Carleton University
Department of Systems and Computer Engineering
SYSC 3303 - Real-Time Concurrent Systems - Summer 2013

Assignment 3

The following concurrent program uses a variation of the `WorkerQueue` class that was discussed in one of the lectures. Reverse engineer the program into a UML class diagram. You don't need to draw the `DisplayQueueTester` class, which just contains the `main()` method that creates the active and passive objects. Draw the other classes, and show the relationships between them (i.e., generalization, association, aggregation, composition). Also produce a UML collaboration diagram for each active object, and at least one UCM of the system.

```
import java.util.*;

/**
 * This class is similar to the WorkQueue class presented in
 * class. The major difference is that the boolean variable used
 * to stop the WorkerThread has been moved from WorkerThread to
 * WorkQueue.
 * When clients no longer need the WorkQueue, they invoke stop()
 * to ask the WorkerThread to terminate gracefully, regardless of
 * how many work items remain in the WorkQueue.
 */
abstract class WorkQueue
{
   private LinkedList queue = new LinkedList();
   private boolean stopRequested = false;

   protected WorkQueue() {
      new WorkerThread().start();
   }

   public final void enqueue(Object workItem) {
      synchronized (queue) {
         queue.add(workItem);
         queue.notify();
      }
   }

   public final void stop() {
      synchronized(queue) {
         stopRequested = true;
         queue.notify();
      }
   }

   protected abstract void processItem(Object workItem);

   private class WorkerThread extends Thread
   {
      public void run() {
         Object workItem = null;
         while (true) {
            synchronized (queue) {
               try {
                  while (queue.isEmpty() && !stopRequested)
```

```
                    queue.wait();
                } catch (InterruptedException e) {
                    return;
                }
                if (stopRequested)
                    return;
                workItem = queue.removeFirst();
            } // end of critical section
            processItem(workItem);
        }
    }
}

/**
 * This subclass of WorkQueue prints each workItem on the system
 * console, printing no more than one item per second, no matter
 * how frequently items are enqueued.
 */
class DisplayQueue extends WorkQueue
{
    protected void processItem(Object workItem) {
        System.out.println(workItem);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}

/**
 * A Client thread creates Integer objects and deposits them in a
 * work queue, to be printed asynchronously.
 */
class Client extends Thread
{
    private DisplayQueue displayQ;
    private int min;
    private int max;

    /**
     * Creates a Client thread that creates Integer objects
     * Integer(min), Integer(min+1), ..., Integer(max) and deposits
     * them in the specified DisplayQueue.
     *
     * @param workQ the DisplayQueue where the Integer objects are
     * enqueued for background printing.
     * @param min the value to be wrapped in the first Integer
     * created by this thread.
     * @param max the value to be wrapped in the last Integer
     * created by this thread.
     */
    public Client(DisplayQueue displayQ, int min, int max) {
        this.displayQ = displayQ;
        this.min = min;
        this.max = max;
    }

    public void run() {
        System.out.println(Thread.currentThread() + " starting");
        for(int i = min; i <= max; i++) {
            displayQ.enqueue(new Integer(i));
```

```
            try {
               sleep(200);
            } catch (InterruptedException e) {}
         }
         // Remove the // from the following statement if you want to
         // test the stop() method.
         // displayQ.stop();
         System.out.println(Thread.currentThread() + " terminating");
      }
   }

   /**
    * Some test code to exercise the DisplayQueue class.
    */
   public class DisplayQueueTester
   {
      public static void main(String[] args) {
         DisplayQueue q = new DisplayQueue();
         Thread client1 = new Client(q, 1, 10);
         Thread client2 = new Client(q, 11, 20);
         Thread client3 = new Client(q, 20, 30);
         client1.start();
         client2.start();
         client3.start();
      }
   }
```

Work Products

Put these in a folder called *part2*.

1. A “README.txt” file explaining the names of your files, etc.
2. Your reverse-engineered design diagrams:
   - the UML class diagram for the system
   - a UML collaboration diagram for each active object
   - a UCM of the system

   Hand-drawn scanned diagrams are acceptable, as long as they are neatly drawn and your handwriting is legible, and the software required to view them is present in the lab.

Notes on UML Class Diagrams

You can exclude the class that contains only the `main()` method when drawing your class diagrams. Draw the other classes, and show the relationships between them; i.e., generalization, association, aggregation, composition. Your class icons should indicate the class' attributes and operations; i.e., they should have an attributes compartment and an operations compartment. Indicate which operations have sequential, guarded or concurrent synchronization semantics by labelling them with the corresponding synchronization property. Feel free to use UML notes to explain any concurrency, mutual exclusion, or condition synchronization aspects of the Java programs that are not adequately modeled by the UML notation for classes and associations. But, remember that a UML diagram is supposed to abstract away from the source code, and not be a literal picture of a program. For example, local variables in methods are not drawn in a UML diagram.

Reminder

The TAs will mark your assignments in the lab. It is your responsibility to ensure that any software required for viewing your files is also present in that lab.

Submitting Assignments

Assignments are to be submitted electronically using the assignment "submit" program. Emailed submissions will not be accepted. See the course outline for the procedure to follow if illness causes you to miss the deadline.

Due: Tuesday, May 28th at 8pm SHARP!