CSE 331 Summer 2017 Final Exam

Name _____

The exam is closed book and closed electronics. One page of notes is allowed.

The exam has 6 regular problems and 1 bonus problem. Only the regular problems will count toward your final exam score. The bonus problem will be included in your bonus score, which may be used to break ties if you end up on the border between different grades.

Please **wait to turn the page** until everyone is told to begin.

Score _____ / 79

1. _____ / 17
2. _____ / 10
3. _____ / 14
4. _____ / 14
5. _____ / 6
6. _____ / 12
7. _____ / 6

Problem 1 (Subtypes)

1. Which of the following is **NOT** an example of a way that a super- and sub-class can be tightly coupled? Circle one.

   Subclass may depend on the pattern of self-calls in the superclass.

   Subclass may depend on the order of self-calls in the superclass.

   Subclass may depend on the names of private fields in the superclass.

   Subclass methods could be called when rep invariant does not hold.

2. Call the superclass from the question above A and the subclass B.

   Which of the following is a problem with B as a subclass but would **no longer be** if B were instead written with composition? Circle all that apply.

   B may depend on the pattern of self-calls in A.

   B may depend on the order of self-calls in A.

   B may depend on the names of private fields in A.

   A's methods could be called when its rep invariant does not hold.

   B's methods could be called when its rep invariant does not hold.

3. Which of the following would cause a subclass to **fail** to be a subtype? (Assume each change was legal in Java.) Circle all that apply.

   Subclass adds a new method.

   Subclass removes a method.

   Subclass changes a return type from Number to Integer.

   Subclass changes a return type from Integer to Number.

   Subclass strengthens the precondition of a method.

   Subclass strengthens the postcondition of a method.

4. Which Java keyword disallows a subclass from overriding a method?

5. Which of the following does Josh Bloch suggest **may be** necessary when designing for inheritance? Circle all that apply.

   Declare all private fields final.

   Document self-calls of override-able methods

   Eliminate calls of override-able methods from the constructor.

   Provide access to the internal workings of the class by making private methods or fields protected instead.

Problem 2 (Subtypes II)

Consider the following Java class:

```
/** Represents a mutable sequence of integers. */
public class IntList {
  // RI: list is not null, no list entries are null
  // AF(this) = vals
  private List<Integer> vals;

  // If non-null, stores the current value of toString.
  // This makes most calls to toString take O(1) time.
  private String cachedStr;

  /** Creates an empty list. */
  public IntList() {
    this.vals = new ArrayList<>();
    this.cachedStr = toString();
  }

  /** Adds the given integer to the list. */
  public void add(int val) {
    vals.add(val);

    cachedStr = null;
    cachedStr = toString();
  }

  /** Returns a string description of the values. */
  public String toString() {
    if (cachedStr != null)
      return cachedStr;
    return vals.toString();
  }
}
```

(continued on next page…)

Now, consider the following subclass of `IntList` above:

```
/** Stores a mutable list of integers and their product. */
public class IntListWithProduct extends IntList {
  // RI: prod is the product of all the values (or 1 if empty)
  private int prod;

  /** Creates an empty list. */
  public IntListWithProduct() {
    super();
    this.prod = 1;
  }

  @Override public void add(int val) {
    super.add(val);
    prod *= val;
  }

  // Ex: for [2,3] this will return "[2, 3] (product is 6)".
  @Override public String toString() {
    return super.toString() + " (product is " + prod + ")";
  }
}
```

1. Explain how a method of `IntListWithProduct` can be called when the representation invariant does not hold.

2. What will `cachedStr` contain after the *self-call* of `toString` in the `IntList` constructor? (Note that `int` variables are initially zero before assigned.)

3. What will `new IntListWithProduct().toString()` actually return?

4. Would you say `IntList` was designed to support inheritance?

Problem 3 (Generics)

1. Any class that uses (other) generic classes should usually be generic.

   True False

2. Any class that implements a container ADT should usually be generic.

   True False

3. Suppose that `T` is a generic type parameter. For which of the following types can you perform an `instanceof` check? Circle all that apply.

   ```
   T

   T[]

   List<T>

   List<?>
   ```

4. Which of the following is most useful for working around the lack of co- and contra-variant subtyping in generic parameters? Circle one.

   arrays ArrayList

   generic methods generic type erasure

Let `S` be a class and `T` a subtype. Suppose that our method wants to take a sequence of `S`'s as an argument and use it safely in a **contravariant**[1] manner.

5. Which of these would be legal in Java? Circle all that apply.

   Declare argument as `S[]` and pass in `T[]`

   Declare argument as `List<S>` and pass in `List<T>`

6. Which of these would always work correctly (i.e., run without error) in Java? (Assume each was allowed / legal in Java.) Circle all that apply.

   Declare argument as `S[]` and pass in `T[]`

   Declare argument as `List<S>` and pass in `List<T>`

[1] Recall: `Foo<X>` is covariant in `X` if `Foo<X>` is a subtype of `Foo<Y>` when `X` is a subtype of `Y` and contravariant if `Foo<X>` is a supertype of `Foo<Y>` as `X` is a subtype of `Y`.

Problem 4 (Design Patterns)

1. Which of the following are advantages of static factory methods over constructors? Circle all that apply.

   Has a name that can provide useful information about what it creates

   Can return an existing object instead of creating a new one

   Can have two with the same argument list

   Can return a subtype

2. Which of the following are examples of **wrappers**? Circle all that apply.

Adapter	Decorator
Proxy	Visitor

3. Traversing an abstract syntax tree (AST) using the procedural approach (i.e., with code grouped together by operation rather than node type) demonstrates which of the following design patterns? Circle all that apply.

Adapter	Composite
Interpreter	Visitor

4. Give an example of a design pattern used in the standard Java library for creating instances of `String`. Name both the class / method of the Java library and the pattern it represents.

Problem 5 (Style)

Kevin is writing a Matrix class. He starts writing his class as follows:

```
public class Matrix {

  /** Creates a matrix from a list of values given in
    * row-major order. */
  public Matrix(double[] vals, int rowSize) { ... }

  /** Creates a matrix from a list of values given in
    * column-major order. */
  public Matrix(double[] vals, int colSize) { ... }

  ...
}
```

Why does this fail to compile?

To fix it, Kevin changes the code to the following:

```
public class Matrix {

  /** Creates a matrix from a list of values given in
    * row-major order. */
  public Matrix(double[] vals, int rowSize) { ... }

  /** Creates a matrix from a list of values given in
    * column-major order. */
  public Matrix(int colSize, double[] vals) { ... }

  ...
}
```

Why, even though it now compiles, does it demonstrate bad style?

Problem 6 (Debugging & Testing)

Consider the following class:

```
/** Represents a list of doubles. */
public class DoubleList {

  // RI: vals != null and contains no nulls
  // AF(this) = [vals[0], vals[1], ..., vals[vals.length-1]]
  private Double[] vals;

  /** Creates an empty list */
  public DoubleList() {
    this.vals = new Double[0];
  }

  /** Adds the given value to the end of the list. */
  public void add(double val) {
    // Make a new array containing vals plus the new value.
    Double[] newVals = new Double[vals.length + 1];
    for (int i = 0; i < vals.length; i++)
      newVals[i] = vals[i];
    newVals[vals.length] = val;

    this.vals = newVals;
  }

  /** Return the list as an array. */
  public Double[] asArray() {
    return vals;
  }

  @Override public String toString() {
    StringBuilder buf = new StringBuilder("[");
    for (int i = 0; i < vals.length; i++) {
      if (i > 0)
        buf.append(", ");
      buf.append(Double.toString(vals[i]));
    }
    buf.append("]");
    return buf.toString();
  }
}
```

1. Suppose that the code is crashing with a `NullPointerException` on the line that reads[2]

   ```
   buf.append(Double.toString(vals[i]));
   ```

   With the code above, what object must be null if that exception occurs?

2. Where is the defect that made that object null? Circle one.

   `DoubleList()` client code

   `DoubleList.add` `DoubleList.toString`

3. Most likely, how hard will it be to find this defect versus an average bug?

   harder than average easier than average

4. What should the author of DoubleList have done differently in this class to avoid this failure?

5. Write the method body for a JUnit test that would demonstrate the crash discussed in Problem 6 (i.e., that would fail by crashing in that manner).

[2] The signatures of the methods called here are `append(String)` and `toString(double)`.

Problem 7 (Testing II)

Consider the following class:

```
/** Represents a mathematical set of integers.
  * Typical instances are {}, {1, 2}, and {6, 5, 3}.
  */
public class IntSet {
  ...

  /** Adds the given value to the set. */
  public void add(int val) { ... }

  ...

  @Override public String toString() { ... }
}
```

1. Suppose that we write the following JUnit test:

   ```
   @Test public void testToString() {
     IntSet set = new IntSet();
     set.add(6);
     set.add(5);
     set.add(3);
     assertEquals("{6, 5, 3}", set.toString());
   }
   ```

   What *unspecified* aspects of the behavior of `IntSet.toString` does this test depend on?

2. Why might it **not** be a good idea to add this behavior to the **specification** even though the tests depend on it?