Exercise 5 - Mailboxes, Lab 3 preparation

Washing machine

Your task is to develop the control software for a washing machine. Use mailboxes to send events for thread synchronization and communication.

During the exercise session, the general design of the control software is discussed, including threads and communication mechanisms. *A complete design should be prepared and implemented before the lab session.* You should answer the following questions:

- Which concurrent activities (i.e., threads) do you need? Think about which inputs the system should handle, and how it should handle them.
- Is there any thread that is more time-critical than the rest (and should be given a higher priority)?
- Is it appropriate to introduce more threads to accomplish a clearer and more understandable design?
- Will the CMLA requirements be easy to verify (by quickly inspecting the code) in your implementation?
- Which threads need to communicate? Draw a simple diagram to show your design. What information (which classes) should be communicated?

Specification

The machine is equipped with water level and temperature sensors. In addition, four buttons allow the user to select a washing program (1, 2, 3) or stop the machine (0). Output signals from your software will control the following:

- A valve for letting water into the machine.
- A drain pump for draining the machine.
- A heater for heating the water.
- A motor for the rotation of the barrel. The motor has the modes *left spin*, *right spin*, *fast spin* (centrifuge), and *stop*.

Basic requirements

The machine shall have three washing programs with the following behavior:

Program 1 Color wash: Lock the hatch, let water into the machine, heat to 60°C, keep the temperature for 30 minutes, drain, rinse 5 times 2 minutes in cold water, centrifuge for 5 minutes and unlock the hatch.

Program 2 White wash: Like program 1, but with a 15 minute pre-wash in 40°C. The main wash should be performed in 90°C.

Program 3 Draining: Turn off heating and rotation. After pumping out the water (if any), unlock the hatch. Note: the user should select this program as soon as possible after 0 (stop).

In addition, the machine shall have a special program for immediately stopping it:

Program 0 Stop: All motors, as well as any filling/draining of water, should be turned off immediately.

The main difference between 3 and 0 is that 0 can used as an emergency stop. Program 3 can be used to end a washing after interrupting it (with 0) or after a power failure.

The program selector should require the user to always stop the machine (i.e., press 0) before switching from one program to another.

Additional requirements:

- The temperature must not exceed the ones stated above (60, 40, and 90°C) but may sporadically drop down to 2°C below.
- An electromechanical relay controls the power to the heater. To avoid wearing out the relay, it should not be switched on/off too often.
- While washing, the barrel should alternate between left and right rotation. The direction should be changed once every minute.

Requirements for CMLA certification

To market the washing machine in the EU, the control software must be reviewed and certified by a special committee, CMLA (Comit de Machines Laver Automatiques), in Brussels. The special requirements from CMLA are listed below. Since the review is a costly and time-consuming process, it is very important that the software is written in a way that allows the CMLA requirements to be verified easily. In other words, it should be easy to convince oneself that the requirements are fulfilled by simply inspecting the code.

CMLA states the following requirements, motivated by environmental and safety reasons, for a washing machine to be certified:

1. The drain pump must not be running while the input valve is open.
2. The machine must not be heated while it is free of water.
3. There must not be any water in the machine when the hatch is unlocked.
4. Centrifuging must not be performed when there is any measurable amounts of water in the machine.

Handout code

The hardware for the new washing machine is developed in parallel with the control software. To make it possible to test the control software early (and save some water), a software simulation of a washing machine has been developed. The simulator is available in the *done* folder of the handout code. Your control software should be implemented in the *todo* folder.

Hardware access interface

The abstract class *AbstractWashingMachine* describes the properties of a washing machine and provides operations for manipulating the (possibly simulated) hardware. Your control software will only communicate with the *AbstractWashingMachine* class, not the concrete subclasses implementing the actual hardware interface and the simulation.

```
1  package done;
2
3  /**
4   * Abstract description of a washing machine. Specialized
5   * into simulations and actual hardware implementations.
6   */
7  public abstract class AbstractWashingMachine {
8
9    /**
10    * Read the water temperature.
11    * @return Temperature in Centigrades (0-100).
12    */
13   public abstract double getTemperature();
14
15   /**
16    * Read the water level in the machine.
17    * @return A real number between 0 and 1, where 1 indicates an
18    *   absolutely full (i.e. overflowing) machine and 0
19    *   indicates a (practically) empty one.
20    */
21   public abstract double getWaterLevel();
22
23   /**
24    * Check if the front door is open.
25    * @return True if the door is locked, false if it is open.
26    */
27   public abstract boolean isLocked();
28
29   /**
30    * Turns the heating on/off.
31    * @param on True means "heat on", false means "heat off".
32    */
33   public abstract void setHeating(boolean on);
34
35   /**
36    * Open/close the water input tap.
37    * @param on True means "open tap", false means "close tap".
38    */
39   public abstract void setFill(boolean on);
40
41   /**
42    * Start/stop the water drain pump.
43    * @param on True means "start pump", false means "stop pump".
44    */
45   public abstract void setDrain(boolean on);
46
47   /**
48    * Lock/unlock the front door.
49    * @param on True means "lock door", false means "unlock door".
50    */
51   public abstract void setLock(boolean on);
52
53   /**
54    * Control the turning motor.
55    * @param direction SPIN_OFF, SPIN_LEFT, SPIN_RIGHT, or SPIN_FAST.
56    */
57   public abstract void setSpin(int direction);
```

```
58
59    public abstract void start();
60
61    // ---------------------------------------------------------- PACKAGE PRIVATE METHODS
62
63    /**
64     * Set button listener. The listener's processButton() method will be
65     * called whenever a button is pressed.
66     */
67    abstract void setButtonListener(ButtonListener l);
68
69    // ---------------------------------------------------------- PRIVATE CONSTANTS
70
71    /**
72     * Stop spin.
73     */
74    public static final int SPIN_OFF   = 0;
75
76    /**
77     * Slow left spin.
78     */
79    public static final int SPIN_LEFT  = 1;
80
81    /**
82     * Slow right spin.
83     */
84    public static final int SPIN_RIGHT = 2;
85
86    /**
87     * Fast left spin.
88     */
89    public static final int SPIN_FAST  = 3;
90 };
```

Simulation startup excerpt

The simulator main method is located in the *Wash* class. The simulator basically instantiates a simulation object, registers a ButtonListener that calls the *WashingController* upon keypresses, and starts a simulation thread.

```
1  package done;
2
3  import todo.WashingController;
4
5  public class Wash {
6    public static void main(String[] args)
7      // ...
8      // Create a washing machine simulation and a controller
9      AbstractWashingMachine theMachine
10       = new WashingMachineSimulation(theSpeed, freakShowProbability);
11     // ...
12     theMachine.setButtonListener(new WashingController(theMachine, theSpeed));
13     theMachine.start();
14     // ...
15   }
16 }
```

WashingController

The *WashingController* is implemented by you in the *todo* folder. Note that a reference to the hardware access interface is provided for you in the *WashingController* constructor.

```
1  package todo;
2
3  import se.lth.cs.realtime.*;
4  import se.lth.cs.realtime.event.*;
5  import done.*;
6
7  public class WashingController implements ButtonListener {
8
9    public WashingController(AbstractWashingMachine theMachine, double theSpeed) {
10     // Create and start your controller threads here
11   }
12
13   public void processButton(int theButton) {
14     // Handle button presses (0, 1, 2, or 3). A button press
15     // corresponds to starting a new washing program. What should
16     // happen if there is already a running washing program?
17   }
18 }
```

WashingProgram

Each washing program is implemented as a so-called *transactional* thread, i.e. the thread starts up, accomplishes its task, and ends when done. The *WashingProgram* class acts as a superclass to your washing programs and handles *InterruptedException* so you do not need to. It also stores references to your controllers (temp, water, spin) so you can access them as *myXController* in

your washing program subclasses. The washing program is implemented in the abstract method *wash* called from the *run* method.

```
1  public abstract class WashingProgram extends RTThread {
2
3      // ---------------------------------------------------------- CONSTRUCTOR
4
5      /**
6       * @param   mach              The washing machine to control
7       * @param   speed             Simulation speed
8       * @param   tempController    The TemperatureController to use
9       * @param   waterController   The WaterController to use
10      * @param   spinController    The SpinController to use
11      */
12     protected WashingProgram(AbstractWashingMachine mach,
13                              double speed,
14                              TemperatureController tempController,
15                              WaterController waterController,
16                              SpinController spinController) {
17         super();
18
19         myMachine          = mach;
20         mySpeed            = speed;
21         myTempController   = tempController;
22         myWaterController  = waterController;
23         mySpinController   = spinController;
24     }
25
26     // ------------------------------------------------------ OVERRIDDEN METHODS
27
28     /**
29      * This run() method does two things.
30      * <OL>
31      * <LI>Call the wash() method implemented in the subclass.
32      * <LI>If wash() was interrupted, turn all off machinery.
33      * </OL>
34      */
35     public void run() {
36         boolean wasInterrupted = false;
37         try {
38             wash();
39         }
40         catch (InterruptedException e) {
41             wasInterrupted = true;
42         }
43         catch(RTInterrupted e) {   // Thrown by semaphores
44             wasInterrupted = true;
45         }
46
47         if (wasInterrupted) {
48             System.out.println("Washing program aborted.");
49             myTempController.putEvent(
50                 new TemperatureEvent(this, TemperatureEvent.TEMP_IDLE, 0.0));
51             myWaterController.putEvent(
52                 new WaterEvent(this, WaterEvent.WATER_IDLE, 0.0));
53             mySpinController.putEvent(
54                 new SpinEvent(this, SpinEvent.SPIN_OFF));
```

```
55          }
56      }
57
58      // ---------------------------------------------------------- ABSTRACT METHODS
59
60      /**
61       * Called by run() to perform the actual washing stuff.
62       * Should be implemented by the subclass. Does not need to
63       * catch InterruptedException.
64       */
65      abstract protected void wash() throws InterruptedException;
66
67      // ------------------------------------------------- PROTECTED INSTANCE VARIABLES
68
69      /**
70       * The machine to regulate
71       */
72      protected AbstractWashingMachine myMachine;
73
74      /**
75       * Simulation speed
76       */
77      protected double mySpeed;
78
79      /**
80       * The temperature controller
81       */
82      protected TemperatureController myTempController;
83
84      /**
85       * The water controller
86       */
87      protected WaterController myWaterController;
88
89      /**
90       * The spin controller
91       */
92      protected SpinController mySpinController;
93  }
```

WashingProgram3

The *WashingProgram3* class is an example of a washing program implemented for the washing machine.

```
1  package todo;
2
3  import done.*;
4
5  /**
6   * Program 3 of washing machine. Does the following:
7   * <UL>
8   *   <LI>Switches off heating
9   *   <LI>Switches off spin
10  *   <LI>Pumps out water
```

```
11  *   <LI>Unlocks the hatch.
12  * </UL>
13  */
14 class WashingProgram3 extends WashingProgram {
15
16   // ---------------------------------------------------------- CONSTRUCTOR
17
18   /**
19    * @param   mach              The washing machine to control
20    * @param   speed             Simulation speed
21    * @param   tempController    The TemperatureController to use
22    * @param   waterController   The WaterController to use
23    * @param   spinController    The SpinController to use
24    */
25   public WashingProgram3(AbstractWashingMachine mach,
26                          double speed,
27                          TemperatureController tempController,
28                          WaterController waterController,
29                          SpinController spinController) {
30     super(mach, speed, tempController, waterController, spinController);
31   }
32
33   // ---------------------------------------------------------- PUBLIC METHODS
34
35   /**
36    * This method contains the actual code for the washing program. Executed
37    * when the start() method is called.
38    */
39   protected void wash() throws InterruptedException {
40
41     // Switch of temp regulation
42     myTempController.putEvent(new TemperatureEvent(this,
43                                                    TemperatureEvent.TEMP_IDLE,
44                                                    0.0));
45
46     // Switch off spin
47     mySpinController.putEvent(new SpinEvent(this, SpinEvent.SPIN_OFF));
48
49     // Drain
50     myWaterController.putEvent(new WaterEvent(this,
51                                               WaterEvent.WATER_DRAIN,
52                                               0.0));
53     mailbox.doFetch(); // Wait for Ack
54
55     // Set water regulation to idle => drain pump stops
56     myWaterController.putEvent(new WaterEvent(this,
57                                               WaterEvent.WATER_IDLE,
58                                               0.0));
59
60     // Unlock
61     myMachine.setLock(false);
62   }
63 }
```

The se.lth.cs.realtime package

RTThread excerpt

The *se.lth.cs.realtime.RTThread* class and its subclasses (PeriodicThread, ...) each have their own mailbox, defined in the *mailbox* attribute. Interrupt methods may be needed...

```
1  public abstract class RTThread implements Activity, RTEventListener,
2                  RTEventSource {
3
4          /** The event input buffer. */
5          protected volatile RTEventBuffer mailbox;
6
7
8          /**
9           * Put the time-stamped event in the input buffer of this thread. If the
10          * buffer is full, the caller is blocked.
11          *
12          * @return null, but subclasses are free to return an event if desired to
13          *         inform about blocking or results.
14          */
15         public RTEvent putEvent(RTEvent ev) {
16                 mailbox.doPost(ev);
17                 return null;
18         }
19
20         /**
21          * Interrupts this thread. Thus an interruption-flag is set, which may
22          * raise an InterruptedException in the interrupted thread; otherwise the
23          * flag can be checked via interrupted or isInterrupted.
24          */
25         public void interrupt() { ... }
26
27         /**
28          * Tests if the current thread has been interrupted. The interruption-flag
29          * is reset.
30          *
31          * Note that interrupted is a static method, while
32          * isInterrupted is called on this Thread instance.
33          *
34          * @return true if the current thread has been interrupted;
35          *         false otherwise.
36          */
37         public static boolean interrupted() { ... }
38
39         /**
40          * Tests if this thread is alive. A thread is alive if it has been started
41          * and has not yet died.
42          *
43          * @return true if this thread is alive; false otherwise.
44          */
45         public final boolean isAlive() { ... }
46 }
```

PeriodicThread excerpt

The *PeriodicThread* is used when an activity needs periodic execution. The *perform* method will be called at periodic intervals. The *PeriodicThread* inherits from *RTThread* via *CyclicThread*, and thus have a mailbox.

```
1  public abstract class CyclicThread extends RTThread implements Runnable { ... }
2
3  public class PeriodicThread extends CyclicThread {
4
5    /**
6     * Allocates a new PeriodicThread object.
7     *
8     * @param period   the period of the new thread, in milliseconds.
9     *
10    * @exception IllegalArgumentException if the period is not positive.
11    */
12   public PeriodicThread(long period) { ... }
13
14   /**
15    * To be overridden. The perform method will be called at periodic
16    * intervals.
17    */
18   protected void perform() {}
19 }
```

Mailbox - RTEventBuffer excerpt

Mailboxes are implemented as *RTEventBuffer*:s. Messages are encoded as *RTEvent*:s and posted to other threads mailboxes. A thread can fetch messages posted to its own mailbox.

```
1  public abstract class RTEventBuffer implements RTEventBufferInput,
2                  RTEventBufferOutput, Synchronized {
3
4          /**
5           * Returns the next RTEvent in the queue, blocks if none available.
6           *
7           * @return a non-null event.
8           */
9          public abstract RTEvent doFetch();
10
11         /**
12          * Returns the next available RTEvent in the queue, or null if the queue is
13          * empty. In other words, this method always returns immediately, even if
14          * the queue is empty (non-blocking).
15          *
16          * @return an RTEvent if any available, null otherwise
17          */
18         public abstract RTEvent tryFetch();
19
```

```
20          /**
21           * Adds an RTEvent to the queue, blocks caller if the queue is full.
22           *
23           * @param e    the event to enqueue
24           */
25          public abstract void doPost(RTEvent e);
26 }
```

Message - RTEvent excerpt

Messages are encoded as *RTEvent* objects.

```
1 public class RTEvent extends EventObject {
2
3   /**
4    * Constructs an <code>RTEvent</code> object with the specified
5    * source object and a time stamp from the current system time.
6    *
7    * @param source The object where the event originated.
8    */
9   public RTEvent(Object source) { ... }
10
11
12  /**
13   * Obtain source of the event. This method is inherited from EventObject.
14   *
15   * @return the object which is the source of this event.
16   */
17  public final Object getSource() { ... }
18 }
```

The *source* argument allows the message to remember which thread sent the message. Typical usage: 1. Thread A posts to a thread B and receives a reply (A is the *source* of the message):

```
1 public class A extends RTThread {
2   B b;
3   public A(B b) { this.b = b; }
4   public void run() {
5     b.putEvent(new RTEvent(this));
6     RTEvent ack = this.mailbox.doFetch();
7   }
8 }
```

2. Thread B receives a message and replies (note that B does not need to know about A):

```
1 public class B extends RTThread {
2   public void run() {
3     RTEvent msg = this.mailbox.doFetch();
4     ((RTThread)msg.getSource()).putEvent(new RTEvent(this));
5   }
6 }
```

Miscellaneous

Washing

The simulation is based on the following data:

Volume:	20 l (filled up to 10 l)
Power:	4.2 kW
Water input flow:	0.1 l/s
Water output flow:	0.2 l/s
Rotation speed:	15 rpm (slow), 800 rpm (fast)

A complete wash typically takes 45min - 1h to complete. To facilitate testing the simulation has a speed parameter which can be used to speed up the simulation. Speed=1 corresponds to actual time, and e.g. speed=10 makes the simulation go 10 times as fast. Although the pre- written simulation software takes the speed parameter into account, you still have to use it yourself in some parts of your code. For instance, the washing times (15, 30, and 45 minutes) must be scaled in the control software to get a consistent simulation.

Terminating washing programs from WashingController

When the user presses button 0, the current washing program should be terminated. You can do this by calling the *interrupt()* method on that washing program. This method will, in turn, result in an *InterruptedException* being thrown in your washing program (see below).

Allowing your washing program to be terminated

Your washing program may be terminated at any point. This will result in an *InterruptedException* being thrown from, for example, a *waitEvent* call to the event queue in your washing program. You should not catch the *InterruptedException* in your washing programs. Instead, use the *WashingProgram* class as a superclass and write your washing program in the *wash()* method. That method propagates the *InterruptedException* to the *run()* method of the *WashingProgram* superclass, which handles that exception (by, for example, disconnecting any listeners).

(*InterruptedException* may be thrown in a few other situations, as you may know, such as sleep calls. You should handle these the same way, that is, not at all.)

Additional message classes

Additional subclasses of the *RTEvent* class are available in the *todo* folder, providing messages suitable for controlling the washing machine.

```
1 /**
2  * This event is sent by the temperature and water level control
3  * processes to the washing programs. Indicates that the previous order
4  * has been carried out.
5  */
6 public class AckEvent extends RTEvent {
7   public AckEvent(Object source) { ... }
8 }
```

```
1 /**
2  * This event is sent by washing program processes to the water level
3  * controller process. It is an order to reach and hold a given level.
4  */
5 public class WaterEvent extends RTEvent {
6
7     /**
8      * @param   mode      Regulation mode (WATER_IDLE, WATER_FILL, WATER_DRAIN)
9      * @param   target    Water level to reach and hold
10     */
11    public WaterEvent(Object source, int mode, double level) { ... }
12
13    /**
14     * @return Water regulation mode (WATER_IDLE, WATER_FILL, WATER_DRAIN)
15     */
16    public int getMode() { ... }
17
18    /**
19     * @return Target level
20     */
21    public double getLevel() { ... }
22
23    /** Regulation off, turn off all pumps */
24    public static final int WATER_IDLE  = 0;
25
26    /** Fill water to a given level */
27    public static final int WATER_FILL  = 1;
28
29    /** Drain, leave drain pump running when finished */
30    public static final int WATER_DRAIN = 2;
31 }
```

```
1  /**
2   * This event is sent by washing program processes to the temperature
3   * controller process. It is an order to reach and hold a given
4   * temperature.
5   */
6  public class TemperatureEvent extends RTEvent {
7
8      /**
9       * @param   mode      Temperature regulation mode (TEMP_IDLE, TEMP_SET)
10      * @param   target    Target temperature to reach and hold
11      */
12     public TemperatureEvent(Object source, int mode, double temp) { ... }
13
14     /**
15      * @return Temperature regulation mode
16      */
17     public int getMode() { ... }
18
19     /**
20      * @return Target temperature
21      */
22     public double getTemperature() { ... }
23
24     /** Temperature regulation off (no heating) */
25     public static final int TEMP_IDLE  = 0;
26
27     /** Reach and hold the given temperature */
28     public static final int TEMP_SET   = 1;
29 }
```

```
1  /**
2   * This event is sent by washing program processes to the spin
3   * controller process. It is an order to set a particular spin.
4   */
5  public class SpinEvent extends RTEvent {
6
7      /**
8       * @param   mode   Spin regulation mode (SPIN_OFF, SPIN_SLOW, SPIN_FAST)
9       */
10     public SpinEvent(Object source, int mode) { ... }
11
12     /**
13      * @return Spin regulation mode (SPIN_OFF, SPIN_SLOW, or SPIN_FAST)
14      */
15     public int getMode() { ... }
16
17     public static final int SPIN_OFF  = 0;
18     public static final int SPIN_SLOW = 1;
19     public static final int SPIN_FAST = 2;
20 }
```