University of Washington
CSE 331 Software Design & Implementation
Spring 2010

Final exam

Monday, June 7, 2010

Name: ***Solutions*** _____

CSE Net ID (username): _____

UW Net ID (username): _____

This exam is closed book, closed notes. You have **110 minutes** to complete it. It contains 25 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.**

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

Good luck!

Page	Max	Score
2	14	
4	14	
5	14	
6	14	
7	26	
8	28	
9	16	
10	28	
11	40	
12	26	
Total	220	

1 True/False

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

1. **T /** **F** In order to instantiate a class, it must have at least one public constructor.

In the following questions, consider two method specifications S1 and S2, where S1 is stronger than S2.

2. **T** **/ F** S1 may have a weaker (easier-to-satisfy) `requires` clause than S2.
3. **T /** **F** S1 may have a stronger (harder-to-satisfy) `requires` clause than S2.
4. **T** **/ F** S1 may have fewer items in its `modifies` clause than S2.
5. **T /** **F** S1 may have more items in its `modifies` clause than S2.
6. **T /** **F** S1 may have a weaker (easier-to-satisfy) `effects` clause than S2.
7. **T** **/ F** S1 may have a stronger (harder-to-satisfy) `effects` clause than S2.

2 Equality

Questions 8–10 use the following code for a 1–dimensional point. This code is duplicated at the end of the exam on page 13. You may rip out the extra copy, for reference, if you like.

```
public class OneDPoint {
  double x;
  public boolean equals(Object o) {
    if (o instanceof OneDPoint) {
      OneDPoint other = (OneDPoint) o;
      return x == other.x;
    } else {
      return false;
    }
  }
}
```

8. (14 points) Consider the following subclass.

```
public class TwoDPoint extends OneDPoint {
  double y;
  public boolean equals(Object o) {
    if (o instanceof TwoDPoint) {
      TwoDPoint other = (TwoDPoint) o;
      return x == other.x && y == other.y;
    } else {
      return false;
    }
  }
}
```

Indicate whether each of the following properties holds for the `equals` method, and if the answer is false (the property does not hold), give a counterexample.

(a) **[T] / F** Reflexivity
Counterexample: ***N/A***

(b) **T / [F]** Symmetry
Counterexample:
OneDPoint a; a.x = 5;
TwoDPoint b; b.x = 5; b.y = 10;
a.equals(b) $\Rightarrow$ ***true***
b.equals(a) $\Rightarrow$ ***false***

(c) **[T] / F** Transitivity
Counterexample: ***N/A***

9. (14 points) Consider the following different subclass.

```
public class TwoDPoint extends OneDPoint {
  protected double y;
  public boolean equals(Object o) {
    if (o instanceof OneDPoint) {
      return o.equals(this);
    } else if (o instanceof TwoDPoint) {
      TwoDPoint other = (TwoDPoint) o;
      return x == other.x && y == other.y;
    } else {
      return false;
    }
  }
}
```

Indicate whether each of the following properties holds for the `equals` method, and if the answer is false (the property does not hold), give a counterexample.

(a) **T / [F]** Reflexivity
Counterexample: ***For any `TwoDPoint p`, execution of `p.equals(p)` suffers an infinite loop and stack overflow.***

(b) **[T] / F** Symmetry
Stack overflow does not violate symmetry, because the behavior of `a.equals(b)` is the same as `b.equals(a)`.
Counterexample: ***N/A***

(c) **[T] / F** Transitivity
Counterexample: ***N/A***

10. (14 points) Consider the following different subclass.

```
public class TwoDPoint extends OneDPoint {
  protected double y;
  public boolean equals(Object o) {
    if (o instanceof TwoDPoint) {
      TwoDPoint other = (TwoDPoint) o;
      return x == other.x && y == other.y;
    } else if (o instanceof OneDPoint) {
      return o.equals(this);
    } else {
      return false;
    }
  }
}
```

Indicate whether each of the following properties holds for the `equals` method, and if the answer is false (the property does not hold), give a counterexample.

(a) **[T] / F** Reflexivity
Counterexample: ***N/A***

(b) **[T] / F** Symmetry
Counterexample: ***N/A***

(c) **T / [F]** Transitivity
Counterexample:
TwoDPoint a; a.x = 5; a.y = 10;
OneDPoint b; b.x = 5;
TwoDPoint c; c.x = 5; c.y = 20;
a.equals(b) $\Rightarrow$ ***true***
b.equals(c) $\Rightarrow$ ***true***
a.equals(c) $\Rightarrow$ ***false***

3 Short answer

11. (3 points) How many arguments does method `Object.hashCode` take? ***1 (the receiver)***

12. (8 points) For each pattern, mark whether the the functionality and the interface is the same or different, compared to the delegate. Write the word "same" or "different" in each box.

Pattern	Functionality	Interface
Adapter	***same***	***different***
Decorator	***different***	***same***
Proxy	***same***	***same***

13. (6 points) For each of the following situations, indicate whether throwing a checked or unchecked exception is more appropriate. Write "checked" or "unchecked" in the space provided.

    (a) The client supplied an argument that violates a precondition. ***unchecked***

    (b) The client specified a file name, but the file cannot be found. ***checked***

    (c) The implementation contains a bug that its own self-checks detected. ***unchecked***

14. (4 points) What should you do first when a bug is reported to you (or you find it yourself)? Circle the best answer.

    (a) explain the bug to a peer

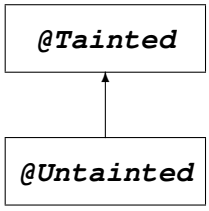
    (b) fix the bug by changing the code

    (c) reproduce the bug

    (d) understand the bug

    Reproduce the bug.

15. (5 points) A tainted value is one that comes from arbitrary, unvalidated data, such as user input. An untainted value is one that can be trusted, such as a value that has been validated or was hard-coded in the program. It is safe to use untainted values where tainted values are needed, but not vice versa. Draw the type hierarchy for the qualifiers `@Tainted` and `@Untainted`. The hierarchy may introduce a third annotation, if needed.



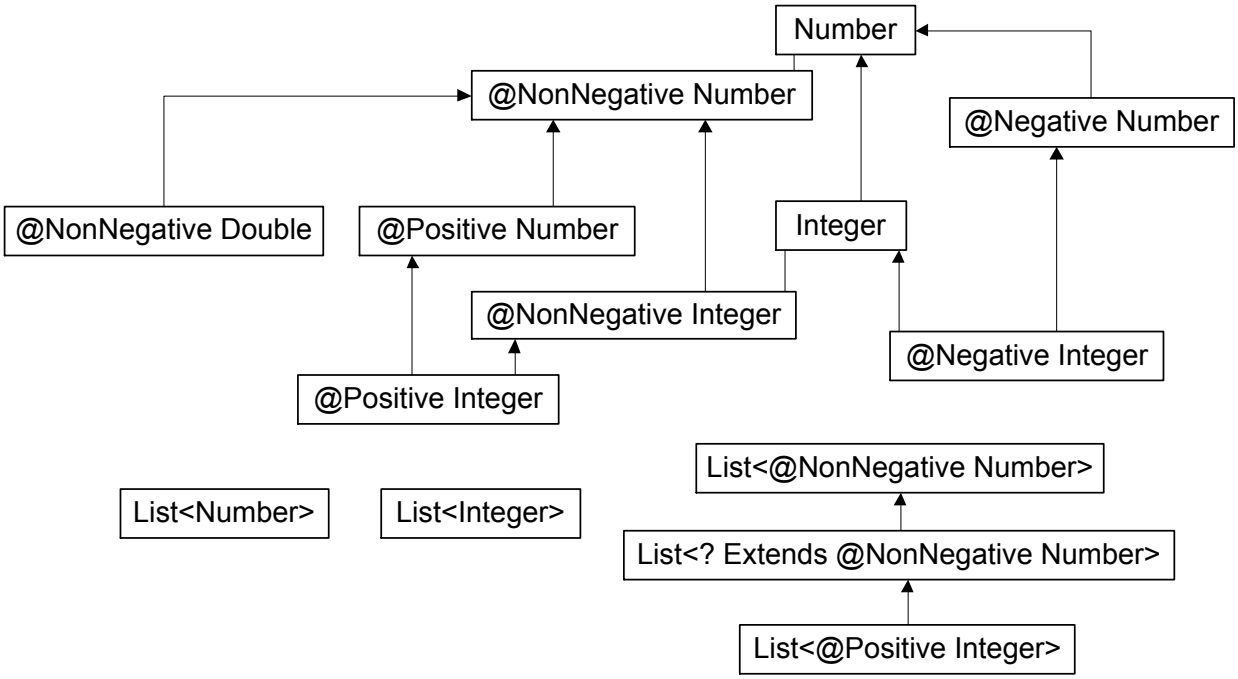
16. (28 points) Suppose that you have defined the following type qualifiers:
    - `@Positive` for values > 0
    - `@NonNegative` for values ≥ 0
    - `@Negative` for values < 0

    These only apply to `java.lang.Number` and its subtypes (`Integer`, `Double`, etc.).

    Express the subtyping relationship among all the following types.

    - `Number`
    - `Integer`
    - `@Positive Number`
    - `@NonNegative Number`
    - `@Negative Number`
    - `@Positive Integer`
    - `@NonNegative Integer`
    - `@Negative Integer`
    - `@NonNegative Double`
    - `List<Number>`
    - `List<@NonNegative Number>`
    - `List<? extends @NonNegative Number>`
    - `List<Integer>`
    - `List<@Positive Integer>`

    Please draw a subtyping diagram (please draw neatly!), with the following properties:
    - Every subtype appears lower (on the page) than all its supertypes, and there is an arrow from the subtype up to the supertype.
    - Draw as few arrows as possible: if A is a subtype of B, B is a subtype of C, and A is a subtype of C, don't draw the A→C arrow that is implied by the other two. (If B isn't in the diagram, then do draw the A→C arrow.)
    - For brevity, you may use the following abbreviations: N for Number, I for Integer, D for Double, @P for @Positive, @NN for @NonNegative, @N for @Negative.
    - Number and Integer are done for you (without the abbreviations).
    - Tip: double-check that there are 14 boxes in your diagram when you are done.



17. (16 points) Suppose that you as a programmer know that the Queue `q` is guaranteed to be non-null and non-empty at a particular code location. However, this fact is not immediately evident to you from looking at the nearby code. Assume these declarations:

```
Queue<Foo> q = ...;
Foo headElt = null;
```

and rank the following ways to get the head element into `headElt`, from best style to worst style. Recall that the `remove` method pops a value from the head of a Queue, or throws `NoSuchElementException` (a subclass of `RuntimeException`) if the Queue is empty.

(a)
```
if (q != null && ! q.isEmpty())) {
    headElt = q.remove();
}
```

(b)
```
if (q == null || q.isEmpty())
    throw new RuntimeException("q should be non-null and non-empty")
headElt = q.remove();
```

(c)
```
assert q != null && ! q.isEmpty() : "q should be non-null and non-empty";
headElt = q.remove();
```

(d)
```
headElt = q.remove();
```

(e)
```
try {
    headElt = q.remove();
} catch (NullPointerException e) {
    e.printStackTrace();
} catch (NoSuchElementException e) {
    e.printStackTrace();
}
```

Ranking (from best to worst): (best) ***d***, ***c***, ***b***, ***e***, ***a*** (worst).

Explanation of ranking (use 1 phrase per option):

d does not clutter the code with tests that will never fail. If the programmer was wrong, it gives the most informative stack backtrace, distinguishing between a nullness and an emptiness error. The call `q.remove` is already asserting the programmer's belief that `q` is non-null and non-empty, and we do not write an assertion before every field dereference or method call.

b is about the same as c — there is no particular benefit to throwing a `RuntimeException` rather than an `AssertionException` — but c is wordier and cannot be turned on and off. Both give less useful debugging information than d does.

e and a are worst because the code continues execution — a does so silently — rather than failing fast. Note that e does not re-throw the exception; some people seemed to think that it did.

Actually, none of these options is very satisfactory. The best solution would be to write a comment explaining how you know that `q` is non-null and non-empty.

18. (12 points) Suppose that a programmer started with a program whose module dependence diagram (MDD) is M1, and performed a refactoring to end up with a program with improved design whose MDD is M2.

    (a) Is it possible that M2 has fewer edges than M1? Explain why or why not. (1 sentence)

    Yes. The refactoring may have eliminated undesired coupling.

    (b) Is it possible that M2 has more edges than M1? Explain why or why not. (1 sentence)

    Yes. The refactoring may have introduced a new interface and replaced a dependence on a concrete class by a dependence on the interface, as shown in lecture.

19. (8 points) What is the difference between efficiency and learnability in the context of interface usability? (1–2 sentences)

    Learnability is how quickly* new *users get used to the interface.
    Efficiency is how quickly* experienced *users can use the interface to do work.

20. (8 points) Lecture gave an example of a visitor that could be applied to a tree. If the visitor code was applied to a cyclic data structure, such as a graph, the code would suffer an infinite loop and a stack overflow. How could the code be changed so that it would not suffer this problem? ($\leq$2 sentences)

    Maintain a list of all nodes visited so far. If one is visited again, return immediately from that invocation of the visitor.

21. (6 points) Suppose that class `C` cannot be instantiated. How could a client use the class? ($\leq$1 sentence)

    There are two reasons that a class cannot be instantiated: its constructors always throw an exception, or the class is abstract. In either case, a client can call static methods that `C` defines. If the class is abstract, its non-static methods can be used by a subclass. (For full credit for the latter answer, you would have had to mention that the class was abstract.)

    "Via a factory method" is incorrect, because the factory would need to instantiate the class, and the question says the class cannot be instantiated.

22. (12 points) You wish to deliver a program that does not suffer from representation exposure that can corrupt your data structures (violate the rep invariant). In 1 phrase each, give 4 approaches. Choose 4 approaches that are as different from one another as possible.

    (a) ***design your data structures to be immutable***

    (b) ***call `checkRep` frequently, to detect problems during testing***

    (c) ***use an immutability type-checker***

    (d) ***perform an inductive proof that the rep never escapes***

    (e) perform defensive copies where appropriate

23. (12 points) Give 4 strategies for creating a method stub (1 phrase each). Give strategies that are as different from one another as possible.

    Recall that a stub is a small, partially-functional implementation that permits testing client code before the method being stubbed has been fully implemented.

    (a) ***Print a message describing the call, if the calling program doesn't need a result***

    (b) ***Generate a canned set of responses***

    (c) ***Provide a partial implementation with a lookup table***

    (d) ***Return an arbitrary or random value***

    (e) Use an inefficient implementation

    "Throw an exception" is not a good answer, because the client code must continue running after calling the client code.

24. (10 points) Give an example of code that cannot go wrong at run time, but which the Java type system rejects (no more than 3 lines of code, 1 is enough). Explain why the Java type system rejects it and why it is actually safe (1 sentence each).

    `Integer i = new Integer(22).clone();`
    The compile-time type of `new Integer(22).clone()` is `Object`, and an assignment of an `Object` value to an `Integer` variable is illegal.
    The the run-time type of `new Integer(22).clone()` is `Integer`, and so the assignment is always legal.

25. (26 points) Prove the correctness of the following loop. You can (and should) do so in 4 bullet points. Do not show any work between the two statements of the loop body. You need to state every fact that is relevant, but you do not need to justify facts based on the laws of arithmetic. All variables are integers, and you can ignore overflow.

precondition *PRE*: $\texttt{r} = 1 \wedge \texttt{i} = 0 \wedge \texttt{n} > 0$

```
while (i != n) {
    r = r * x;
    i = i + 1;
}
```

postcondition *POST*: $\texttt{r} = \texttt{x}^{\texttt{n}} \wedge \texttt{i} = \texttt{n}$

You need to (a) provide a loop invariant LI, (b) show it is true initally, (c) show it is maintained by the loop, and (d) show it implies the postcondition.

Let B be the loop body and $pred \equiv \texttt{i} \neq \texttt{n}$ ***be the predicate that controls the loop.***

(a) ***Loop invariant LI:*** $\texttt{r} = \texttt{x}^{\texttt{i}} \wedge \texttt{i} \leq \texttt{n}$

(b) $PRE \Rightarrow LI$ ***obvious by substitution***

(c) $LI \wedge pred \;\; \{ \;\; B \;\; \} \;\; LI_{post}$

$(\texttt{r} = \texttt{x}^{\texttt{i}} \wedge \texttt{i} \leq \texttt{n}) \wedge \texttt{i} \neq \texttt{n} \;\; \{ \;\; \texttt{r} = \texttt{r} * \texttt{x};\ \texttt{i} = \texttt{i} + 1 \;\; \} \;\; \texttt{r}_{post} = \texttt{x}^{\texttt{i}_{post}} \wedge \texttt{i}_{post} \leq \texttt{n}$

(d) $LI \wedge \neg pred \Rightarrow POST$ ***obvious by substitution***

A commom mistake was assuming the inductive step is after one loop iteration rather than after zero. Another common mistake was noting that execution is guaranteed to execute the loop at least once; this is true here but irrelevant in the general case. Another common mistake was an assumption that the decrementing function indicates the number of loop iterations; it is a **bound** ***rather than an exact count.***

4 Code for `OneDPoint`

Questions 8–10 use the following code for a 1–dimensional point. You may rip out this page if you find doing so useful. You do not need to turn in this page.

```
public class OneDPoint {
  double x;
  public boolean equals(Object o) {
    if (o instanceof OneDPoint) {
      OneDPoint other = (OneDPoint) o;
      return x == other.x;
    } else {
      return false;
    }
  }
}
```