University of Washington
CSE 331 Software Design & Implementation
Spring 2013

Final exam

Monday, June 10, 2013

Name: ***Solutions*** _____

Section: _____

CSE Net ID (username): _____

UW Net ID (username): _____

This exam is closed book, closed notes. You have **110 minutes** to complete it. It contains 21 questions and 13 pages (including this one), totaling 220 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** (in case a page gets separated during test-taking or grading).

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

Good luck!

Page	Max	Score
2	26	
3	8	
4	24	
5	18	
6	18	
7	24	
8	24	
9	24	
10	16	
11	16	
12	22	
Total	220	

1 True/False

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

1. **T / F** The `List` method `get` returns a reference to an element in the list, and therefore commits representation exposure.

2. **T / F** A specification for method `m` that says "`m` may modify variable `x`" is not useful to clients, because it does not give the client knowledge of the post-value in the same way as "`m` increments `x` by 1" (or, equivalently, "$\texttt{x}_{\text{post}¿} = \text{x}_{\text{pre}} + 1$"). ***The client now knows that they can't rely on the value of `x` after calling `m`.***

3. **T** / **F** Every getter method is an observer method, but not all observer methods are getters. ***Consider the observer `compareTo(T other)`, which is not a getter.***

4. **T** / **F** It is possible (and useful) for a specification to mention a specification field that does not correspond to any concrete field or concrete method result.

5. **T** / **F** It is permitted to specify a precondition (@requires clause), *and* for the implementation to check that condition and perform a specific action (such as throwing a specific exception).

6. **T / F** If every method in a Java class returns only immutable objects, then the class is immutable.

7. **T** / **F** In an ADT, more than one concrete state can represent the same abstract value.

8. **T / F** A test suite that detects every defect in an implementation has 100% statement coverage.

9. **T** / **F** Specification tests and black box tests are different names for the same concept.

10. **T / F** Implementation tests and white box tests are different names for the same concept.

    White-box vs. black-box refers to the methodology for creating the test; specification vs. implementation refers to the test's oracle or assertion. Likewise, a white-box test can test for specific specification-level behavior.

    For the remaining questions, suppose you have an implementation and a test suite that detects every defect in that implementation (for example, the test inputs were generated by taking a set of revealing subdomains and choosing one element from each).

11. **T / F** If you change the implementation without changing the specification or the tests, the test suite still detects every flaw in the implementation.

12. **T / F** If you change the specification without changing the implementation or the tests, the test suite still detects every flaw in the implementation.

13. **T** / **F** If you change the implementation in a correct way, without changing the specification or the tests, then the implementation tests may fail.

2 Multiple choice — single answer

For each part of each question, circle the *single* best choice.

14. (4 points) Given Spec A which asserts

```
@requires value occurs in a
@returns i such that a[i] = value
int find(int value, int[] a)
```

and Spec B which asserts

```
@returns i such that a[i] = value
   or -1 if value is not in a
int find(int value, int[] a)
```

Mark one of the following:

(a) Spec A is stronger than spec B

(b) Spec B is stronger than spec A

(c) The specifications are equally strong

(d) The specification are incomparable (they differ, and neither is stronger than the other)

15. (4 points) Choose the best specification for the `add` method for a collection class:

```
public void add(E element) {
   if (element == null) {
      throw new IllegalArgumentException();
   }
   ... // do things that add element to the collection
}
```

(a)
```
@requires nothing
@throws   IllegalArgumentException when element is null
@modifies this
@effects  adds element to this collection
```

(b)
```
@requires nothing
@modifies this
@effects  adds element to this collection
```

(c)
```
@requires element is not null
@throws   IllegalArgumentException if element is null
@modifies this
@effects  adds element to this collection
```

(d)
```
@requires element is not null
@modifies this
@effects  adds element to this collection
```

(d) allows the specification to be strengthened in a subclass to allow `null` to be stored in the collection, but (a) does not. (b) and (c) are just wrong.

16. (8 points) For each part of each question, circle the single best choice from among those in curly braces.

    - True subtyping for generics is the same as **{ contravariant / covariant / invariant }** subtyping.
    - Java generics use **{ contravariant / covariant / invariant }** subtyping.
    - True subtyping for arrays is the same as **{ contravariant / covariant / invariant }** subtyping.
    - Java arrays use **{ contravariant / covariant / invariant }** subtyping.

17. (16 points) Suppose you have defined multiple objects (such as representations of code expressions) and multiple operations on each object (such as type-checking and printing). There is an implementation of each operation for each each object (the blank cells of this table):

		Objects	
		CondExpr	EqualOp
Operations	type-check		
	pretty-print		

    Selecting an implementation to execute is equivalent to selecting a row and selecting a column. When is are the selections made, and based on what information?

    The possible answers for when the selection is made are:

    - run time
    - compile time

    The possible answers for how the selection is made are:

    - by dynamic dispatch on the type of the receiver object
    - by overloading resolution on the types of the arguments
    - by `instanceof` tests
    - by the name of the method being invoked

    Circle one choice for each place you have an option below.

    (a) For the interpreter pattern, the column is determined at ***run*** time by ***by dynamic dispatch on the type of the receiver object***.

    (b) For the interpreter pattern, the row is determined at ***compile*** time by ***by the name of the method being invoked***.

    (c) For the visitor pattern, the column is determined at ***compile*** time by ***by overloading resolution on the types of the arguments***.

    (d) For the visitor pattern, the row is determined at ***run*** time by ***by dynamic dispatch on the type of the receiver object (which is a visitor, a row of the table)***.

3 Multiple choice — multiple answers

Mark all of the following that must be true, by circling the appropriate letters.

18. (9 points) Sometimes, attempting a formal proof of a particular software component is not a productive use of time. Mark each reason that justifies *not* attempting a formal proof.

    (a) The code is complex, making the formal proof difficult.
    The proof might lead you to simplify your code.

    (b) The code is simple and therefore is unlikely to contain unnoticed errors.
    You might have overlooked something.

    (c) Representation exposure is not possible, eliminating a major source of errors.

    (d) Representation exposure is a possibility, complicating the proof.

    (e) The specification is vague, making the goal of the formal proof unknowable.
    It's OK for a specification to be lenient, but not for it to be vague. Attemting a formal proof will lead you to improve the specification.

    (f) Visual appearance is the measure of quality, as in the look of user interface components.

    (g) It is early in the design process, when the specification is subject to change.

    (h) The code is returning results for an Internet search algorithm, where usefulness to people is the measure of quality.
    The specification will still be precise.

    (i) The code is an isolated or relatively unimportant module, whose failure does not jeopardize the success of the overall system.

19. (5 points) What kinds of files should be committed to your source control repository?

    (a) code files

    (b) documentation files

    (c) output files

    (d) automatically-generated files that are required for your system to be used

    (e) your "scientific notebook" of experiments about debugging a problem

20. (4 points) Which of the following are benefits of good software architecture? Select all that apply.

    (a) a well architected system will uses parallel processing

    (b) a well architected system will help ensure components which are are constructed separately will interact correctly.

    (c) a well architected system allows clear monitoring of development progress

    (d) a well architected system will cluster code in as few classes as possible (often only one)

21. (6 points) Assume that the following code passes the Java typechecker:

```
A a = new A();
B b = new C(a);
D d = b;
```

Mark all of the following that must be true:

(a) `A` is a supertype of `B`
(b) `A` is a subtype of `B`
(c) `A` is a supertype of `C`
(d) `A` is a subtype of `C`
(e) `A` is a supertype of `D`
(f) `A` is a subtype of `D`
[(g)] `B` is a supertype of `C`
(h) `B` is a subtype of `C`
(i) `B` is a supertype of `D`
[(j)] `B` is a subtype of `D`
(k) `C` is a supertype of `D`
[(l)] `C` is a subtype of `D`

22. (6 points) Mark every word that completes the following sentence to make a true statement.

In Java, every _____ has a compile-time type.

(a) declaration
[(b)] expression
(c) statement
(d) value
[(e)] variable

A value is not a compile-time construct. A declaraton (including a class declaration) does not itself have a compile-time type.

23. (6 points) Suppose that you have a private inner class that is only intended to be instantiated once. What are the benefits of writing the class as an anonymous inner class instead?

[(a)] shorter code (less boilerplate code)
(b) prevents the client from accessing the class
(c) prevents the client from instantiating the class

4 Short answer

24. (8 points) Suppose that a specification field's value can be determined from the values of other specification fields. Such a relationship will be stated in the specification. You may declare the specification field as a regular specification field or as a derived specification field.

    In one sentence, state an advantage of declaring the specification field as a derived specification field rather than as a regular specification field.

    The method specifications and the abstraction function are shorter: they don't have to mention that specification field.

25. (8 points) Given the following declarations, write additional code that contains no casts and satisfies the Java type system (that is, there is no compile-time error or warning), but does not satisfy true subtyping and therefore crashes at run time due to an `ArrayStoreException`.

```
Date[] da = new Date[] { new Date() };
// The above line is the same as:  Date[] da = new Date[1]; da[0] = new Date();
Object[] oa = new Object[] { "hello" };

oa = da;
oa[0] = "hello";
```

26. (8 points) Write a `min` method that returns the smaller of its two arguments and passes the following two tests. You do not have to write documentation — just the code. For ease of grading, please name the two formal parameters `a` and `b`.

```
assert min(3, 4) == 3;
assert min(2.718, 3.14) == 2.718;
assert min("hello", "goodbye") == "goodbye";
```

    Either of the following is acceptable:

```
public static <E extends Comparable<E>> E min (E a, E b) {
  return (a.compareTo(b) < 0) ? a : b;
}

public static <T extends Comparable<T>> T min(T a, T b) {
  if (a.compareTo(b) < 0) {
    return a;
  }
  return b;
}
```

27. (8 points) Unlike the CSE 331 Campus Maps application, the UW map application at `http://www.washington.edu/maps/` permits users to select a building using either free text search *or* a dropdown menu.

    (a) Explain a circumstance and/or class of users for whom the free text search is likely to be more useful, and explain why. (1–2 sentences)

    A user may not have memorized the names of the buildings, but wish to search for a part of the same. For instance, searching for "Medical" is more natural than remembering a name like "UMSA".

    (b) State a circumstance and/or class of users for whom the dropdown menu is likely to be more useful, and explain why. (1–2 sentences)

    For a user who is familiar with the campus and the building names, the dropdown prevents typos that can occur when typing, such as "Pal Allen Center".

28. (4 points) Name the key design pattern used in event-driven programming, in one word or short phrase: The ***Observer*** Pattern.

29. (12 points) State three benefits of using a version control system. Use one sentence or phrase each. State benefits that are as different from one another as possible.

    (a) ***Allows multiple people to concurrently modify source code, and elegantly handle combining their changes together.***

    (b) ***Keeps a history of how software has changed over time, which facilitates finding bugs and older code more easily.***

    (c) ***Manages multiple versions or branches of source code, which allows for stable releases to be developed, and makes it relatively easy to move bug fixes and features between the trunk version and stable verisons.***

30. (4 points) In no more than one sentence, state an advantage of bottom-up implementation.

Bottom-up implementation avoids the need to create stubs and temporary implementations for testing, since all of a module's dependencies will be complete before that module is developed. It finds some (but not all) efficiency problems faster: those that have to do with algorithms implemented by low-level modules.

Here are some common incorrect answers. "You can (unit-)test the modules before integrating": that is possible using either approach. "You know that the callees are correct when writing the calling code and its tests": during those tasks (as opposed to when performing debugging tasks), you should rely only on the specification, not anything about correct implementations. "It is easier to find and isolate problems": stubs enable one to find and isolate problems, as well.

31. (4 points) In no more than one sentence, state an advantage of top-down implementation.

Top-down implementation allows higher-level components (such as a UI) to be demoed and refined, and allows experimentation with integrating all the top-level modules early in the development process — without the cost of fully developing all of the system. It finds some (but not all) efficiency problems faster: those that have to do with multiple calls to low-level modules, or with transformations from one data structure to another. It enables easier tracking of development progress (from a functionality point of view).

32. (12 points) Suppose that you have written a class that contains a private method. Your test class cannot call that method. In one sentence each, give three distinct approaches for writing tests that detect errors in the private method's implementation. Give reasons that are as different from one another as possible.

(a) ***Indirect testing: Devise tests of the public methods that exercise all the functionality of the private methods.***

(b) ***Embed tests: Write the tests inside the class, where they have access to the private method. The test method itself can be exposed as*** `testInternal`***, whose specification is like that of*** `checkRep`***.***

(c) ***Reflection: Use reflection to circumvent Java's access restrictions on private methods.***

(d) Publicize: Make the method non-private (say, public or package-protected), and write normal tests. Or, create a new, public method that calls the private one. A variant is make the method non-private only during testing, then change it back to private.

(e) Cut-and-paste: make a copy of the method in the test class, and test it there.

(4 points) State which of the three is best, and justify your choice with a single sentence.

Indirect testing is best. It creates a specification test that will continue to be useful even if your implementation changes.
Embedded tests is second-best. A downside is that not all of your test code are in the same place — some are in your test class and some are in the implementation.
Reflection makes the tests very difficult to read.
Publicizing is poor, because it breaks modularity and changes the specification. That's true even for creating a new public method, since clients will see that new method in the Javadoc: any public method is de facto a part of the specification.
It's a bad idea to change the method to non-private only temporarily, because that would force you to delete or comment out the tests (which wouldn't compile once the method was private) and lose all benefits of regression testing. Besides, testing is never truly "done".
Cut-and-paste is a bad idea (you can tell that just from the name) because it requires the developer to remember to keep the two versions of the code in sync, which would be easy to forget and unlikely to be noticed.

An incorrect answer is to add assertions within the implementation; that does not create a test as required by the problem, and there is no guarantee that the test suite would ever exercise those assertions.

5 Free Willy (from ignorance)

20. (8 points) Recall Willy Wazoo's implementation of the absolute value function:

```
int abs(int x) {
  if (x < -2) return -x;
  else return x;
}
```

Suppose that you partition a procedure's domain so that every subdomain is revealing. (Recall that in a partition, each input is in exactly one subdomain.) Then creating a set of test inputs by choosing one element from each subdomain will guarantee the discovery of the error.

Give a different partition of the domain of `abs` — not into revealing subdomains — such that creating a set of test inputs by choosing one element from each subdomain still guarantees the discovery of the error.

{ -infinity . . . -2 }, {-1}, {0 . . . infinity}

The first subdomain is not revealing, but so long as there exists one revealing subdomain* that fails the test*, the error is found.

Here is another possible answer:

{ $x \neq$ -1 }, { $x =$ -1 }

21. (8 points) Willy Wazoo specified the following method, but he forgot to write down the generic types.

```
/**
 * Make "pruned" a copy of "orig" without duplicates.  Does not modify "orig".
 * @modifies pruned
 */
static void removeDuplicates(Collection orig, Collection pruned);
```

Please rewrite the entire method signature (you don't have to rewrite the Javadoc) with the most appropriate signature.

Any of the following is acceptable:

```
<T> static void removeDuplicates(Collection<T> orig, Collection<? super T> pruned)
<T> static void removeDuplicates(Collection<? extends T> orig, Collection<T> pruned)
<T> static void removeDuplicates(Collection<? extends T> orig, Collection<? super T> pruned)
```

22. (16 points) Willy Wazoo has written the following code. He is sure that the final line, `count.toString()`, never throws a `NullPointerException`, but he isn't sure *why* he's sure.

    Help Willy by adding enough Nullness Checker annotations to the class `Keiko` to guarantee that `count.toString()` does not throw a `NullPointerException`, (under the assumption that every method body satisfies its specification as expressed by the annotations). Write all necessary properties; do not rely on default annotations. Do not write any extra, unnecessary annotations. Do not use postcondition annotations such as `@AssertNonNullAfter`.

```
class Keiko {

  /*@NonNull*/ Map<String, /*@NonNull*/ Integer> counts;

  String name;

  Keiko() {
    ...
  }

  /*@KeyFor("counts")*/ String getAString() {
    ...
  }

  void doSomeThings() {
    ...
  }

  /*@Pure*/ Object doOtherThings() {
    ...
  }

  void doMoreThings(Object arg) {
    ...
  }

  void myMethod() {
    doSomeThings();
    name = getAString();
    Object o = doOtherThings();
    Integer count = counts.get(name);
    doMoreThings(o);
    count.toString();
  }

}
```

It is possible to omit the `@NonNull` annotation from the type of `counts`. To compensate, there would need to be a postcondition annotation somewhere that establishes its non-nullness, as well as more purity annotations and/or an `@LazyNonNull` annotation on the type of `counts`.

The return value of `getAString` does* not *have to be annotated as `@NonNull`, because `null` can be used as a key to a map.

`doMoreThings` does not need to be annotated as `@Pure`; a method call cannot affect the binding of a local variable such as `count`.

Willy Wazoo has written the `Point2D`, `Point3D`, and `Point4D` classes that appear on page 13.

Answer the following questions without defining new classes or modifying the existing code.

23. (6 points) Give an example of how the **symmetry** of equals could be broken.
The end of your answer should show a series of calls to `equals` and should state the values to which they evaluate.

```
Point2D a = new Point2D(1, 1);
Point3D b = new Point3D(1, 1, 1); // This could also be a Point4D
a.equals(b) // returns TRUE
b.equals(a) // returns FALSE
```

24. (6 points) Give an example of how the **transitivity** of equals could be broken.
The end of your answer should show a series of calls to `equals` and should state the values to which they evaluate.

```
Point4D a = new Point4D(1, 1, 1, 1);
Point3D b = new Point3D(1, 1, 1);
Point4D c = new Point4D(1, 1, 1, 3);
a.equals(b) // TRUE
b.equals(c) // TRUE
c.equals(a) // FALSE
```

25. (10 points) Mark all of the following `hashCode` implementations that would be valid (they satisfy the spec of `hashCode`) in the `Point3D` class.

(a)
```
public int hashCode() {
  return 42;
}
```

(b)
```
public int hashCode() {
  return (new Random()).nextInt();
}
```

(c)
```
public int hashCode() {
  return x + y;
}
```

(d)
```
public int hashCode() {
  return x * y * z;
}
```

(e)
```
public int hashCode() {
  if (this instanceof Point4D)
    return new Random().nextInt() * ((Point4D) this).w;
  else
    return z;
}
```

You may tear off this page if you wish. You do **not** need to turn it in.

```
public class Point2D {
  protected int x, y;

  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object o) {
    if (! (o instanceof Point2D))
      return false;
    return ((Point2D) o).x == x &&
           ((Point2D) o).y == y;
  }
}

public class Point3D extends Point2D {
  protected int z;

  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }

  public boolean equals(Object o) {
    if (!(o instanceof Point3D))
      return false;
    return ((Point3D) o).x == x &&
           ((Point3D) o).y == y &&
           ((Point3D) o).z == z;
  }
}

public class Point4D extends Point3D {
  protected int w;

  public Point4D(int x, int y, int z, int w) {
    super(x, y, z);
    this.w = w;
  }

  public boolean equals(Object o) {
    if (o instanceof Point4D) {
      return ((Point4D) o).x == x &&
             ((Point4D) o).y == y &&
             ((Point4D) o).z == z &&
             ((Point4D) o).w == w;
    } else if (o instanceof Point3D) {
      return ((Point3D) o).equals(this);
    } else {
      // not a Point3D
      return false;
    }
  }
}
```