Name:____

CSE331 Spring 2015, Final Examination
June 8, 2015

Please do not turn the page until 8:30.

Rules:

- The exam is closed-book, closed-note, etc.
- **Please stop promptly at 10:20.**
- There are **158 points (not 100)**, distributed **unevenly** among **11** questions (all with multiple parts):

Question	Max	Earned
1	21	
2	15	
3	8	
4	21	
5	8	
6	12	
7	9	
8	17	
9	17	
10	10	
11	20	

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.** Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (**21** points) Here is correct Java code for a mutable ADT that maintains the average of a collection of ints.

```java
class IntAverage {
    private int count = 0;
    private int sum = 0;

    // add the argument x to the collection whose average is maintained
    public void add(int x) {
        sum += x;
        count++;
    }

    // return the current average, or 0 if no ints have been added
    // (truncates using integer division)
    public int average() {
        if(sum==0) // don't misread this line
            return 0;
        return sum / count;
    }
}
```

To reason about this ADT would require a representation invariant and an abstraction function. This problem concerns only the representation invariant. Here are seven proposed representation invariants:

(a) `count >=0`

(b) `count > 0`

(c) `true` (the logical assertion satisfied by all program states)

(d) `if count==0 then sum==0`

(e) `if sum==0 then count==0`

(f) `count >=0 and (if count==0 then sum==0)`

(g) `count >=0 and (if sum==0 then count==0)`

For each proposed invariant above, indicate which one of the statements **A**–**E** below is true (no explanation required). Assume ints can be positive or negative, but ignore overflow.

A. The invariant is incorrect because it may not hold initially.

B. The invariant is incorrect because it holds initially but may not hold later.

C. The invariant is correct but is not strong enough to prove that division-by-zero (which causes an `ArithmeticException`) will not occur.

D. The invariant is correct but cannot be used to prove that it is preserved by all operations.

E. The invariant is correct, the invariant is strong enough to prove that no division-by-zero can occur, and we can use it to prove that the invariant is preserved by all operations.

Solution:
(a) C (b) A (c) C (d) D (e) B (f) E (g) B
(For (e) and (g), it's because ints can be negative or 0. For (d), it's because count could, according to the invariant, be -1 before calling add.)

Name:____

2. (**15** points) Consider these two interfaces and their specifications:

```
// A list of ints where each int is at a position where the first
// position is 0, second position is 1, etc.
interface IntList {

   @requires receiver has >= i elements
   @effects puts x at position i, moving all elements at positions >= i one position higher
   @modifies this
    void insertAt(int x, int i);

   @effects puts x at some unspecified position i, moving all elements
            at positions >= i one position higher
   @modifies this
    void insert(int x);

   @requires receiver has > i elements
   @returns the int currently at position i
    int get(int i);
}

// A sorted list of ints where each int is at a position where the first
// position is 0, second position is 1, etc. Ints are in increasing order.
interface SortedIntList {

   @requires receiver has >= i elements and the effect of the operation
             maintains a sorted list (i.e., i is a "legal" position to add x)
   @effects puts x at position i, moving all elements at positions >= i one position higher.
   @modifies this
    void insertAt(int x, int i);

   @effects puts x at some position i that maintains sorted order (moving
            all elements previously at position i or greater to one position higher).
   @modifies this
    void insert(int x);

   @requires receiver has > i elements.
   @returns the int currently at position i.
    int get(int i);
}
```

For both parts below, **explain your answer**. Refer to particular parts of the specifications above. You might find it helpful to show and explain example client code as well, but it is not required.

(a) Is `SortedIntList` a *true subtype* of `IntList`?

(b) Is `IntList` a *true subtype* of `SortedIntList`?

Solution:

(a) No. The specification for `insertAt` in `SortedIntList` is weaker than in `IntList`. It is weaker because it has a stronger requires-clause, forcing clients to choose an `i` that keeps sorted order. A true subtype cannot weaken a specification; clients cannot substitute the subtype for the supertype because they may be relying on the stronger specification (in this case, that any `i` that is not larger than the list length is okay).

(b) No. The specification for `insert` in `IntList` is weaker than in `SortedIntList`. It is weaker because it provides no constraint on where the item is inserted (no longer necessarily maintains sorted order). A true subtype cannot weaken a specification; clients cannot substitute the subtype for the supertype because they may be relying on the stronger specification (in this case, that sorted order will be maintained).

Name:____

3. (**8** points)

   (a) Is **Java** subtyping *transitive*? Explain your answer in 1-3 sentences.

   (b) Is **true** subtyping *transitive*? Explain your answer in 1-3 sentences.

   Solution:

   (a) Yes. Java subtyping follows from extends-clauses and implements-clauses. For example, if we have `class A extends B implements I`, then `A` is a subtype of `B` and `I`. More relevant here is if `class C extends B`, then `C` is also a subtype of `B` and `I` and similarly when we have a chain of interfaces extending other interfaces: by Java's definition, all subtyping is transitive.

   (b) Yes. True subtyping is about substitutability, i.e., meeting a stronger specification. If any "A" can be substituted for any "B" and any "B" can be substituted for any "C," then it must be that any "A" can be substituted for any "C".

Name:____

4. (**21** points) Consider these six class definitions, where we have omitted fields, other methods, method bodies, and constructors. (We assume each class has a zero-argument constructor, not shown.)

```
class Matching<T1,T2> {
    public void addPair(T1 x, T2 y) { ... }
    public T2 getMatchLeft(T1 x) { ... }
    public T1 getMatchRight(T2 x) { ... }
}
class OptionallyLabeledMatching<T1,T2,T3> extends Matching<T1,T2> {
    public void setLabel(T1 x, T2 y, T3 s) { ... }
    public T3 getLabel(T1 x, T2 y) { ... }
}
class Socks { ... }
class FancySocks extends Socks { ... }
class Shoes { ... }
class PrettyShoes extends Shoes { ... }
```

(a) List *all* Java subtyping relationships among the 6 types below. For example, if **A** is a Java subtype of **B**, write, "**A** subtype of **B**." You do *not* need to say that types are subtypes of themselves.

A. `Matching<Socks,Shoes>`
B. `Matching<FancySocks,PrettyShoes>`
C. `OptionallyLabeledMatching<Socks,Shoes,Number>`
D. `OptionallyLabeledMatching<Socks,Shoes,Integer>`
E. `OptionallyLabeledMatching<FancySocks,PrettyShoes,Number>`
F. `OptionallyLabeledMatching<FancySocks,PrettyShoes,Integer>`

(b) Here is client code that does not type-check. For *each* line in the code, circle "yes" if that line type-checks or "no" if it would produce a type-checking error. Note:

- It is fine for a method call to ignore a non-void result.
- If a line declaring a variable does not type-check, still assume the variable has the declared type in later lines (so a use of the variable may or may not type-check).

```
Socks s = new Socks();              //  yes    no
PrettyShoes p = new PrettyShoes();  //  yes    no
Number n = new Integer(42);         //  yes    no
OptionallyLabeledMatching<Socks,Shoes,Integer> m
    = new OptionallyLabeledMatching<Socks,Shoes,Integer>(); //  yes    no
Matching<Socks,Shoes> m2 = m;                               //  yes    no
OptionallyLabeledMatching<Socks,Shoes,Number> m3 = m;       //  yes    no

m.addPair(s, p);     //  yes    no
m2.getMatchLeft(s);  //  yes    no
m3.addPair(s,p);     //  yes    no
m.setLabel(s,p,n);   //  yes    no
m3.setLabel(s,p,n);  //  yes    no
m.getLabel(s,p);     //  yes    no
m2.getLabel(s,p);    //  yes    no
```

Solution:

(a) C is a subtype of A
D is a subtype of A
E is a subtype of B
F is a subtype of B

(b)
```
Socks s = new Socks();              //  yes
PrettyShoes p = new PrettyShoes();  //  yes
Number n = new Integer(42);         //  yes
OptionallyLabeledMatching<Socks,Shoes,Integer> m
    = new OptionallyLabeledMatching<Socks,Shoes,Integer>(); //  yes
Matching<Socks,Shoes> m2 = m;                               //  yes
OptionallyLabeledMatching<Socks,Shoes,Number> m3 = m;       //         no

m.addPair(s, p);     //  yes
m2.getMatchLeft(s);  //  yes
m3.addPair(s,p);     //  yes
m.setLabel(s,p,n);   //         no
m3.setLabel(s,p,n);  //  yes
m.getLabel(s,p);     //  yes
m2.getLabel(s,p);    //         no
```

Name:____

5. (**8** points) This method does not type-check:

```
static <T> boolean hasTies(T[] arr) {
     for(int i=0; i < arr.length-1; i++) {
         for(int j=i+1; j < arr.length; j++) {
             if(arr[i].compareTo(arr[j])==0)
                 return true;
         }
     }
     return false;
 }
```

(a) What line produces a type-checking error?

(b) How would you add a type bound to the code above so that the method type-checks? Be completely precise about what change or changes you would make.

Solution:

- The line with the call to `compareTo` because we do not know that array elements (which have type `T`) contain such a method.
- Change `<T>` on the first line to `<T extends Comparable<T>>`.

Name:____

6. (**12** points)

   (a) For each of the Java method signatures below, give an equivalent method signature that does not use wildcards.

   i. `boolean m1(Set<? super Foo> x, Foo y);` *This turns out to be a bad question. See the sample solution for why.*

   ii. `void m2(Set<? extends Foo> x, Set<? extends Foo> y, boolean z);`

   (b) Consider these two method signatures:

   A. `void m(Set<?> x);`

   B. `void m(Set<Object> x);`

   i. Give an example client that can use **A** but cannot use **B**. (Assume there is already a variable `obj` defined that refers to an object containing method `m`, so you can call `obj.m(...)`.)

   ii. Give an example method body that can be used for **B** but not for **A**.

Solution:

(a) (Any names for type parameters can be used.)

i. The expected answer that your instructor thought was correct is:
`<T super Foo> boolean m1(Set<T> x, Foo y);`
Most people got this or something close, but, somewhat embarrasingly, it's wrong. This is actually an impossible question — Java supports lower bounds (bounds with `super`), only for wildcards, so you cannot remove wildcards like this. (The problem would work fine with `extends` instead of `super`.) Much *more* embarrasing is that the lecture materials have suggested you can bound a (non-wildcard) type parameter with `super` for about 4 years and only this exam question caused 2 students to notice and bring it to the attention of your instructor. Sigh!
A lot of online sites suggesting *why* Java does not support the sample solution are not very high quality. It seems that it *could* be supported *and* is occasionally useful, at least in theory, but is not useful enough in practice to complicate the language, according to "those in charge." Your instructor would prefer the uniformity even if in practice it's only marginally useful.

ii. `<T1 extends Foo,T2 extends Foo> void m2(Set<T1> x, Set<T2> y, boolean z)`

(b) i. Can use any type other than `Object` in place of `Integer`.

```
Set<Integer> s = new Set<Integer>();
obj.m(s);
```

ii. Any type, including `Object`, can be used here. The point is we cannot add to a `Set<?>`.

```
x.add(new Integer(42));
```

(Though not intended, we also can give full credit for assigning `x` to a variable of type `Set<Object>` since it does answer the question correctly.)

Name:____

7. (**9** points)

   (a) Explain in roughly 2–3 English sentences why high-quality regression testing makes debugging easier.

   (b) Explain in roughly 2–3 English sentences how the debugging process we advocated in lecture leads to better regression testing.

   Solution:

   (a) By running a good suite of regression tests often, we can discover when a test that used to pass no longer does quickly. When that happens, the reason must be in some way code that recently changed (since before the change the test passed). The code that changed thus is a big clue that helps localize the defect. (It is likely but not guaranteed that the defect is in the code that changed.)

   (b) An early step is to produce a minimal reproducible test that exhibits the failure. A late step is to add this test to the regression-test suite. Hence in the future if a defect similar to the one fixed is reintroduced, regression testing will be able to detect it.

Name:____

8. (**17** points) Below is working code for a small application that includes a text box that is spell-checked, with misspelled words colored red. The contents of the text-box are simply re-spell-checked every time the text-box contents are edited. You may want to rip this page out of your exam. On the next page, you will rewrite the application to *invert a dependency*.

```
class Main {
    private SpellCheckedTextBox b = new SpellCheckedTextBox("English");
    // ... use b in various methods in the application
}
class Dictionary {
    public static Dictionary findDictionary(String language) { ... }
    public boolean contains(String s) { ... }
    ...
}
class SpellCheckedTextBox {
    private Dictionary dictionary;
    private TextBox textbox;
    public SpellCheckedTextBox(String language) {
        dictionary = Dictionary.findDictionary(language);
        textbox = new TextBox();
    }
    public void addLetter(char c, int position) {
        textbox.addLetter(c,position);
        performSpellCheck();
    }
    public void deleteLetter(int position) {
        textbox.deleteLetter(position);
        performSpellCheck();
    }
    public void performSpellCheck() {
        String[] allWords = textbox.getAllWords();
        Set<String> wrongWords = new HashSet<String>();
        for(String w : allWords) {
            if(!dictionary.contains(w))
                wrongWords.add(w);
        }
        textbox.resetFormatting();
        for(String w : wrongWords)
            textbox.formatMisspelledWord(w);
    }
}
class TextBox {
    private StringBuffer text;
    public TextBox() { text = new StringBuffer(); }
    public void addLetter(char c, int position) { text.insert(position,c); }
    public void deleteLetter(int position) { text.delete(position,position+1); }
    public String[] getAllWords() {
        return text.toString().trim().split("\\s+"); // no need to understand this line in detail
    }
    public void resetFormatting() {
        // change all text to Black (details not shown)
    }
    public void formatMisspelledWord(String word) {
        // change all occurrences of word to be Red (details not shown)
    }
}
```

Name:____

Previous problem continued:

On the previous page, `Main` depends on `SpellCheckedTextBox` and `SpellCheckedTextBox` depends on `TextBox`. Here you will rewrite the application, so `Main` depends on `TextBox` and `TextBox` depends on `SpellChecker` and an interface you will define. Your rewritten application should:

- Have the same behavior as the original one, with re-spell-checking after each edit
- Use the given code below
- Define only the interface and class requested

Given code:

```
class Main {
    private TextBox b = new TextBox("English");
    // ... use b in various methods in the application
}
class Dictionary { /* no change from previous page */ }
class SpellChecker {
    private Dictionary dictionary;
    public SpellChecker(String language) {
        dictionary = Dictionary.findDictionary(language);
    }
    public void performSpellCheck(TextHolder textholder) {
        String[] allWords = textholder.getAllWords();
        Set<String> wrongWords = new HashSet<String>();
        for(String w : allWords) {
            if(!dictionary.contains(w))
                wrongWords.add(w);
        }
        textholder.resetFormatting();
        for(String w : wrongWords)
            textholder.formatMisspelledWord(w);
    }
}
```

(a) Write an interface `TextHolder` containing exactly 3 methods such that `SpellChecker` type-checks.

(b) Write a full `TextBox` class to complete the application (omitting only the same details that the previous page does). You can, of course, add field(s), change method bodies, and implement an interface as needed.

The next page is blank in case you need more room.

Name:____

More room if needed for previous problem.

Solution:

```
interface TextHolder {
    String[] getAllWords();
    void resetFormatting();
    void formatMisspelledWord(String word);
}

class TextBox implements TextHolder {
    private StringBuffer text;
    private SpellChecker spellchecker;

    public TextBox(String language) {
        text = new StringBuffer();
        spellchecker = new SpellChecker(language);
    }
    public void addLetter(char c, int position) {
        text.insert(position,c);
        spellchecker.performSpellCheck(this);
    }
    public void deleteLetter(int position) {
        text.delete(position,position+1);
        spellchecker.performSpellCheck(this);
    }
    public String[] getAllWords() {
        return text.toString().trim().split("\\s+");
    }
    public void resetFormatting() {
        // change all text to Black (details not shown)
    }
    public void formatMisspelledWord(String word) {
        // change all occurrences of word to be Red (details not shown)
    }
}
```

Name:____

9. Short Answer on Design Patterns (**17** points)

   (a) Give the two limitations of Java constructors that motivate the creational design patterns we studied.

   (b) For each of your answers to part (a), give two design patterns that work around an aspect of the limitation. (So you'll list four design patterns, two for each limitation.) No explanation required.

   (c) For the Visitor design pattern, let's consider visitors (i.e., implementors of the appropriate `Visitor` interface) to be clients. From this perspective, does the design pattern use *synchronous callbacks*, *asynchrounous callbacks*, *both*, or *neither*. Briefly explain your answer.

   Solution:

   (a) (1) A constructor cannot produce an instance of a subclass. (2) A constructor has to produce a new object.

   (b) (1) Any of Factory Method, Factory Object, Prototype, or Dependency Injection. (2) Any of Singleton, Interning, and Flyweight.

   (c) synchronous callbacks: We callback the object (via the visit methods), passing objects being traversed (via this). The callback is synchronous because it completes before the call from the client completes.

Name:____

10. (**10** points) Short Answer on Systems Integration and Software Teams

    (a) In roughly one sentence, why does adding software developers to a project that is behind schedule often not improve progress?

    (b) Are *stubs* used in top-down development or bottom-up development?

    (c) *What* is a stub?

    (d) *Why* do we implement stubs?

    Solution:

    (a) It can consume more time/effort than it saves since new developers have to learn the system design and needs, as well as communicate with other members of the team.

    (b) top-down

    (c) An incomplete/incorrect implementation of a module that is "good enough" for testing other modules that (will eventually) depend on the module

    (d) So that we can test modules or prototype systems that depend on a module that has not yet been implemented — without stubs, we cannot test a systems implemented top-down until all the code is wrttien.

Name:______________________________________

11. (**20** points) Still More Short Answer:

(a) Choose one. Putting almost all the code for an application in one module has:

 i. Poor coupling and poor cohesion
 ii. Okay coupling and poor cohesion
 iii. Poor coupling and okay cohesion
 iv. Okay coupling and okay cohesion

(b) For each of the course topics below, are there core concepts directly related to the idea of a stronger versus weaker specification? (Answer yes or no for each.)

 i. Whether an overriding method meets a superclass specification
 ii. Whether it is okay for Java to allow covariant return types in overriding methods
 iii. Guidelines for when it is good style to use method overloading
 iv. Whether one generic type is a subtype of another
 v. Deciding what should be part of the view versus the controller in an application using the MVC design pattern
 vi. The benefits of using an anonymous inner class rather than a named class

(c) Consider overriding `paintComponent` in a class you define as a subclass of a Java Swing class. For each of the following, in the body of `paintComponent`, should you *always*, *sometimes* (i.e., "it depends what you're doing"), or *never* do it?

 i. Call `paintComponent` on all the components contained within `this`.
 ii. Call `super.paintComponent`.
 iii. Open a file containing an image.
 iv. Cast the argument of `paintComponent` to `Graphics2D`.

(d) True or false for each in reference to Java's Swing library:

 i. To register an event listener, you need to use an anonymous inner class.
 ii. It is a bug to register the same object to listen for multiple events in different Swing components.
 iii. In terms of the MVC design pattern, event-listener code is usually part of the controller.

Solution:

(a) ii. Okay coupling and poor cohesion

(b) yes, yes, no, yes, no, no

(c) i. Never
 ii. Always
 iii. Never
 iv. Either Sometimes or Always (it's never a problem, but isn't always needed depending what other methods you use)

(d) False, False, True