University of Washington
CSE 331 Software Design & Implementation
Spring 2013

Midterm exam

Wednesday, April 24, 2013

Name: _____

Section: _____

CSE Net ID (username): _____

UW Net ID (username): _____

This exam is closed book, closed notes. You have **50 minutes** to complete it. It contains 21 questions and 9 pages (including this one), totaling 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** (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	16	
3	12	
4	16	
5	8	
6	14	
7	14	
8	20	
Total	100	

1 True/False

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

1. **T / F** There may exist two logically distinct weakest preconditions A and B for a given bit of Java code. (Logically distinct means that A and B are not just different ways of writing exactly the same logical formula.)

2. **T / F** There may exist two logically distinct loop invariants LI_1 and LI_2 that enable proving a loop correct, where $LI_1 \Rightarrow LI_2$.

3. **T / F** There may exist two logically distinct loop invariants LI_1 and LI_2 that enable proving a loop correct, where LI_1 and LI_2 are incomparable, that is $LI_1 \not\Rightarrow LI_2$ and also $LI_2 \not\Rightarrow LI_1$.

4. **T / F** There may exist two distinct decrementing functions that enable proving a given loop terminates.

5. **T / F** When the loop terminates, the decrementing function is 0.

6. **T / F** When the decrementing function is 0, the loop terminates.

7. **T / F** An implementer has the responsibility to implement all behaviors permitted by a specification.

8. **T / F** You can tell by looking at a test case whether it was written using a black-box or a clear-box testing methodology.

Consider the following code for an `add` method of a collection class:

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

For each of the following specifications, indicate whether it is a valid specification for this code (T) or not (F).

9. **T / F**

```
@requires nothing
@modifies this
@effects  adds element to this collection
```

10. **T / F**

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

11. **T / F**

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

12. **T / F**

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

13. (4 points) Which of the above specifications do you think is best? Write its number, and write a sentence of justification.

2 Multiple choice

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

14. Suppose you have two correct implementations P1 and P2 of the same spec.

    (a) A correct clear-box test written with P1 in mind **{ will / might / won't }** pass when run on P2.

    (b) A correct black-box test written with P1 in mind **{ will / might / won't }** pass when run on P2.

15. Suppose specification S1 is stronger than specification S2, P1 is a correct implementation of S1, and P2 is a correct implementation of S2.

    (a) Suppose P1 passes test T1 and P2 passes test T2. P2 **{ will / might / won't }** pass T1.

    (b) Suppose T1 is a valid test of S1, and T2 is a valid test of S2. P1 **{ will / might / won't }** pass T2.

    (c) Suppose T1 is a valid test of S1, and T2 is a valid test of S2. P2 **{ will / might / won't }** pass T1.

16. (6 points) Assume that the following propositions are true:

$$\{b\}\ \texttt{mycode}\ \{y\}$$
$$a \Rightarrow b$$
$$b \Rightarrow c$$
$$x \Rightarrow y$$
$$y \Rightarrow z$$

    Circle all of the following that must be true:

    (a) $\{a\}$ `mycode` $\{y\}$

    (b) $\{c\}$ `mycode` $\{y\}$

    (c) $\{b\}$ `mycode` $\{x\}$

    (d) $\{b\}$ `mycode` $\{z\}$

17. (4 points) Order the following assertions from strongest to weakest:

    (a) x is even and y = x + 1

    (b) y is not even

    (c) x is even and y is odd

    (d) x = 10 and y = 11

    Answer: _____, _____, _____, _____

18. (4 points) Make a true statement by filling in each blank with one of the words "abstract", "concrete", "derived", "implementation", and "specification":

    A _____ field can be computed from one or more _____ fields.

3 Code reasoning

19. (14 points) Prove that the `trailingZeros` procedure returns a correct answer, assuming it terminates.

    You may use any part of this page for your answer, but ensure that your solution is readable for grading. Hint: your answer should have four clearly-marked parts.

    Hint: try thinking of -882340000 as -88234 * 10^4. Note that -88234 % 10 $\neq$ 0.

```
/**
 * Requires: x != 0
 * Returns the number of trailing zeros of x -- that is, the number of 0s at the
 * end of the string representation of x. For example:
 *   trailingZeros(1) == 0
 *   trailingZeros(10) == 1
 *   trailingZeros(100) == 2
 *   trailingZeros(29831) == 0
 *   trailingZeros(-882340000) == 4
 */
public static int trailingZeros(int x) {

    int zeros = 0;

    while (x % 10 == 0) {

        x = x/10;

        zeros = zeros + 1;
    }

    return zeros;
}
```

20. (14 points) Prove that the `trailingZeros` procedure terminates. It's the same procedure as in problem 19.

    You may use any part of this page for your answer, but ensure that your solution is readable for grading. Hint: your answer should have three clearly-marked parts.

```
/**
 * Requires: x != 0
 * Returns the number of trailing zeros of x -- the number of 0s at the
 * end of the string representation of an integer. For example:
 *   trailingZeros(1) == 0
 *   trailingZeros(10) == 1
 *   trailingZeros(100) == 2
 *   trailingZeros(29831) == 0
 *   trailingZeros(-882340000) == 4
 */
public static int trailingZeros(int x) {

    int zeros = 0;

    while (x % 10 == 0) {

        x = x/10;

        zeros = zeros + 1;
    }

    return zeros;
}
```

21. (20 points) Consider the `IntMap` interface and (unrelated) `IntStack` class from page 9.

    Willy Wazoo wants to write his own implementation for `IntMap`, but the only data structure he knows how to use is an `IntStack`! So he started out like this before he got stuck:

```
class WillysIntMap implements IntMap {

    // Represents the IntMap
    private IntStack theRep;

}
```

    Help Willy write a rep invariant and abstraction function for his implementation. Don't change his representation. Do not write the implementation itself. It must be possible to implement `IntMap` using the rep invariant and abstraction function, but don't worry about the efficiency of the implementation. If you don't see how to implement `IntMap`, answer the questions as well as you can without regard for the implementation.

    (a) Rep invariant:

    (b) Abstraction function:

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

```
/** An IntMap is a mapping from integers to integers.
 * It implements a subset of the functionality of Map<int,int>.
 * All operations are exactly as specified in the documentation for Map.
 *
 * IntMap can be thought of as a set of key-value pairs:
 *
 * @specfield pairs == { <k1, v1>, <k2, v2>, <k3, v3>, ... }
 */
interface IntMap {
  /** Associates the specified value with the specified key in this map. */
  bool put(int key, int val);
  /** Removes the mapping for the key from this map if it is present. */
  int remove(int key);
  /** Returns true if this map contains a mapping for the specified key. */
  bool containsKey(int key);
  /** Returns the value to which the specified key is mapped, or 0 if this
   * map contains no mapping for the key. */
  int get(int key);
}

/**
 * An IntStack represents a stack of ints.
 * It implements a subset of the functionality of Stack<int>.
 * All operations are exactly as specified in the documentation for Stack.
 *
 * IntStack can be thought of as an ordered list of ints:
 *
 *   @specfield stack : List<int>
 *
 *   stack == [a_0, a_1, a_2, ..., a_k]
 */
interface IntStack {

  /** Pushes an item onto the top of this stack.
   * If stack_pre == [a_0, a_1, a_2, ..., a_(k-1), a_k]
   * then stack_post == [a_0, a_1, a_2, ..., a_(k-1), a_k, val].
   */
  void push(int val);

  /**
   * Removes the int at the top of this stack and returns that int.
   * If stack_pre == [a_0, a_1, a_2, ..., a_(k-1), a_k]
   * then stack_post == [a_0, a_1, a_2, ..., a_(k-1)]
   * and the return value is a_k.
   */
  int pop();
}
```