University of Washington
CSE 403 Software Engineering
Spring 2009

Final Exam

Friday, June 5, 2009

Name: ***Solutions*** _____

Initials: _____

UW net id: _____

UW id number: _____

This quiz is closed book, closed notes. You have **50 minutes** to complete it. It contains 25 questions and 11 pages (including this one), totalling 100 points. Before you start, please check your copy to make sure it is complete. Turn in all pages, together, when you are finished. **Write your initials on the top of *ALL* pages.**

Please write neatly; we cannot give credit for what we cannot read.

Good luck!

Page	Max	Score
2	14	
3	7	
4	8	
5	15	
6	6	
7	16	
8	14	
9	8	
11	12	
Total	100	

1 True/False

(2 points each) Circle the correct answer. T is true, F is false.

1. **T/F** A representation invariant should be part of the specification of every non-trivial class. ***False. The specification is an external view, but the representation invariant has to do with implementation details.***

2. **T/F** In general, designing a system such that its UML class/dependence diagram has more edges is good. ***False. It is better for things to be decoupled.***

3. **T/F** Assertions should never be enabled in production code. ***False***

4. **T/F** Bottom-up testing is not an effective way to reveal architectural design flaws. ***True. It* may *reveal such flaws, but isn't an effective way to do so. Top-down testing is an effective way to reveal architectural design flaws.***

5. **T/F** Top-down testing can be time-consuming, because you need to write stubs for many modules. ***True. Top-down testing tends to be more time-consuming than bottom-up, given that you need to write unit tests anyway, and they effectively perform bottom-up testing on their own.***

6. **T/F** One drawback of bottom-up system testing (as compared to top-down system testing) is that when a test fails, there are more places you have to look for the bug. ***True. Unless unit testing was perfect, the bug might appear in any class that was executed as part of the test. In top-down testing, all the system tests are written at the beginning and are simply re-run with fewer and fewer stubs.***

7. **T/F** Design patterns typically increase the amount of code that needs to be written to accomplish a specific purpose. ***This question was unnecessarily vague. If "specific purpose" is interpreted as "achieve specific functionality from the viewpoint of the user", then the answer is true: they more often increase the generality or reusability of the code, not its brevity.***

2 Multiple choice

8. (4 points) Which of the following reasons are valid ones for choosing a top-down process? (Circle all that apply.)

   (a) A top-down process is more time consuming because of the unit tests.

   (b) Developers can present a demo of the project to the management faster than using a bottom-up process.

   (c) In a top-down design, if an error is detected it's always because a lower-level module is not meeting its specifications (because the higher-level ones are already been tested).

   (d) A top-down process makes it possible to detect performance problems faster

   (e) A top-down process makes it easier to fix a global conceptual problem

   b. e

9. (3 points) The tests that run every night during the project development or before a programmer checks in any changes to the global repository are normally characterized as:

   (a) Regression tests

   (b) Integration tests

   (c) Validation tests

   (d) System tests

   (e) None of the above

   a

10. (4 points) Which of these statements about black-box tests are commonly true? (circle all that apply)

    (a) Black-box tests generally aim to provide as much coverage of the lines of code in a module as possible, because coverage is usually positively-correlated with test quality.

    (b) Black-box tests can be used to find aliasing errors that occur when two different formal parameters of a method both refer to the same object.

    (c) Black-box tests can be effective at detecting representation exposure.

    (d) In practice, it is often difficult to re-use the same black-box test when crucial parts of the implementation change, even when the specification remains the same.

    (e) Black-box tests are able to partition the input space into finer (smaller) partitions than glass-box tests.

    b, c. Black-box tests can find aliasing errors, especially in the inputs to the black-box region of code. They can even find aliasing errors elsewhere, but they are not guaranteed to find all such errors. Likewise, good black-box tests can detect representation exposure.

11. (4 points) Circle all of the below that are advantages of sketching a proposed user interface (as opposed to using a drawing program or building a prototype).

    (a) It is easier to make small changes to a large drawing

    (b) It is faster

    (c) It looks less polished to users, encouraging them to suggest changes

    (d) It makes it easier to ignore irrelevant details

    b, c, d

3 Short answer

Answer each question with **one sentence or phrase**.

12. (3 points) What is the most important reason for comparing a procedure to a specification?

    To determine whether the implementation is correct (with respect to the specification).

13. (3 points) What is the most important reason for comparing one specification to another?

    To determine whether an implementation of one specification can be replaced by (substituted by) another. Partial credit for: to determine whether one type is a subtype of another.

14. (3 points) In Java, `Integer` is a subtype of `Number`.
    `List<Integer>` is not a subtype of `List<Number>`. Why? (Don't give a Java-specific answer; explain why Java's choice is the right one.)

    `List<Integer>` does not support the operation `put(Float)`, but `List<Number>` does.

15. (3 points) `List<Number>` is not a subtype of `List<Integer>`. Why?

    `List<Number>` does not support the operation `Integer get()`, but `List<Integer>` does.

16. (3 points) For physical objects, maintenance is required to repair the effects of wear and tear. For non-buggy software, what is the most frequent cause that requires "maintenance"?

    Use of the software in a new environment for which it was not originally designed, but in which it is desired to be used.

    Partial credit for specific examples of this. The best such example is new user requirements. (That does not cover all cases, because users do not usually think of the software they depend on, such as the format of results from a given website, as part of their requirements.) New features are another good example. A bad example is operating system and programming language upgrades: they are rare, and they do not necessarily require software to be changed.

17. (6 points) Give the weakest precondition for the following code, with respect to the postcondition $x > y$. Assume that `p` is `boolean` and `x` and `y` are `int`.

```
p = x>y;
if (p) {
  x++;
} else {
  y = x + y;
}
```

Answer (show work below for partial credit): $x > y$ ***OR*** $y < 0$

Our plan is to work backward: first through the `if`, then through the `p =`.

$wp(\text{“}\texttt{if ...}\text{”}, x > y)$
$= p \Rightarrow wp(\text{“}\texttt{x=x+1}\text{”}, x > y)$ ***AND*** $\neg p \Rightarrow wp(\text{“}\texttt{y=x+y}\text{”}, x > y)$
$= p \Rightarrow x + 1 > y$ ***AND*** $\neg p \Rightarrow x > x + y$

$wp(\text{“}\texttt{p = x>y}\text{”}, p \Rightarrow x + 1 > y$ ***AND*** $\neg p \Rightarrow x > x + y)$
$= x > y \Rightarrow x + 1 > y$ ***AND*** $\neg(x > y) \Rightarrow (x > x + y)$
$= \mathit{true}$ ***AND*** $\neg(x > y) \Rightarrow y < 0$
$= \neg(x > y) \Rightarrow y < 0$
$= x > y$ ***OR*** $y < 0$

18. (8 points) Compare incremental (per-checkin) code reviews to comprehensive (whole-module, but now whole-system) code reviews.

    Give two benefits of incremental code reviews.

    - ***Quicker feedback if something is wrong with the code***
    - ***Prevents bugs from ever entering the codebase.***
    - ***Low cost per review.***
    - ***Incentive to write quality code and documentation from the beginning.***

    Common incorrect answers include:
    - ***Easier to see problems in a specific part of the code.***
    - ***Find different types of problems.***

    Give two benefits of comprehensive code reviews.

    - ***Reviewers get the big picture, can consider high-level design as well as low-level code issues, can find to miss buggy interactions between changed code and non-modified code.***
    - ***Gives an opportunity for brainstorming and discussion.***
    - ***Educates other developers.***
    - ***Does not require a reviewer who is already an expert on the code.***

    Give two differences in the mechanics of how they are typically performed.

    - ***Comprehensive are more often face-to-face; incremental are more often electronic (e.g., email).***
    - ***Incremental are more often before a checkin; comprehensive are more likely to be (but by no means universally) after checkin.***
    - ***Incremental are more likely to be done by one other person; comprehensive are more likely to be done by a group.***

19. (4 points) What are two aspects of a software system that are explicitly omitted from a UML class diagram?

    - ***Timing/ordering of calls; in fact, all dynamic information. More generally, information that appears in other UML diagrams such as sequence diagrams.***
    - ***Implementation details such as algorithms, data structures, and the actual code.***
    - ***Implementation language.***

    We were not looking for answers like the user manual, webpage, requirements, etc. that are part of your project but not part of the software system per se.

20. (4 points) Name the two key advantages of factory methods, when compared to constructors. (Use no more than 10 words each.)

    - ***Can return an existing object***
    - ***Can return a subtype of the declared type***

    More minor benefits include:
    - ***Being able to choose a name for the factory method (whereas all constructors are required to have the same name).***
    - ***It can return null***
    - ***It can be easily replaced by another factory, at run time or compile time***
    - ***A single factory can return different subtypes, choose at run time which subtype to return***

21. (6 points) Exhaustive testing (testing every value in the input domain reveals every defect, but is impractical. Partition testing splits the input domain into parts, and chooses just one test for each of the parts. Partition testing reveals every defect, under what condition? (One sentence or phrase)

    If any input in a part reveals a defect, then every input in the part does. ("Same behavior" instead of "reveals a defect" gets partial credit.)

    What might happen if some of the partitions are too large?

    Testing may miss errors, also known as suffering a false negative. You could test a good input in a partition that contains a failure, and miss the failure.

    What might happen if some of the partitions are too small?

    Testing may be inefficient: the suite may contain extra (unnecessary, redundant) tests. But, the approach is still sound and reveals every defect.

22. (8 points) State 3 distinct benefits of writing tests before writing the code.

    - ***Can be done by someone other than the coder; permits parallelizing human effort.***
    - ***Can reveal problems with the specification early.***
    - ***If the code exists, then it can bias a tester into the same thought processes, leading the tester to make the same mistakes as the coder.***
    - ***You are more likely to actually write the tests, and they are more likely to be complete.***
    - ***If no specification exists, tests can provide an approximation of one.***

    No credit for answers that are a benefit of writing tests in general, whether before or after writing the code.

    Is this approach more applicable to black-box tests, more applicable to clear-box tests (aka glass-box or white-box tests), or equally applicable to both?

    More applicable to black-box tests. Clear-box tests assume the existence of the source code.

4 Long answer

23. (8 points) Suppose that each procedure in my program has a specification. I wish to prove the whole program is correct; that is, each procedure satisfies its specification.

    If the program has no recursion, it is easy to reason *modularly* — that is, one procedure at a time. Here is how you could do that.

    (a) First, prove correctness of each procedure at the leaves of the procedure call tree (the ones that call no other procedure).

    (b) Next, prove correctness of the procedures that call only the leaf procedures. During this process, it is valid to rely on the specifications of the leaf procedures, and assume that each leaf procedure call does exactly what its specification says.

    (c) Now, continue up the procedure call tree.

    Now, suppose that the program contains *recursion*. When reasoning about a procedure call (including a self-call) it would be circular reasoning to assume that the procedure is correct if I have not already proved it — that is, to assume that the procedure is correct in order to prove it correct.

    What approach do you suggest in this circumstance? Explain why it works and any potential downside. Use no more than 5 sentences. (It is possible for a 1- or 2-sentence answer to get full credit.)

    Use induction over the dynamic call graph (in other words, over the length of an execution) rather than the static call graph.

    The base case is a sequence of 1 run-time call; does each procedure's base case behave properly (that is, obey the specification if it terminates)?

    The inductive case assumes that every sequence of $\leq n$ run-time calls behaves correctly, and shows (via examination of each procedure's recursive case) that every sequence of n run-time calls behaves correctly.

    You also need to prove that the program terminates.

    The most common problem was stating the mechanics of how to do the induction, but failing to say what the induction is over.

5 Reasoning

(This information is for questions 24–25 on page 5.)

Ben Bitdiddle joins the Megasensible Corporation's famous Flunk spreadsheet group. Bens first assignment is to refactor the Flunk source base to eliminate duplicate code that had accumulated over the years. Ben finds 4 implementations for the function that copies the first `n` elements from List `src` to List `dest`.

```
void partialcopy(int n, List<Integer> dest, List<Integer> src)
```

Fortunately for Ben, Megasensible only hires CSE 403 graduates, and all the implementations have specifications.

(Note: in this codebase, arrays are indexed starting with 1, not 0.)

```
Specification A
  requires: n > 0
  modifies: dest
  throws: ArrayOutOfBoundsException if src.length < n
  effects: for i=1..n  dest[i]post = src[i]pre
  returns: nothing

Specification B
  requires: n > 0
  modifies: src, dest
  throws: ArrayOutOfBoundsException if src.length < n
  effects: for i=1..n  dest[i]post = src[i]pre
  returns: nothing

Specification C
  requires: n > 0
  modifies: dest
  throws: nothing
  effects: for i=1..min(n, src.length):  dest[i]post = src[i]pre
           for i=src.length+1..n:  dest[i]post = 0
  returns: nothing

Specification D
  requires: n > 0 and src.length >= n
  modifies: dest
  throws: nothing
  effects: for i=1..n:  dest[i]post = src[i]pre
  returns: nothing
```

24. (9 points) In the following diagram, for all pairs of specifications, draw an arrow from X to Y (X→Y) if and only if X is stronger than Y. (There are 12 possible arrows that you might draw in the diagram, for each direction among the 6 pairs of specifications.)

A B

D C

A → B
A → D
C → D

Common incorrect edges:

B → D: B and D are incomparable because B modifies more but requires less.

C → A: A and C are incomparable because their postconditions are incomparable; when n > src.length, A guarantees to throw an exception, while C guarantees to set extra elements to 0. A client expecting one of the two specifications will not be satisfied by an implementation of the other.

25. (3 points) According to the specifications, which method(s) should Ben choose to replace all the others?

The correct answer is A and C. These are the only specifications of the four for which there is no stronger type. The answer "A or C" is incorrect because you cannot replace A with C or vice-versa, as they are incomparable specifications.

End of quiz.

Make sure that you have written your initials on the top of ALL pages.