University of Toronto

Faculty of Arts and Science
Dept of Computer Science

CSC302S – Engineering Large Software Systems

April 2012
Instructor: Steve Easterbrook

No Aids Allowed
Duration: 2 hours
Answer all questions.

Make sure your examination booklet has 11 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) [Software Architectures – 5 marks] What are coupling and cohesion, and why are they important in software design? Suggest measurable properties of a software design that can be used as indicators of the amount of coupling and cohesion.

Coupling – a measure of the degree of interaction between software modules, e.g. the amount of knowledge each module needs to have about the other modules. Can be measured by looking at the number of method calls between modules, the number of parameters passed, etc.

Cohesion – a measure of how well the things grouped together in a module belong together logically. Could be measured by looking at the number of internal method calls within a module, etc.

These are important because they affect maintainability – if coupling is low and cohesion is high, it should be easier to change one module without affecting others.

(b) [Software Design – 5 marks] The Law of Demeter states that an object may not call the methods of an object that was returned as the result of 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] What are the advantages and limitations of using automated tools to measure code coverage during testing? If such a tool tells you that your test suite covers 100% of the statements in your program, does this mean you've tested everything?

Code coverage tools measure which lines of code are being executed when a test suite is run.
Advantages: they give an overall sense of what parts of the code are being tested by a given set of test cases, can point to missing test cases; can detect dead code.
Limitations: They report which lines of code were executed, but not what paths through the code have been executed. Hence can give a very misleading sense of the completeness of a test suite.

Many reasons why "100% coverage" doesn't mean you've tested everything because you might have missed:

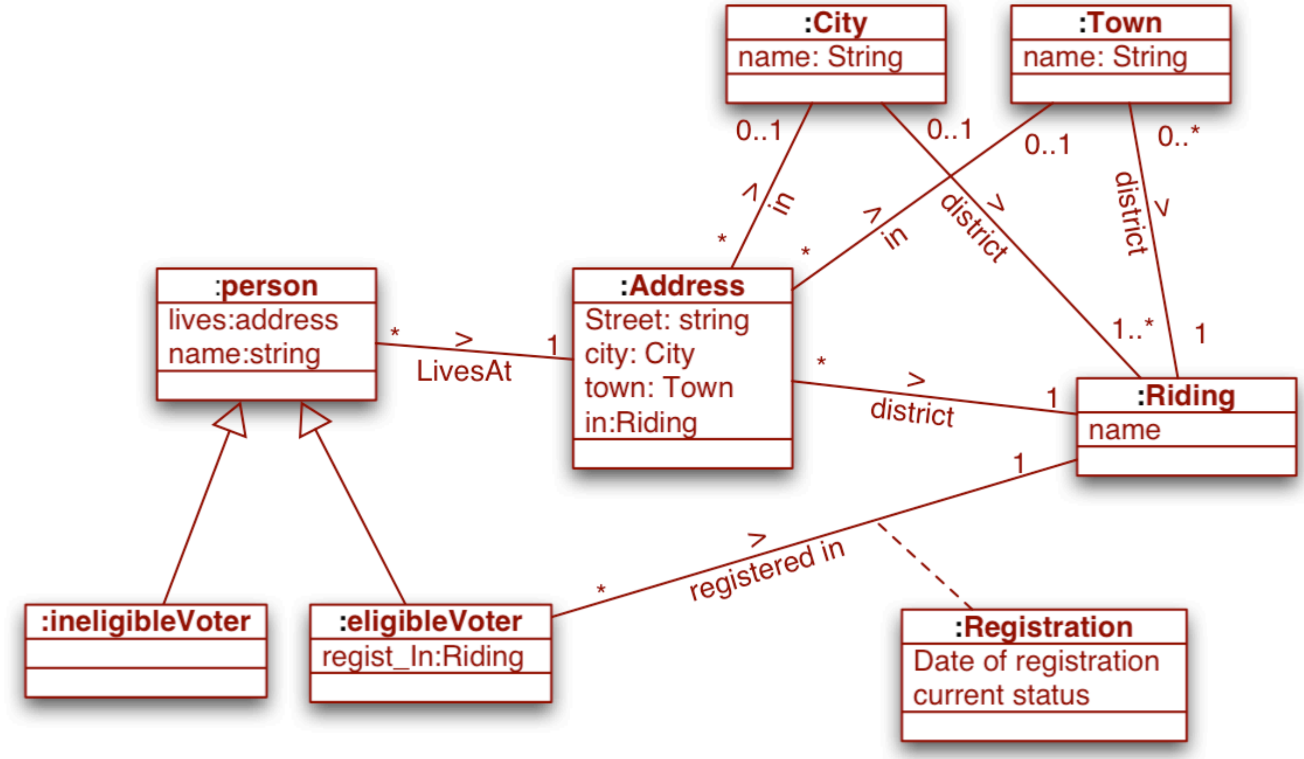
- alternative paths (e.g. zero passes through a loop, empty else statements, etc)
- how inherited code behaves in each subclass
- boundary cases, bad data inputs, recovery from errors, etc
- whether the code actually meets user requirements

(d) [Specifications – 5 marks] Project managers sometimes regard work put into writing detailed specifications as "gold plating", and claim that it is unnecessary as it doesn't contribute to producing program code. Under what circumstances is this view sensible, and under what circumstances is it foolish? In the latter case, how would you persuade such a manager that detailed specifications are essential?

This view is sensible for small projects where there is a well-understood problem to be solved, and where the programmers exploring the requirements will also implement the system (i.e. the specification does not have to bridge this gap).
This view is foolish for big projects, where there is a large team and lots of stakeholders. In this setting, proper communication among the team, and with the stakeholders is important, and the specification is used to make sure everyone understands the problem fully. Higher quality specs should result in easier integration and higher quality software.

If the list of benefits above doesn't convince the manager, then one could look for articles and research reports that describe the role of a high quality specification in large projects. One could also use anecdotal evidence from past projects (e.g. identify problems on a previous project that could have been avoided with a good specification).
[Give credit for any reasonable suggestions for how to persuade the manager]

2. **[Class Diagrams – 20 marks]** Draw a class diagram to represent the following description of domain objects involved in developing a registration system for voters in provincial elections:
"Each person can be either an eligible voter or an ineligible voter. Eligible voters are registered in only one electoral district (known as a 'riding'), and for each registration instance, details such as the date of registration and current status need to be stored. Eligibility is determined by address. Each person can be assumed to have one current address, and each address is in either a town or city. Some ridings contain several towns, while some large cities may contain more than one riding."
Be sure to include names and multiplicities on all associations, and state any assumptions you make. How would you modify your diagram to handle the following situations: (a) we may need to retain information on people who no longer live in this province; (b) some people own property in more than one riding, and hence apply for registration in more than one riding; (c) people with more than one property are required to designate one as their 'primary residence' to determine which riding they can vote in.



Note: Other variations are possible. The association class "registration" is a particularly elegant solution, but not particularly obvious.

Modifications:

(a) Requires that we can capture out-of-province addresses. Either make the association from address to riding optional (0..1), or add an extra class for "out of province address", perhaps as subclass of address.

(b) Allow more than one address to be associated with a person, in which case we'll also need to add an association from the Registration class to the address class, to indicate which address the registration is for.

(c) Add an additional class for "primary residence", subclassed from address, and *require* each person to have exactly one primary residence (other solutions are possible, but must enforce "exactly one" to get full credit)

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

Use Case Name: Withdraw Cash from ATM

Summary: *Bank Customer* uses the ATM to withdraw money from her bank account.

Preconditions: There is an active network connection to the Bank. The ATM has cash available.

Basic Course of Events:

1. The use case begins when *Bank Customer* inserts her Bank Card.
2. The Use Case for Validate User is performed.
3. The ATM displays the different functions that are available on this unit {withdraw cash, check balance, deposit funds}. In this use case the Bank Customer always selects "Withdraw Cash".
4. The ATM prompts for an account to use, from a list of three account types {Chequing, Savings, Other}.
5. The Bank Customer selects an account to use.
6. The ATM prompts for an amount.
7. The Bank Customer enters an amount.
8. Card ID, PIN, amount and account is sent to Bank as a transaction. The Bank replies with a go/no go reply saying if the transaction is ok, and adjusts the account balance if it is.
9. The Bank Card is returned.
10. The money is dispensed.
11. The receipt is printed.
12. The use case ends successfully.

Postcondition: Cash was dispensed; Card was returned; Balance on the user's account has been updated.

Alternative Paths:

A1) *Invalid User*: At step 2 – Failure to validate card. An "invalid PIN" message is displayed, the bank card is returned, and the use case ends.

A2) *Wrong Account*: At step 8 – If the selected account type is not associated with this card, an error message is displayed and the use case resumes at step 4.

A3) *Wrong Amount*: At step 7 – If the user enters an amount that cannot be dispensed with the bills in the ATM, a message is displayed indicating what bills are available, and asking the customer to re-enter the amount; Use case resumes at step 7.

A4) *Transaction Declined*: At step 8 – if the transaction is declined by the Bank, a "declined" message is displayed, the bank card is returned, and the use case ends.

A5) *Premature Exit*: At any step prior to step 8, if the user presses "Cancel", the bank card is returned and the use case ends.

Exception Paths:

E1) *Comms Failure*. At step 8, If there is no response from the bank within 3 seconds, the ATM will retry up to 3 times. If there is still no response, an error message is displayed, the user's card is returned, and the ATM shuts down with an "Out of Order" display. The use case ends with a failure condition.

E2) *Card not removed*. At step 9, if the card isn't removed within 15 seconds, the ATM issues a warning sound. If the card is not removed after a further 15 seconds, the card is retained, the transaction is cancelled, and the use case ends with a failure condition.

E3) *Cash not removed*. At step 10, if the cash is not removed from the dispenser within 15 seconds, the ATM issues a warning sound. If the cash is not removed after a further 15 seconds, the cash is retained, the transaction is cancelled, and the use case ends with a failure condition.

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 4, test each of the possible account types. This checks that the use case works correctly no matter which 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: if the connection to the bank doesn't work, or is intermittent, and if the machine has no (or very little) cash. This tests whether the software will still do reasonable things when users try to use it in unexpected ways.

(6) test for empty data fields. 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.

(7) test for invalid data entry. For example, test what happens if the user enters very large dollar amounts, or uses buttons that aren't numbers. 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 dollars and cents, or just dollars. 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. E.g. the user enters zero dollars, or the exact amount of cash in the machine;

(10) test for overfull data in data entry fields. For example, keep pressing number keys. This checks that the software doesn't crash.

(11) perform stress tests – for example, test on a network with heavy network load; Test what happens if a user repeatedly withdraws cash.

(12) test for internationalization. For example, test with foreign bank cards. If instructions are given in multiple languages, test they are accurate, etc.

(13) do interference testing. For example, what happens if the connection is dropped during step 8; what happens if the user cancels simulataneously with the transaction in step 9. These tests that the transaction is completed or rejected, and is not left in some strange state.

(14) 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 tests related to the use case]

[Software Quality – 20 marks] In manufacturing, quality management tends to focus on the statistical analysis of defects, which is then used to guide design improvements, to reduce variability and hence to reduce the number of defective components. For example, Motorola's Six Sigma strategy seeks to reduce defect rates down to fewer than 3 defective parts per million, by ensuring upper and lower specification limits are at least six standard deviations from the mean, for each design variable. It has been suggested that such approaches should be applied to software quality management. Critique this suggestion by answering the following questions:
(a) What is software quality and how would you go about measuring it?
(b) What are the key differences between software development and hardware manufacturing?
(c) How do those differences affect the application of ideas like Six Sigma to software?
(d) What techniques for quality improvement can be applied to software development?

4. **[Software Tools – 20 marks]** Name the three *development tools* that you feel have been most valuable on the course project this term. For each tool, explain what the tool does, why it was valuable, and how you used it. Make sure you describe any problems or weaknesses encountered for each tool. *Note: for the purposes of this question, a "tool" means a piece of software.*

Many possible answers; be generous in accepting different types of tool.
Answer must explain what the tool was, and address all of: (1) What the tools does (2) why it was valuable (3) how the team used it (4) any weaknesses.
Only give full marks if a convincing argument is made as to why these tools in particular were ***the most valuable*** (i.e. better than all the other tools!)
Answer should show good insights and reflection on the software development processes.

[scratch paper]

[scratch paper]

[scratch paper]