**cs2220: Exam 2**

**Name:**

UVa Email ID:

Out: 18 November

Due: Tuesday, 23 November (3:31pm)

**Directions**

**Work alone.** You may not discuss these problems or anything directly related to the material covered by this exam with anyone except for the course staff between when this exam is posted and after class Tuesday. You may not obtain any help from other humans except for the course staff (who will only answer clarification questions). You may, and should, continue to work in your project teams during the exam period, but should be careful not to discuss anything directly related to the exam questions.

**Open resources.** You may use any books you want, lecture notes, slides, your notes, and problem sets. You may use Eclipse and Java any other language interpreter you want. You may also use external non-human sources including books and web sites. If you use anything other than the course book, problem sets, slides, and notes, you should cite what you used.

**Answer well.** Answer all four questions (including all sub-parts). You may either: (1) write your answers on this exam or (2) type and write your answers into the provided Word template:

*http://www.cs.virginia.edu/cs2220/exams/exam2/exam2.docx*

Whichever you choose, you must turn in your answers printed on paper and they must be clear enough for us to read and understand. If you use the template, keep the pagination similar to the original exam.

The spaces are designed to be large enough to fit a full-credit answer, but if you want more space you may use the backs or attach extra sheets. If you do, make sure the answers are clearly marked. The questions are not necessarily in order of increasing difficulty. There is no time limit on this exam, but it should not take a well-prepared student more than two hours to complete, and is designed to be short enough to still allow you time to work on your projects while the exam is out. It may, however, take you longer, so please do not delay starting the exam.

Full credit depends on the clarity and elegance of your answer, not just correctness. Your answers should be as short and simple as possible, but not simpler. Your programs will be judged for correctness, clarity and elegance, but you will not lose points for minor syntactic errors.

**Subtyping**

1. Molly Molewhacker wants to build a University simulator starting from the simulator from Problem Set 5. Her University will have buildings (which can contain classrooms and offices and do not move), and students and professors. Classrooms and offices can contain students and professors.

She has two possible designs in mind (all of the boxes in the design are classes, and all the arrows indicated subtype-of relationships):



**Design A**



**Design B**

1. Describe one clear advantage of Design A over Design B.
2. Describe one clear advantage of Design B over Design A.
3. Draw a better design than either Design A or Design B, and explain clearly why your design is better.
4. Consider the following (not very sensical or peaceful) Java code:

**public** **class** Annihilator {

// OVERVIEW: An Annihilator is a mutable object that can kill and be killed by other objects. A

// typical Annihilator is state where state is either Alive, Dying, or Dead.

**private** **int** state;

// A.F.(c): if c.state = 2, Alive; if c.state = 1, Dying; if c.state = 0, Dead.

// RepInvariant(c): 0 <= c <= 2

**public** Annihilator () // EFFECTS: Initializes this to Alive.

{ state = 2; }

**public** **boolean** isAlive() { **return** state == 2; }

**public** **boolean** isDying() { **return** state == 1; }

**public** **boolean** isDead() { **return** state == 0; }

**public** **void** destroy(Annihilator a)

// REQUIRES: a == Dying

// MODIFIES: a

// EFFECTS: a\_post = Dead

{ a.state = a.state - 1; }

// Specification not shown

**public** Annihilator pickStronger(Annihilator a) {

**if** (**this**.state > a.state) { **return** **this**; }

**else** { **return** a; }

}

**public** **void** kill()

// MODIFIES: this

// EFFECTS: this\_post = Dead

{ state = 0; }

}

**public class** Obliterator **extends** Annihilator {

@Override

**public** **void** destroy(Annihilator a)

// REQUIRES: a != Dead

// MODIFIES: a

// EFFECTS: a\_post = Dead.

{ a.kill(); }

}

1. Does the implementation of the Annihlator **destroy** method satisfy its specification? Your answer should either explain clearly why it does not, or use precise reasoning to argue why it does.
2. Does the Obliterator **destroy** method satisfy the substitution principle? A good answer will include a clear and convincing argument supporting your answer.
3. Suppose an Obliterator is always stronger than an Annihilator, so we override the Obliterator **pickStronger** method as:

@Override

**public** Obliterator pickStronger(Annihilator a)

// EFFECTS: If a is not an Obliterator, returns this. Otherwise, returns

// the stronger of this and a (or either one if they are equally strong),

// where Alive is stronger than Dying which is stronger than Dead.

{

**if** (a **instanceof** Obliterator) {

**if** (isAlive() || (isDying() && !a.isAlive())) { **return** **this**; }

**else** { **return** (Obliterator) a; }

} **else** {

**return** **this**;

}

}

Note that we have not provided a specification for the Annihilator **pickStrong** method. *Could* the Obliterator **pickStrong** method satisfy the substitution principle? To answer this, you should either argue that there is no possible specification for the Annihilator **pickStrong** method that would make the Obliterator **pickStrong** method satisfy the substitution principle, or your should provide a suitable specification for the Annihilator **pickStrong** method.

1. **(challenge bonus)** Note that our pickStronger comparisons are now not symmetric: a.pickStronger(b) is not necessarily equal to b.pickStronger(a). Explain a general solution to this problem. For maximum bonus, your answer should include correct code for your solution and an argument why it satisfies the symmetry property, clearly stating any assumptions on which that argument relies.

**Concurrency**

1. Our philosophers from PS5 have decided that is it not natural or efficient to only argue with one other philosopher at a time. Instead, they should be able to argue with as many other philosophers as they want at once. The philosophers sit around a table (represented by the Table class), and take turns making their argument.

The code for our simulation is on pages 10-11. It has the problem though, that philosophers will interrupt each other’s argument. Our philosophers are civilized enough that they always let their colleagues finish their points without interruption. So, only one philosopher at the table may be speaking at once, and once a philosopher starts arguing she gets to finish arguing with everyone before the next philosopher speaks.

1. Explain how to modify the code to prevent this race condition. Your solution to not introduce any deadlocks in the code. For this part, ignore what happens when the philosopher wins enough points to leave the game.)
2. Are there any deadlocks or race conditions in **leaveTable**? If so, explain how to fix them. If not, explain why not.

**class** Table {

// OVERVIEW: A Table is a place where philosophers argue.

}

**class** PhilosopherThread **implements** Runnable {

**private** Philosopher philosopher;

// Rep Invariant: philosopher != null

**public** PhilosopherThread(Philosopher p) {

philosopher = p;

}

**public** **void** run() {

**try** {

**while** (philosopher.philosophize()) {

Thread.*sleep*(1000);

}

} **catch** (InterruptedException e) {

;

}

}

}

**public** **class** Philosopher {

**private** ArrayList<Philosopher> colleagues;

**private** **final** String name;

**private** **final** String quote;

**private** Table table;

**private** **int** points;

// Rep Invariant: name != null, quote != null (colleague may be null)

**public** Philosopher(Table p\_t, String p\_name, String p\_quote) {

table = p\_t;

name = p\_name;

quote = p\_quote;

points = 0;

colleagues = **new** ArrayList<Philosopher>();

}

**public** String getName() { **return** name; }

// REQUIRES: p must be at the same table as this.

// EFFECTS: If p is already a colleague of this, does nothing. Otherwise, adds p as a colleague

// of this, and symmetrically makes this a colleague of p.

**public** **synchronized** **void** addColleague(Philosopher p) {

**assert** p.table == table;

**if** (!colleagues.contains(p)) {

colleagues.add(p);

p.addColleague(**this**);

}

}

**public** **void** goodbye(Philosopher p) {

say("Goodbye " + p.getName());

colleagues.remove(p);

}

**public** **void** leaveTable() {

**for** (Philosopher p : colleagues) {

p.goodbye(**this**);

}

colleagues = **null**;

table = **null**;

}

**public** **void** say(String s) {

System.*err*.println(getName() + " says: " + s);

}

**public** **boolean** philosophize () {

say("My turn!");

**for** (Philosopher p : colleagues) {

say(p.getName() + ", you are wrong! " + quote);

points++;

**if** (points > 20) {

// Ignore this code for question 3a

say("Haha, I win!");

leaveTable();

**return** **false**;

}

}

say("Okay, I'm done.");

**return** **true**;

}

**Language Design**

1. As mentioned in Class 17, Barbara Liskov identified four main problems with Simula that motivated the design of CLU (from Barbara Liskov, *A History of CLU*, 1992). For each item below, explain how well the design of Java addresses the identified problem. Especially good answers will use concrete examples to show how the problem she identified either still exists in Java or has been avoided by Java.
2. “Simula did not support encapsulation, so its classes could be used as a data abstraction mechanism only if programmers obeyed rules not enforced by the language.”
3. Simula associated operations with objects, not with types.
4. Simula “treated built-in and user-defined types non-uniformly. Objects of user-defined types had to reside in the heap, but objects of built-in type could be in either the stack or the heap.”

**End of Exam**