]