CMSI 186

PROGRAMMING LABORATORY

Spring 2008

Program 2: Discrete Event Simulation

This program introduces the concept of *discrete event simulation*, a "classic" simulation approach that is applicable to a variety of situations.

Program to Write

Write a class called *clock.Clock* that simulates, well, a clock. Clock objects should track *hours*, *minutes*, and *seconds* properties, and it should have a *tick()* method that advances the clock's hour and minute hands by the given *grain*. The class must have a constructor that takes in an initial time and grain.

Once you have written tests for *then* implemented your *clock.Clock* class, write a *clock.Solver* class that answers the question: for what time(s) during a twelve-hour day is the angle between a clock's hour and minute hands some given value *angle*?

Invoking *clock.Solver* should look like this:

java clock.Solver angle grain

…where *angle* is the desired angle and *grain* is the amount by which to move the clock at each *tick()*. The program should then display all of the times for which the angle between a clock's hands is *angle*.

Design Notes

Additional specifications on the parameters given to *clock.Solver*:

- *angle* is required, assumed to be in degrees, must be of type *double*, and must be within the range $0.0 \le angle < 360.0$
- *grain* is optional, but, if supplied, it is assumed to be in seconds, must be of type *double*, and must be within the range $0.0 < grain \le 1800.0$; if not supplied, *grain* should default to 1.0

Gotchas

- What happens when the grain gets very small (e.g., 0.0001)? Try this out and take note of any unusual behavior in your lab reports.
- Remember that a key point of this program is to get firsthand experience on discrete event simulation — so stick to the approach discussed in class, and resist the temptation to find other ways to solve the problem. Sure, there are other ways, but that's not the point of this exercise.

When You're Done with the Clock…

After you finish the clock simulation, and if you'd like to tackle something else, write a class called *tank.Tank* that can be initialized to some maximum *capacity* of water. Tank objects should also have a collection of *tank.Pipe* objects as well as a *clock.Clock* object for keeping time.

Pipe objects either bring water into the tank or take water out. Flow rates for each pipe may vary in a variety of ways; most importantly, they may change over time or based on the amount of water that is currently in the tank.

Once you have written tests for *then* implemented your *tank.Tank* and *tank.Pipe* classes, write a *tank.Solver* class that answers the question: at what time, if any, will the tank be filled to some given capacity for the pipes described below?

Pipe 1 brings water in at the following rates (*gallons per minute* or "gpm"):

- For $0 \le t < 5$ minutes, 10 gpm
- For $5 \le t < 10$ minutes, 15 gpm
- For $t \ge 10$ minutes: 20 gpm

Pipe 2 brings water in at the following rates:

- For $0 \le t < 5$ minutes: amount in tank at time t (i.e., *amount(t)*) ÷ 10 gpm
- For $5 \le t < 12$ minutes *amount(t)* ÷ 8 gpm
- For $t \ge 12$ minutes: alternates every minute between *amount(t)* ÷ 6 and *amount(t)* ÷ 4 gpm

Pipe 3 takes water *out* at the following rates:

- For $0 \le t < 4$ minutes: 4 gpm
- For $4 \le t < 11$ minutes: 9 gpm
- For $t \ge 11$ minutes: alternates *every 90 seconds* between *amount(t)* ÷ 7 and *amount(t)* ÷ 5 gpm

Invoking *tank.Solver capacity* should start the simulation for a tank with the given *capacity* (in gallons) and produce tank status reports (amount in tank, flow rates for each pipe) for every minute of simulated time, stopping when the tank has filled to capacity (which may be never).