CSE 331 Final Exam 3/16/15 Sample Solution

Question 1. (12 points, 3 each) A short design exercise. Suppose Java did not include a Set class in the standard library and we need to store a set of Strings for an application. We know that the maximum size of the set will be supplied when it is constructed. Because of this we choose to use a Java array of Strings (type `String[]`) to hold the data, and the array will be allocated when the set is created.

(There is way more space here than you probably need – don't feel compelled to use all of it. ☺)

(a) Give a complete declaration of the variables that make up the *representation* (rep) of the set of Strings, including, of course, the array, and any other variables that form part of the rep. You don't need to give any comments here – that's next.

```
private String[] items;
private int size;
```

(b) Give a suitable *representation invariant* (RI) for your set that is appropriate given the variables you've listed in part (a). You do not need to format this as Javadoc comments – just give a short answer.

`Items != null`
`0 <= size <= items.length.`
For 0 <= i, j < `size`, `items[i] != null` and if `i != j`,
`items[i].equals(items[j])` is false.

(i.e., the strings in the set are stored in `items[0..size-1]`, no strings are `null`, and no two strings are equal.)

Note: there was no deduction if you allowed a `null` value in the set as long as the rest of the answers in the question were consistent with that decision.

(continued next page)

Question 1. (cont.) (c) What is the *abstraction function* (AF) for your set, given your answers to the previous parts of the question? As in (b), just give an answer, don't worry about Javadoc formatting.

A `StringSet` is a set of non-null strings such that size is the number of strings in the set (the cardinality) and for `0<=i<size`, `items[i]` is an element of the set.

(It might also be worth adding that the maximum size of the set is given by `items.length`, but that was secondary enough that we let it go during grading.)

(d) Give the implementation of a suitable `checkRep()` method for your set. You should not worry about performance – assume that this method will be used during testing and development.

```
private void checkRep() {
  assert items != null;
  assert 0 <= size && size <= items.length;

  for (int j = 0; j < size; j++) {
    assert items[j] != null;
    for (int k = 0; k < size; k++) {
      if (j != k)
        assert (! items[j].equals(items[k]));
    }
  }
}
```

Question 2. (14 points) Proofs 'R Us. Proofs are useful not only for avoiding bugs when designing programs, but also for locating and fixing problems if they do slip in. We've got a small class that maintains a list of integers using an array. The class heading, instance variables, and representation invariant are as follows:

```
public class TinyList {
  // rep
  private int[] nums;
  private int  size;

  // RI: data is stored in nums[0..size-1] && nums != null &&
  //     0<=size<=nums.length
```

One method in the class returns the sum of the elements in the list. Here is the code:

```
  /** return sum of items in this */
  public int sum() {
    int result = 0;
    int k = 0;
    while (k <= size) {
      result = result + nums[k];
      k = k + 1;
    }
    return result;
  }
```

But we're not sure that it is correct. Give a proof below showing that the code is correct, or, if you discover errors in the code and can't prove that it works, fix the code so it does work properly and provide a proof that demonstrates that it does.

(There is an additional blank page after this one for your work, if you need it.)

See solution on next page.

Question 2. Additional space for code and proof if needed.

There is a bug. The postcondition we want is

{ result = nums[0] + nums[1] + … + nums[size-1] }

The obvious loop invariant for the method, which matches the existing code, is

{ result = nums[0] + nums[1] + … + nums[k-1] }

With the given loop condition of k <= size, when the loop terminates we have

{ inv && k > size } => { result = nums[0] + … + nums[size-1] + nums[size] }

assuming that when the loop terminates we have k = size+1. That adds the spurious array element nums[size] to the result.

The fix is to change the loop condition from k <= size to k != size or k < size. Then the code and proof are as follows (with a few obvious steps omitted):

```
public int sum() {
  int result = 0;
  int k = 0;
  { inv: result = nums[0] + nums[1] + ... + nums[k-1] }
  while (k != size) {
    { inv && k < size }
    result = result + nums[k];
    { result = nums[0] + nums[1] + ... + nums[k] }
    k = k + 1;
    { inv }
  }
  { inv && k = size } => { result = nums[0] + nums[1] + ... + nums[size-1] }
  return result;
}
```

Question 3. (10 points, 2 each) Suppose we have the following specifications for a method that computes the average weight of the items in a list of Things.

Note that `IllegalArgumentException` is a standard library exception and is a subclass of `RuntimeException`

S1: @return average weight of Things in the list

S2: @requires list is not empty
@return average weight of Things in the list

S3: @return average weight of Things in the list or 0.0 if the list is empty

S4: @return average weight of Things in the list
@throws IllegalArgumentException if the list is empty

S5: @return average weight of Things in the list
@throws RuntimeException if the list is empty

(a) List all the specifications above that are stronger than (or equal to) S1.

(b) List all the specifications above that are stronger than (or equal to) S2.

S1, S3, S4, S5

(c) List all the specifications above that are stronger than (or equal to) S3.

(d) List all the specifications above that are stronger than (or equal to) S4.

(e) List all the specifications above that are stronger than (or equal to) S5.

S4

Notes: recall that a specification S is stronger than a specification W if every implementation that satisfies S also satisfies W.

Every specification is trivially equivalent to itself. It would be fine to list that in your answer, but we did not deduct points if you did not include that.

It almost seems that S1 and S3 are equivalent, but only if we assume that the average weight of an empty list is 0.0, which is something that isn't explicitely stated in S1. So neither of them is actually stronger than the other, because an implementation that satisfies either one isn't guaranteed to satisfy the other.

Question 4. (10 points, 1 each) Generic things. Suppose we have the following classes defined to specify the drawing implements in an artist's toolkit:

```
class ArtistTool { ... }
class Brush extends ArtistTool { ... }
class DetailBrush extends Brush { ... }
class Pencil extends ArtistTool { ... }
```

Now suppose we have the following objects and lists:

```
ArtistTool t;
Brush b;
DetailBrush d;
Pencil p;

List<Brush> lb;
List<? extends Brush> leb;
List<? super Brush> lsb;
```

For each of the following, circle OK if the statement has the correct Java types and will compile without error; circle ERROR if there is a type checking error of some sort.

(OK) ERROR `lb.add(d);`

(OK) ERROR `lsb.add(d);`

OK (ERROR) `leb.add(d);`

OK (ERROR) `t = lsb.get(0);`

(OK) ERROR `t = leb.get(0);`

(OK) ERROR `b = leb.get(0);`

OK (ERROR) `d = leb.get(0);`

OK (ERROR) `lsb.add(p);`

(OK) ERROR `lsb.add(b);`

OK (ERROR) `leb.add(b);`

Question 5. (14 points) In Homework 4 we had several classes to represent rational numbers and polynomials. One key class was `RatNum`, which was a subclass of Number. Suppose we were to add a new subclass of `RatNum` to represent integer values:

```
public class RatInt extends RatNum {
  public RatInt(int n) {
    super(n,1);
  }
  // ... rest omitted
}
```

For each of the following pairs of types, describe the **Java subtype relationships** that exist between them. If the types are related by subtyping, describe which type is a subtype of the other. If the types are not related (i.e., invariant subtyping) or are the same, say so.

Example: if the types are `RatInt` and `RatNum`, the answer is that `RatInt` is a (Java) subtype of `RatNum`.

(a) (2 points) `List<RatNum>` and `List<RatInt>`

Invariant (not related)

(b) (2 points) `List<RatInt>` and `ArrayList<RatInt>`
(recall that `ArrayList<E>` implements `List<E>`)

`ArrayList<RatInt>` is a Java subtype of `List<RatInt>`

(c) (2 points) `RatNum[]` and `RatInt[]` (i.e., Java arrays)

`RatInt[]` is a Java subtype of `RatNum[]`.

(Note that it is not a true [specification] subtype, but it is a subtype according to the Java language rules.)

(continued next page)

Question 5. (cont.) `RatNum` and `RatInt` continued. Recall that `RatNum` contained the following method for producing a new `RatNum` that is the sum of two `RatNums`:

```
public RatNum add(RatNum arg) { ... }
```

We want to include an `add` method in `RatInt`. Here are several possibilities. For each one, indicate whether adding this method to `RatInt` results in `RatInt` being a **true specification subtype** of `RatNum`. If so, answer yes. If not, answer no and give a brief reason why. If the method would result in `RatInt` being a true subtype of `RatNum`, but not a legal Java subtype, you should say so and describe why. (2 points each)

(d)
```
@Override
public RatInt add(RatInt other) {
  return new RatInt(this.intValue() + other.intValue());
}
```

Not a specification subtype (argument type is a subtype of `RatNum`). (Also does not properly override method `add` for Java subtyping.)

(e)
```
/** @require other represents an integer with its numerator
 *  divisible by its denominator   */
@Override
public RatNum add(RatNum other) {
  return new RatInt(this.intValue() + other.intValue());
}
```

Not a specification subtype because it is weaker than the original specification (extra precondition). (It is a legal Java subtype.)

(f)
```
@Override
public RatNum add(RatNum other) {
  if (other instanceof RatInt) {
    return new RatInt(this.intValue() + other.intValue());
  } else {
    return super.add(other);
  }
}
```

Yes. Proper specification subtype.

(g)
```
@Override
public RatInt add(RatNum other) {
  if (other instanceof RatInt) {
    return new RatInt(this.intValue() + other.intValue());
  } else {
    return super.add(other);
  }
}
```

The method heading would be a proper specification subtype (ok to return a subtype of the original method result), but this is not a legal Java method because super.add returns a RatNum, not a RatInt.

Question 6. (4 points) Suppose we have the following class in a game. The idea is that this class implements the one unique object representing the game world.

```
public class World {
   private static World instance;

   public World() { /* details omitted */ }

   public static World getWorld() {
      if (instance == null) {
         instance = new World();
      }
      return instance;
   }

   // other fields and methods omitted
}
```

Is this a correct implementation of the singleton pattern? Why or why not (in one or two sentences)?

No. The public constructor allows clients to create multiple instances of the class.

Question 7. (4 points) You've been hired to create a shopping application, and the application needs a class to represent a shopping list containing the items the customer wants to buy. You've started to create a small class to do this and so far here's what it looks like:

```
public class ShoppingList {
  private List<String> items;

  /** Create new empty shopping list */
  public ShoppingList() {
    items = new ArrayList<String>();
  }

  /** Add an item to the shopping list */
  public void add(String item) {
    items.add(item);
  }
}
```

The hot-shot new programmer Ben Bitdiddle says that your code would be much better and shorter if you used inheritance instead of composition, and the whole job could be done with a single line of code:

```
public class ShoppingList extends ArrayList<String> { }
```

Both versions "work" in the sense that they provide a way of storing a list of items. Is one of them better than the other? If so which one? Why? (Give a short reason to justify your answer.)

The second version that extends `ArrayList` is a poor design since it implies that a `ShoppingList` is a specification subtype of `ArrayList` and we should be able to substitute instances of `ShoppingList` anywhere an `ArrayList` is expected. Even though it might "work", it is an inappropriate abstraction. The first version, using composition, is the correct design.

Question 8. (4 points) As the new programmer at Whatsamatta U., you've been put in charge of maintaining the old software that manages academic records for the university. A lot of the data is stored in a large class that contains the following data and methods, among much else.

```
/** store information about students, teachers, and classes */
public class SchoolRecords {
  // information about teachers and students, keys = names
  private Map<String, Integer> teacherIDs;
  private Map<String,Integer>  teacherYearsEmployeed;
  private Map<String, Integer> studentIDs;
  private Map<String,Integer>  studentYearsEnrolled;

  // map teacher names to courses taught
  Map<String,Set<String>> coursesTaught;

  // map course names to ratings
  Map<String, List<Double>> courseRatings;

  // map student names to current courses
  Map<String, Set<String>> courseEnrollments;

  // add and retrieve information about courses taken and taught
  public void studentAddClass(String student, String course){...}
  public void teacherAddClass(String teacher, String course){...}
  public List<String> getStudentClasses(String student) { ... }
  public List<String> getTeacherClasses(String student) { ... }
  public void recordRating(String course, Double rating) { ... }
  public Double getAverageRating(String course) { ... }

  // ... many more ...
}
```

In terms of design, what is wrong with this class and how should it be fixed? Just give a brief description of the basic design problem(s) and solution(s) – you are not being asked to actually fix the code.

Terrible cohesion. The class has several unrelated concepts in it (student schedules, teacher schedules, teacher employment histories, course ratings, and more). These different abstractions should be split out into separate classes.

Question 9. (8 points, 2 each) There are several ways to handle unusual or unexpected situations in a program. For the situations below, indicate which of the following would be the most appropriate way to handle or check for the situation in a typical Java program by writing one of the following choices below each description:

- Use a `throw` statement (i.e,. to throw an exception)
- Use an `assert` statement
- Specify a precondition
- Return a special value (e.g., -1 or `null`)
- Ignore it

(a) After opening a file and reading some data from it, attempting to read more data fails because of an I/O error.

Throw an exception

(b) When searching a list, the value we are looking for is not found.

Special value

(c) The contents of a list need to be in ascending order for binary search.

Precondition

(d) A violation of the rep invariant is discovered in your `checkRep` method.

assert statement

Question 10. (10 points) Debugging. We have a small method with a bug in it:

```
/** return absolute value of x */
public static int abs(int x) {
  if (x <= 1) {
    return -1 * x;
  }
  return x;
}
```

For this question, fill in the steps needed to properly diagnose and fix the bug.

(a) (4 points) Write a small, repeatable test case that demonstrates the failure. You should write this as a proper JUnit test (just the single test method – you don't need to write an entire JUnit class). The `@Test` annotation is written for you:

```
@Test
public void testOne() {
   assertEquals(1, abs(1))
}
```

(b) (2 points) What is the defect in the original code that causes the problem? (A brief description is all that is needed.)

Treats +1 as a negative number and returns -1 as its absolute value.

(c) (2 points) What change(s) is(are) needed in the code to fix the defect? (You can either give a brief description or rewrite the code with appropriate changes.)

Change `if (x<=1)` to `if (x<1)` (or `if (x<0)`).

(d) (2 points) After updating the code in the repository, what else should you do before you're finished with this bug, if anything? If nothing further needs to be done, write "nothing".

Add the new test to your collection of regression tests and perform regression tests.

Question 11. (10 points, 2 each) A few short questions to finish up.

(a) What does DRY stand for?

Don't Repeat Yourself

(b) Copy-and-paste coding is when you solve the same problem in more than one place by duplicating code. Briefly describe one disadvantage of doing this.

If changes need to be made or bugs fixed, it is easy to overlook one of the copies that needs to be modified.

(c) What is regression testing? What is the reason for doing it?

Rerun old (existing) tests on newly modified code. It increases our confidence that the changes have not created problems or caused old errors to reappear.

Note that this is not done only after fixing a bug or only about checking to see if old bugs have resurfaced. It is running all old tests to verify that changes made for any reason leave the code still working as expected, including not recreating old bugs that have been previously fixed.

(d) Give a brief reason in terms of module design or testing or some other concept(s) we've discussed about why widespread use of global variables is generally a bad idea in a program.

A global variable creates coupling between all of the modules that can potentially access it. Tightly coupled code is hard to get right, modify, and debug, so we want to decrease coupling as much as possible.

(e) You are in managing a large software project. One of the essential components is a software layer to handle network traffic from thousands of customers at the same time. You're not sure whether it will be possible to handle that much traffic using the software and hardware available. To minimize risk, would it be better to build the system top-down or bottom-up? (No explanation needed, just give your answer.)

Bottom-up.

Have a great spring break! The CSE 331 staff