Problem Set 5

Distributed Simulations

(Object-Oriented Design and Concurrency)

Out: 11 October
Design Reviews: (in class) Monday, 16 October
Due: **Wednesday, 25 October**
(beginning of class)

Collaboration Policy (*read carefully*)

For questions 1-6 of this problem set, you should work alone and turn in your own solution. You may discuss your work with other students in the class, but the work you turn in should be your own.

For questions 7-9 you are expected to work in a team of three students. If you would like to form your own team, send me email before 11:59pm on Thursday, October 12, saying who your teammates are. If I don't receive a team request from you before then, or I receive confusing or inconsistent requests, you will be assigned to a team.

Design Reviews

In class on Monday, 16 October, each team will be expected to present their design (question 7). This design should explain: (1) what it is you are simulating; (2) the key modules you will need to implement and their dependencies and inheritance relationships; (3) your implementation strategy including how you will divide the work amongst your team and how you will implement and test your modules. For the design review, you should have a short paper document that answers these questions and be able to present and answer questions about your design.

Reading: Chapter 14.

Purpose

- Understand race conditions and deadlocks.
- Learn to write concurrent programs.
- Learn to design using inheritance.
- Learn to work in a team to develop a program.

In the first part of this assignment (questions 1-6), you will do some exercises that develop your understanding of concurrent programming. In the second part, you will work with your teammates to design and implement a distributed simulation.

Download: ps5.zip.
Extract the zip file and create a new project in Eclipse that contains the extracted files.

Concurrency

The *ps5.zip* file contains an implementation of a grid simulator similar to the cellular automata simulator from Problem Set 1. The important difference is this simulator runs each object in a separate thread. In the cellular automata simulator, there was only one thread, which called the `getNextState` method of each Cell object in the grid in turn and then updated all the cells. In the distributed simulator, each simulated object runs in a separate thread. This means several objects may be taking steps at the same time. (If we are running on a single core machine, then the machine can only execute one instruction at a time, but we get the illusion of multiple things happening at once since the instructions from different threads are interleaved.)

Objects in the simulator are subtypes of the `SimObject` type, partially specified below:

```
abstract public class SimObject implements Runnable {
   // OVERVIEW: A SimObject is an object than represents a simulator
   //    object.  It has a grid (a circular reference to the grid
   //    containing this object), location (integer row and column),
   //    initialization state (boolean that is true if it has beein
   //    initialized), and thread pause state (boolean that is true
   //    if the thread is paused).
```

```
//    A typical SimObject is
//          < grid, row, col, initialized, paused >.

public SimObject()
  // EFFECTS: Creates a new, uninitalized SimObject.

abstract public void executeTurn()
   // REQUIRES: this is initialized
   // MODIFIES: this
   // EFFECTS: Executes one turn for this object.

public Color getColor()
   // EFFECTS: Returns the color that represents this SimObject,
   //    for painting the grid.

final public int getX()
   // REQUIRES: this is initialized.
   // EFFECTS: Returns the x location that this cell is located in.

final public int getY()
   // REQUIRES: init has previously been called for this.
   // EFFECTS: Returns the y location that this cell is located in.

final public void init(int x, int y, Grid grid)
    throws BadLocationException
   // REQUIRES: this is uninitialized
   // MODIFIES: this
   // EFFECTS: If x, y is not a valid location on grid, throws
   //     BadLocationException. Otherwise, initializes the cell at
   //     row, col on grid with isPaused = true.

public void run()
   // REQUIRES: this is initialized
   // MODIFIES: this
   // EFFECTS: Implements the thread for this object.
   //    Note: This method can do anything.
   //       For the base class, if the object is not paused,
   //       it executes one turn by calling the executeTurn method,
   //       and sleeps for a time and repeats. If the object is
   //       paused, does nothing (until the object is unpaused).
```

Note that `SimObject` is an abstract class — we cannot instantiate objects of type `SimObject`. The abstract method `executeTurn` must be implemented by subtypes.

`SimObject` implements the Runnable interface (that is, it is a subtype of `Runnable`) which requires it to implement the `run` method. A `Runnable` object can be used to create a new thread using the `Thread (Runnable)` constructor.

For example, here is the code in the `init` method of `SimObject.java` that starts a thread for each simulated object:

```
Thread thread = new Thread(this);
thread.start();
```

The `this` object passed to `Thread` is a `SimObject` (or a subtype of `SimObject`). When the thread starts, it will call the `run` method of this object.

The Java API documentation specification of `Runnable.run` method is (what folows is the actualy Java Platform API documentation):

`void run()`

```
When an object implementing interface Runnable is used to
create a thread, starting the thread causes the object's run method to
be called in that separately executing thread.

The general contract of the method run is that it may take any action
whatsoever.
```

1. Explain how the specification of our `SimObject` datatype violates behavioral subtyping principles. Is there any way to specify `SimObject` that doesn't violate the substitution principle?

Walker Simulator

Try running the walker simulator now. You can run the simulator by selecting `Run | Run...` in Eclipse, select `Java Application` and `New`. For the `Main Class` select `WalkerSimulator`. Place several `RandomWalkers` in the grid and click "Start". The `RandomWalker` is a subtype of `MobleSimObject` which is a subtype of `SimObject`.

The important classes in the walker simulator are: `Grid`, `GridDisplay`, `Coordinate`, `Direction`, `MobileSimObject`, `RandomWalker`, `SimObject`, `Simulator`, `WalkerSimulator`.

2. Draw a module dependency diagram showing the design for the walker simulation. It should include all the important classes listed above, and show the subtyping ("is-a") and dependency ("has-a" or "uses-a") relationships between the classes. Identify any circular dependencies in the design and explain why they are necessary (or how they could be avoided).

Race Conditions

If you start with enough objects (or are lucky), you will get some `RuntimeExceptions` in the Console like this:

```
java.lang.RuntimeException:
   BUG: SimObject.setLocation - row: 22 col: 27 already occupied.
     at ps5.MobileSimObject.setLocation(MobileSimObject.java:15)
     at ps5.RandomWalker.executeTurn(RandomWalker.java:39)
     at ps5.SimObject.run(SimObject.java:172)
     at java.lang.Thread.run(Thread.java:536)
```

Our grid allows only one object in each square, but we are getting exceptions in `setLocation` when a `MobileSimObject` attempts to wander into a cell that is already occupied. Each unhandled exception terminates the tread raising the exception, but does not shut down the application. The other threads keep executing normally.

What's going on here? A first guess might be that the `RandomWalker` method for `executeTurn` does not check if the new square is empty before moving into it.

Looking at the code, however, we see that that is not the case:

```
public void executeTurn() throws RuntimeException
// EFFECTS: Picks a random direction and tries to move that way.
//          If the move is successful, return true. If the move fails
//          because the spot being attempted to move into is already
//          occupied then return false.
{
    Direction dir = Direction.randomDirection();
    int newy = getY() + dir.northerlyDirection();
    int newx = getX() + dir.easterlyDirection();
```

```
    if (getGrid().validLocation(newx, newy)) {
       if (getGrid().isSquareEmpty(newx, newy)) {
           setLocation(newx, newy);
       }
    }
}
```

(Note: the actual code is slightly different from this to increase the chances of observing the race condition. There is a delay before the call to `setLocation`. The exception handler in the actual code is also omitted here.)

The code calls `getGrid().isSquareEmpty (newx, newy)` to check if the square is empty, and then `setLocation (newx, newy)` to move into the new square. But, what happens if another object moves into that square in the time between this object's `isSquareEmpty(newx, newy)` test and the call to `setLocation(newx, newy)`?

This is an example of a *race condition* — two threads are reading and writing to the same data (in this case, the `Grid` object square). Depending on the order in which the threads execute, the result may be different.

The most common way to prevent race conditions is to use locks to prevent two threads from executing critical regions of code at the same time. We would like to know that between the time this object's thread calls `isSquareEmpty` and the completion of the `setLocation` call, no other thread can alter the state of the grid. In Java, we do this using a synchronized statement:

```
synchronized (expr) {
   statements
}
```

The *expr* must evaluate to an object reference. There is a lock associated with every object. When a `synchronized` statement is reached, the executing thread will evaluate *expr* and attempt to acquire the lock for the object it evaluates to. If the lock is available (that is, no other thread has acquired it), this thread will acquire the lock and hold it until the end of the synchronized block. Hence, any other thread that synchronizes on the same object will stall until this thread releases the lock.

Using `synchronized` in the method header is a short cut for synchronizing on this. So,

```
synchronized public int f () { ... }
```

means the same thing as:

```
public int f () { synchronized (this) { ... } }
```

> **3.** Fix the code for `RandomWalker.executeTurn` so that two MobileSimObjects will never go into the same square. Your fix should not need to modify any code outside the `RandomWalker.executeTurn` method. After your fix, you should be able to run the simulation for as long as you want without ever getting an exception for two objects entering the same square.

Deadlocks

The problem with using locks to prevent race conditions, is that if there are too many locks we can have deadlock instead. The most famous toy problem used to illustrate deadlocks is the "dining philosophers" problem (invented by Edsger W. Dijkstra, and named by Tony Hoare). The problem involves a group of philosophers sitting around a circular table. Each philosopher has a plate of General Tso's chicken, and there is a single chopstick between each pair of philosophers. In order to eat, a philosopher must have two chopsticks. Once a philosopher has finished eating, she puts both chopsticks back down for another philosopher to use. If each philosopher immediately grabs the chopstick to her right, then they will all have one chopstick but no one will be able to eat. This is a deadlock problem, since the philosophers will not put down the chopsticks until they have had a chance to eat. Hence, the philosophers sit around the table and starve.

In our experience, we find philosophers perfer not to share chopsticks, but they do like to philosophize and argue. Consider the `Philosopher` class:

```
// Loosely based on Arnold, Gosling, Holmes p. 252

class PhilosopherThread implements Runnable {
   private Philosopher philosopher;
   // Rep Invariant: philosopher != null

   public PhilosopherThread(Philosopher p) {
      philosopher = p;
   }

   public void run() {
      philosopher.philosophize();
   }
}

public class Philosopher {
   private Philosopher colleague;
   private String name;
   private String quote;
   // Rep Invariant: name != null, quote != null (colleague may be null)

   public Philosopher(String name, String quote) {
      this.name = name;
      this.quote = quote;
   }

   public synchronized void setColleague(Philosopher p) {
      colleague = p;
   }

   public synchronized void philosophize() {
      System.err.println(name   + "[Thread " + Thread.currentThread().getName()
            + "] says " + quote);

      if (colleague != null) { // Need a colleague to start an argument.
         colleague.argue();
      }
   }

   public synchronized void argue()
   // REQUIRES: this.colleague != null
   {
      System.err.println(name
            + "[Thread " + Thread.currentThread().getName()
            + "] argues: No! " + quote);
   }

   public static void main(String[] args) {
      Philosopher descartes =
         new Philosopher("Descartes", "I think, therefore I am.");
      Philosopher plato =
         new Philosopher(
            "Plato",
            "The life which is unexamined is not worth living.");

      descartes.setColleague(plato);
      plato.setColleague(descartes);

      Thread dthread = new Thread(new PhilosopherThread(descartes));
      Thread pthread = new Thread(new PhilosopherThread(plato));
```

```
        dthread.start();
        pthread.start();
    }
}
```

What happens when both philosophers start philsophizing at the same time?

Suppose the `descartes` thread runs first an invokes the `philosophize` method. Since it is declared with `synchronized`, before the method begins executing it must acquire the lock on its this object (in this case, that is the `Philosopher` object `descartes`). Then it calls the `colleague.argue` method. The `argue` method is also declared to be `synchronized`, so it must acquire a lock on its `this` object before proceeding. Here, `argue` is invoked on the object referenced by `colleague` (which is the `plato` object in the example). After the `argue` method finished, it releases the lock on `plato`. Then, the caller, and `philosophize` releases its lock on `descartes`.

But, what would happen if the threads executed in a different order? Suppose the `descartes` thread runs first and invokes the `philosophize` method, and then the `plato` thread invokes its `philosophize` method before the `descartes` thread calls `colleague.argue`. This is called a deadlock.

4. Explain why the execution would get stuck if the threads executed in the order described.

You may want to try running the code to see what is happening more closely. To increase the chances of seeing the deadlock, insert a delay before the call to `colleague.argue`:

```
        try {
            Thread.sleep (500); // Pause for 500 ms.
        } catch (InterruptedException ie) {
            ;
        }
```

Producing multithreaded code that does not have races or deadlocks is very tricky. It is especially difficult since even if the program is tested thoroughly, the problem may appear and disapper based on what other processes running on the machine are doing.

Locking Discipline

One way to prevent deadlocks is to follow a locking discipline. If locks are always acquired in the same order, then we know deadlock can not occur. For the `Philosopher` example, we could require that the lock associated with the alphabetically first philosopher is always acquired first (note that we have removed the `synchronized` from the method header):

```
    public void philosophize () {
        Object lock1, lock2;

        if (colleague != null) { // Need a colleague to start and argument.
            // Always grab the lock for whichever name is alphabetically first
            if (name.compareTo (colleague.name) < 0) {
                lock1 = this;
                lock2 = colleague;
            } else {
                lock1 = colleague;
                lock2 = this;
            }

            synchronized (lock1) {
                synchronized (lock2) {
                    System.err.println (name + "[Thread "
                        + Thread.currentThread().getName () + "] says " + quote);
```

```
                colleague.argue ();
            }
        }
    }
}
```

> **5.** (Tricky, extra credit if you can answer this) Our new `Philosopher` class now has a race condition that could lead to a deadlock or run-time exception (but it would never be apparent from our current main class). Explain what the race condition is, and construct code that reveals it. Feel free to insert sleep pauses as necessary to make it easier to reveal.

The provided code includes a `DrunkPhilosopher` class that is a subtype of the `RandomWalker` type from question 5. A `DrunkPhilosopher` object wanders around until she finds a colleague, and then starts philosophizing. Our `DrunkPhilosopher` suffers from a similar deadlock problem to the `Philosopher` code above. Try running `java PhilosopherSimulator` and observe what happend when two `DrunkPhilosopher` objects encounter each other and deadlock.

> **6.** Fix the `DrunkPhilosopher` class to avoid the deadlock. (Your fix should not involve removing or changing the `delay` calls to make the deadlock less likely.

Distributed Simulation

For the rest of this assignment, your goal is to develop an interesting distributed simulation. You can simulate anything you want.

Be creative! A good simulation will demonstrate some interesting phenomenon or help answer an important social or scientific question. You may with alone, or with any number of students of your choosing (if you can convince people not in cs205 to also contribute to your program, that is fine also, so long as you clearly document what you did).

Listed below are a few ideas for things to simulate, but you are encouraged to think of your own. Since you are working in teams of three, you should be able to implement a simulation with several different types of objects.

- Epidemic — create a simulation that demonstrates how a communicable disease spreads through a society.
- The Lawn on a Fall day — Simulate the Lawn on a summer day. What happens when wandering students, professors, frisbee players, dogs and guided tours interact?
- Pong — implement the classic video game. You will need to figure out how to make object's whose behavior is controlled by the user.
- Trick-or-treating — simulate trick or treating on the Lawn, and determine the best strategy for maximizing the amount of candy obtained. (You can test out your strategy in the real world and compare the results.)
- Stock Market — simulate different types buyers and sellers trading stocks. A useful simulation will provide enough information to predict future price movements in actual stocks (but that is not necessary for cs205).
- Birth of the Universe — simulate the first few nanoseconds of the Big Bang at the sub-atomic level to discover a new unifying theory of physics. (Probably infeasible for this assignment, but a useful result.)

Here are some examples from previous cs201j students:

- Astrosmash by John Franchak — play the classic video game
- Battle by Brian Barrett, Tucker Croft and Darnell Eaton — test your wits in this battle strategy simulator
- Massive Epidemic by Sherlynn Hoon — will the plague destroy society?
- Corner Crawl by Atul Khosla, Heather McGinness, Nancy Fechnay, and Melissa Thomason
- Life Simulator by Marija Cvijetic and Landon Shoop
- Monster Hunter — Sam Baumgardner and Matt Spear
- Tank Hunt — Patrick Rooney, Debra Wesner and Chalermpong Worawannotai

  Note: the source code for these examples is available (usually linked from the example page). If you want to incorporate code from these in your simulation, you may, but you must acknowledge your sources and explain

clearly the changes you made. Obviously, making a small change to a previous answer would not be considered a satisfactory answer for this assignment.

Have fun playing the games...but don't get too distracted from doing your own work!

7. Describe what your simulation will do and your preliminary design. Include:

- A short description of what your simulation will do.
- A modular dependency diagram showing your design (that also shows the subtype relationships).
- A brief description of your implementation and testing strategy, including how you will divide the work amongst your team.

Each team should bring their answer to question 7 to class on Monday, 16 October, prepared to present it and answer questions about it.

8. Implement the simulation you described in question 7. Turn in:

- A modular dependency diagram (that also shows subtype relationships) updated from what you submitted for question 7.
- All the code you wrote.
- A description of your implementation and testing strategy.

Hopefully your simulation will be interesting enough that you will want to share it with your friends and family! You could try and teach them to install and run the Java virtual machine, but they will be much happier if you just point them to a web page. To do this, you need to turn your application into an applet and make a web page that contains it.

Applets

So far, all the Java programs you've worked on have been applications — programs designed to run as standalone applications. It is also possible to create programs that run inside other applications, for example, inside a web browser. Java calls these *applets*. For a quick example, look at the `PhilosopherApplet.java` class to see how we turned our `PhilosopherSimulator` into an applet. A web page that incorporates our applet is http://www.cs.virginia.edu/cs205/ps/ps5/code/philosophers.html. To create an applet, you need to create a subtype of the `java.applet.Applet` type. The `Applet` type is a subtype of java.awt.Panel, which is a subtype of java.awt.Container, which is a subtype of java.awt.Component, which is a subtype of java.lang.Object.

If you look at the API spec for `java.applet.Applet`, you will see that it inherits methods from Panel (1 method), Container (47 methods), Component (56 methods) and Object (10 methods), as well as defining 25 new methods itself. So, there are 139 Applet methods! Fortunately, you can make useful applets only using a few of them directly.

In your `Applet` subtype, you will define replacement methods for some of the Applet methods:

- `public void init()` — this will be called by the browser to initialize the applet.
- `public void start()` — this will be called by the browser when the applet should start executing
- `public void stop()` — this will be called by the browser when the applet should stop executing
- `public void destroy()` — this will be called by the browser when the applet should perform final clean up
- `public void paint(Graphics g)` — called when the applet display should be painted

An applet can be embedded in a web page using `applet`:

```
<html>
<body>
<applet code="ps5.PhilosopherApplet.class" width="520" height="590"></applet>
</body>
```

```
</html>
```

where `ps5.PhilosopherApplet.class` is a class that is a subtype of `Applet` and the classes are found in the ps5/ subdirectory from the web page location.

(We won't cover any more HTML in this class. If you want to incorporate your applets into fancier web pages, a guide to HTML is available here: *http://www.cs.virginia.edu/cs150/guides/html-guide.html*.)

9. Create a web page containing your simulation as an applet.

Turn-in Checklist: You should turn in your answers to all questions on paper at the beginning of class on Wednesday, 25 October. including your code (but not unnecessary printouts of the provided code). Your answers to questions 1-6 should be turned in individually. Each team should turn in a single document that contains your answers to questions 7-9. Also, submit a zip file containing (1) all of your code and (2) the URL for your applet page, by email to *evans@cs.virginia.edu*.