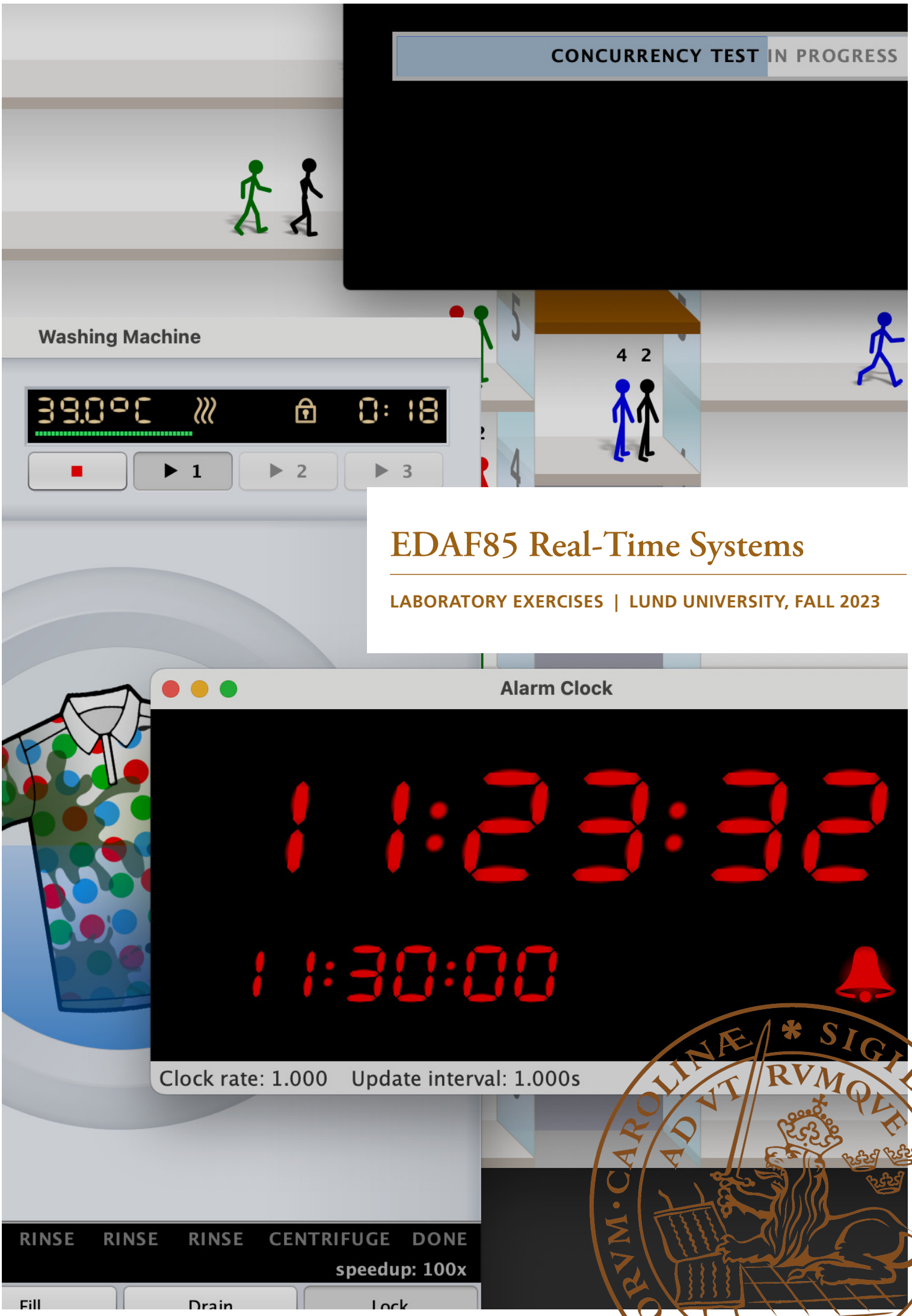


EDAF85 Real-Time Systems

LABORATORY EXERCISES | LUND UNIVERSITY, FALL 2023

Contents

About the labs

Planning your work

Prior to the lab, you will need to come up with a design for your implementation. It is also strongly recommended that you go on to complete most or all of the implementation prior to the lab session. If you come to the lab session without at least a partial implementation, the teacher there may require you to return at a later time.

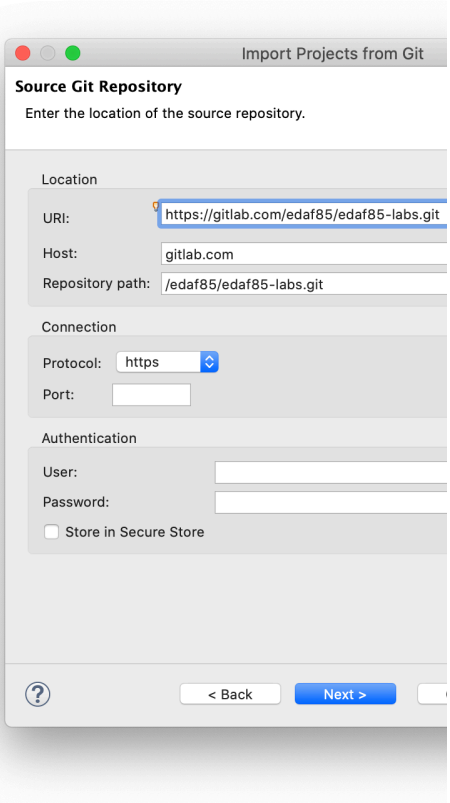
Using your own computer

The labs are based on Java 11 (or later) and JUnit 5. We primarily use the Eclipse development environment.

Download links for Java and Eclipse, in versions that are known to work, are available on the course web. If you use another environment than Eclipse, our teachers and assistants may be unable to help you in some cases.

Getting the lab source code

The source code for the labs includes Eclipse projects. If you choose to use another development environment, make sure you include the course JAR file (`cs/labs.jar`) in your class path.



To bring the projects into Eclipse, clone our Git repository as follows:

1. Start Eclipse with a new, empty workspace.
2. Select *File* → *Import...*
3. In the dialog, select *Git* → *Projects from Git* → *Clone URI*
4. In the dialog "Source Git Repository," in the box "URI," enter the following URL (copy-paste is convenient):

   ```
   https://gitlab.com/edaf85/edaf85-labs.git
   ```

5. Click *Next*
6. Click *Next* (to select the `master` branch)

The git repository is normally placed in directory `git/edaf85-labs` (relative to your home directory), and this is also where we recommend you to place it.

LAB 1

Alarm clock

In the lab you will design the software for a small embedded system—an alarm clock. You will use threads and semaphores. Interfaces for interacting with the hardware are provided.

1.1 Preparation

To prepare for the lab, you will need to sketch a design and consider some of the concurrency and real-time challenges involved.

P1. Read sections 1.1.1–1.1.4 carefully.

P2. Prepare answers to the questions in section 1.1.5 (“Design tasks”). Your will find the answers useful in your implementation work.

Check your design against the checklist in section 1.1.4.

1.1.1 About the target system

You will design and implement the software for an alarm clock, as shown in figure 1.1. The alarm clock has a display, showing the current time and (in smaller digits) the alarm time. A bell icon (bottom right on display) indicates whether the alarm is on. When the alarm rings, the alarm indicator (the bell icon) flashes.

Figure 1.1: Alarm clock with display and six buttons.

Below the display is a panel with six buttons for setting time and alarm, as follows:

- To set the time, press and hold **SET TIME**. Use the left/right **SELECT** buttons to select a digit, and the up/down buttons to change the selected digit. Repeat to change other digits as needed. When **SET TIME** is released, the new time takes effect.
- To set the alarm time, press and hold **SET ALARM**. Use the **SELECT** arrow buttons to set the alarm time. When **SET ALARM** is released, the new alarm time takes effect.
- To disable or enable the alarm, press **SET TIME** and **SET ALARM** simultaneously, then release both. The alarm is then disabled (if it was enabled) or enabled (if it was disabled). The alarm indicator is lit if the alarm is enabled, and off if the alarm is disabled.

The hardware is currently in development, and not yet available. Instead, you will design and implement your software using a software emulator of the alarm clock, shown in figure 1.2. (Such a strategy is commonly used to develop embedded software and hardware at the same time.)

Figure 1.2: Alarm clock emulator.

The emulator mimicks the actual alarm clock closely, and your software can ideally be used unmodified with the actual hardware, once both are available. The emulator provides additional testing functionality, and has as a silent (visual) alarm, without any actual sound. (The real alarm has been carefully designed to wake exhausted users from heavy sleep. This sound would not be conducive to a healthy work environment.)

1.1.2 Hardware input/output

The interface `ClockInput` provides input signals from the clock hardware. The hardware handles the actual time input (selecting and changing digits), and your software will handle the results. In other words, your software will be provided with complete time values (hours, minutes, and seconds, all integers).

Whenever new input is available from the hardware, a semaphore will be `release()`d. Your software can then use the `getUserInput` method to obtain a `UserInput` object, holding the input data. This `UserInput` object, in turn, has a `choice()` method to find out what the user did, and `hours()`/`minutes()`/`seconds()` methods to obtain the time value (if any) entered by the user.

```
public interface ClockInput {

    /** @return  semaphore signaled on user input (via hardware interrupt) */
    Semaphore getSemaphore();

    /** @return  an item of user input (available only when semaphore is signaled) */
    UserInput getUserInput();

    // ---------------------------------------------------------------------

    /** An item of input, entered by the user. */
    interface UserInput {
        /** @return  a value indicating the type of choice made by the user. */
        Choice choice();

        /**
         * These methods return a time set by the user (clock time or alarm time).
         *
         * If choice() returns SET_TIME, these return the time the user entered.
         * If choice() returns SET_ALARM, these return the alarm time the user entered.
         * If choice() returns TOGGLE_ALARM, these return an invalid value.
         */
        int hours();
        int minutes();
        int seconds();
    }
}
```

The `UserInput` method `choice()` returns an `enum` value,[1] indicating the type of entry made by the user:

```
public enum Choice {
    SET_TIME,        // user set new clock time
    SET_ALARM,       // user set new alarm time
    TOGGLE_ALARM;    // user pressed both buttons simultaneously
}
```

[1] For more about `enum`s, see `https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html` (retrieved on March 6, 2023).

Your software is also responsible for updating the current time and alarm status (on/off), as well as signaling the alarm. This is done using the interface `ClockOutput`:

```
public interface ClockOutput {

    /** Display the given time on the display, for example (15, 2, 37) for
        15 hours, 2 minutes and 37 seconds since midnight. */
    void displayTime(int hours, int mins, int secs);

    /** Indicate on the display whether the alarm is on or off. */
    void setAlarmIndicator(boolean on);

    /** Signal the alarm. (In the emulator, only a visual alarm is given.) */
    void alarm();
}
```

As a starting point, you have been provided with a preliminary `main` method (in `ClockMain`), showing how to create an emulator and obtain `ClockInput` and `ClockOutput`. You will obviously need to develop this program further. (In fact, as you will soon see, it crashes immediately when run.)

```
public class ClockMain {
    public static void main(String[] args) throws InterruptedException {
        AlarmClockEmulator emulator = new AlarmClockEmulator();

        ClockInput  in  = emulator.getInput();
        ClockOutput out = emulator.getOutput();

        out.displayTime(15, 2, 37);   // arbitrary time: just an example

        while (true) {
            UserInput userInput = in.getUserInput();
            Choice c = userInput.choice();
            int h = userInput.hours();
            int m = userInput.minutes();
            int s = userInput.seconds();

            System.out.println("choice=" + c + " h=" + h + " m=" + m + " s=" + s);
        }
    }
}
```

1.1.3 Requirements

Your software is subject to the following requirements:

1. The displayed clock time should be updated every second (call `displayTime()` in `ClockOutput`).

   Note: while the user is setting the time, calls to `displayTime()` are ignored, and do not affect the displayed time. (Instead, the display will show the time currently being set by the user.) Nevertheless, your software should call `displayTime()` to update the clock time every second, regardless of mode.

2. It should be possible to set the clock time and the alarm time, as outlined in section 1.1.1.

3. When the clock time is equal to the alarm time, and the alarm is enabled, the alarm should beep once every second for 20 seconds. The `alarm()` method in `ClockOutput` makes **one** such beep. The alarm should stop earlier if **ALARM OFF** (that is, **SET TIME+SET ALARM**) is pressed while the alarm is sounding.

4. Use class `Thread` for concurrent execution and class `Semaphore` for signalling and mutual exclusion. If you wish, you can alternatively use class `ReentrantLock` (interface `Lock`) for mutual exclusion.

5. Keep track of current time using separate variables for hours, minutes, and seconds.

1.1.4 Dos and don'ts

Use the following as a checklist for your design.

Managing time

1. **Don't** use the CPU excessively. Busy-wait is far too inefficient, particularly in a battery-powered device such as an alarmclock. **Don't** use the `Semaphore` method `tryAcquire()` or the `Lock` method `tryLock()`.
2. For similar reasons, polling is not allowed. **Don't** use `Thread.sleep()` except to wait for a second of time (or so) to pass.
3. **Don't** use class `Timer` or thread pools (such as `ScheduledThreadPoolExecutor`) in this lab.

Protecting shared data

4. **Do** introduce a dedicated class to hold shared data. We call this the **Monitor design pattern**.
5. **Don't** make your monitor class a thread. A class could be **either** a monitor **or** a thread. (It could be neither, of course).
6. **Do** use a `Semaphore` or a `Lock` to protect the (shared) monitor data. The `Semaphore`/`Lock` should be declared with the data, in the monitor.
7. **Do** use this `Semaphore`/`Lock` in all public methods of your monitor class.
8. **Do** declare all attributes private, both in your threads and your monitor.
9. **Don't** use array attributes in your monitor. In particular, don't use an array for hours/minutes/seconds, but individual `int` attributes.

   (This is to avoid the temptation to return references to such an array, leading to unsynchronized access to shared data.)

Thread design

10. **Don't** introduce public (or package visible) methods in `Thread` classes. The only public method in a `Thread` class should be the one called `run()`.
11. **Don't** call the `run()` method directly – use `start()`.
12. **Don't** call the `start()` method in the thread constructor.

1.1.5 Design tasks

1. The `main` method (sketched above) handles user input. What additional thread(s) do you need, beyond this `main` thread?
2. What common data needs to be shared between threads? Where is the data to be stored?

   Hint: introduce a dedicated class for this shared data, as outlined above.
3. For each of your threads, consider:
   - What operations on shared data are needed for the thread?
   - Where in the code is this logic best implemented?

4. In which parts of your code is data accessed concurrently from different threads? Where in your code do you need to ensure mutual exclusion?

5. Are there other situations in the alarm clock where semaphores are to be used?

   Hint: have a look at `ClockInput` in section 1.1.2.

1.2 Implementation

I1. Run the program `ClockMain`. It contains the `main` method from section 1.1.2.

When you run it, it crashes immediately, because the `ClockInput` method `getUserInput` is called before any input data is actually available. Read the error message carefully and add some code to make the program block for input. In other words, the ensure that program does not advance to `in.getUserInput()` until the user has entered some input. (You should need to add about one or two lines.)

I2. Now that the program runs properly, acquaint yourself with the emulator. Your computer's keyboard is used for emulating the alarm clock's buttons, using the keys in table 1.1 on the right.

It is not yet possible to set the time or alarm, because you have not implemented that functionality yet.

alarm clock	computer
buttons	keyboard
SELECT	arrow keys (← → ↑ ↓)
SET TIME	shift key
SET ALARM	control or alt key

Table 1.1: Keys used in emulator.

However, you can still try setting the time, setting the alarm, and enabling/disabling the alarm. Examine the printouts from the program. Do they match your expectations?

I3. Implement the classes you designed. Don't expect to get it all working right away—instead, divide your work into steps that (a) bring you closer to the solution, and (b) can be tested.

For example, you could

- start with making the clock tick up every second,
- go on to handle setting the time and alarm, and then
- make the alarm work.

I4. The emulator measures clock drift, by comparing the displayed time to real (actual) time. It also measures the number of clock updates per second. If the timing of your program's clock updates is right, the clock rate should be 1.000, and the clock should be updated once every 1.000 seconds. The status bar at the bottom of figure 1.2 (page 6) shows a correct result.

A transient, small drop in clock rate to, say, 0.999, is fine. Larger, persistent deviations indicate a problem. A clock rate < 1 indicates that the alarm clock is slower than real time.

I5. The emulator also includes a test mode (figure 1.3), designed to detect a particular type of race condition. In this mode, the emulator generates time changes (`UserInput` objects) at a high rate, quickly alternating between 00:00:00 and 12:34:56. The test is started by pressing 'T' and runs for 30 seconds.

If the test passes on your first try, continue to the next step.

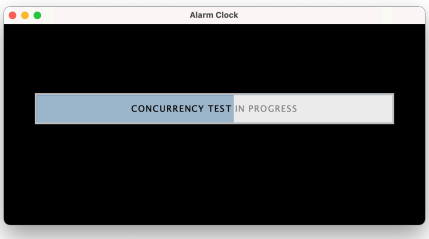


Figure 1.3: Emulator in test mode.

Otherwise examine the error details given in the console window. Make sure you understand how this inconsistency could arise. In particular, try to think of a situation where two or more threads could access your variables for hours, minutes, and seconds in a way that explains your observations.

Once you have understood the problem, rectify it. Verify that your alarm clock now passes the test.

I6. If you identified and fixed an error in step I5, you're done with the implementation work, and should proceed to section 1.3 below (Reflection). If you did **not** encounter an error in step I5, however, you should take this opportunity to observe a race condition in practice.

Make the test fail, by temporarily disabling mutual exclusion in the part of your code where the current time is accessed by multiple threads. (You could, for example, comment a few `acquire`/`release` or `lock`/`unlock` lines out.) Make sure you understand in which situation the inconsistency arises. Then restore your code and verify that your alarm clock, once again, passes the test.

1.3 Reflection

Take some time to reflect on your work in this lab. In particular:

R1. Why is mutual exclusion needed in your program?

R2. How can you use a `Semaphore` (or `Lock`) for mutual exclusion?

R3. How can you use a `Semaphore` for signaling between threads?

R4. How do you use the Monitor design pattern in your design?

R5. What does it mean to say that a thread is blocked?

R6. In your implementation work, tasks I5–I6, you encountered inconsistent output: a clock time value that didn't correspond to the time set. How could this inconsistency arise? How can it be prevented?

R7. The test in step I5 runs for 30 seconds. Why does it have to run for so long? Can we guarantee that this time is sufficient to find the race conditions we are looking for?

If you are unsure, take the opportunity to discuss these questions with your lab teacher when you present your work.

LAB 2

Passenger lift

In this lab, you will design and implement a simulation of a lift in an office building (figure 2.1). In this simulation, passengers arrive from the left, walk to the lift, and wait for the lift to arrive. When the lift arrives, and has enough room for another passenger, the doors open and the passenger enters. When the lift arrives at the passenger's destination floor, the passenger steps off, and walks out to the right.

In your design, you will use a number of threads, and a monitor to synchronize them. You will learn more about monitor design and using monitors for signaling between threads.

In figure 2.1, twelve threads can be seen to be involved: each passenger corresponds to one thread, and the lift corresponds to yet another thread. Digits above passengers indicate their respective destination floors.

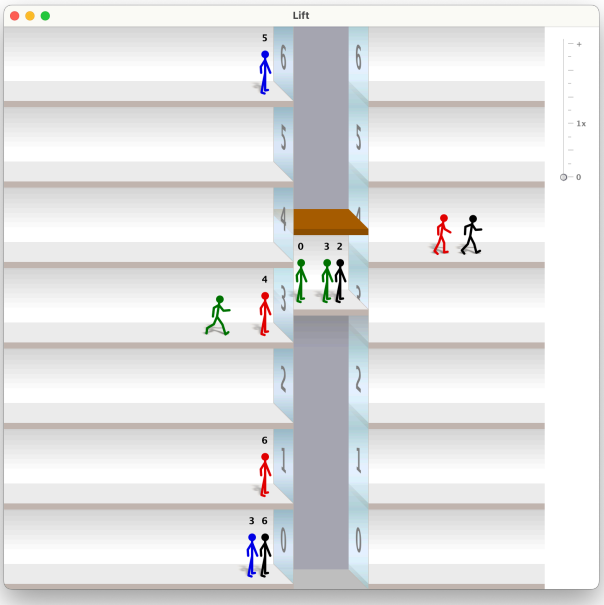


Figure 2.1: Lift simulator.

2.1 Preparation

Your simulation will use one thread to control the lift, and a number of identical threads each simulating a person travelling with the lift. You will use a monitor for synchronization.

P1. Read sections 2.1.1–2.1.3 below.

P2. Design

- a passenger thread class,
- a lift thread class,
- a monitor class, and
- a `main` method.

Your design of the monitor should include its public (monitor) methods. In your design of the threads, also consider which monitor methods a thread calls.

P3. Use the hints in section 2.1.4 as a checklist, to find potential issues in your design.

2.1.1 Visualization of the lift and its passengers

You will use the class `LiftView` for visualization. It has the following public specification:

```
public class LiftView {
    /** Create a new LiftView window, with 'nbrFloors' floors and a lift that can take
        at most 'maxPassengers' passengers. The lift starts at floor 0. */
    public LiftView(int nbrFloors, int maxPassengers)   { /* ... */ }

    /** Move the lift from floor 'here' to floor 'next'. These two must be different. */
    public void moveLift(int here, int next)   { /* ... */ }

    /** Create a new Lift passenger. */
    public Passenger createPassenger()   { /* ... */ }

    /** Open lift doors on the indicated floor.
        All doors must be closed when this method is called. */
    public void openDoors(int floor)   { /* ... */ }

    /** Close lift doors. Doors must be open on exactly one floor when this method is called. */
    public void closeDoors()   { /* ... */ }
}
```

The method `createPassenger` creates a `Passenger`, visualized as a small walking figure (see figure 2.1):

```
public interface Passenger {
    /** @return the floor the passenger starts at */
    int getStartFloor();

    /** @return the floor the passenger is going to */
    int getDestinationFloor();

    /** First, delay for 0..45 seconds. Then animate the passenger's walk, on the entry floor, to the lift. */
    void begin();

    /** Animate the passenger's walk from the entry floor into the lift. */
    void enterLift();

    /** Animate the passenger's walk from the lift to the exit floor. */
    void exitLift();

    /** Animate the passenger's walk, on the exit floor, out of view. */
    void end();
}
```

The following example shows how to create a `LiftView` and a `Passenger`, and visualize their journey.

```
final int NBR_FLOORS = 7, MAX_PASSENGERS = 4;

LiftView  view = new LiftView(NBR_FLOORS, MAX_PASSENGERS);
Passenger pass = view.createPassenger();
int  fromFloor = pass.getStartFloor();
int    toFloor = pass.getDestinationFloor();

pass.begin();                       // walk in (from left)
if (fromFloor != 0) {
    view.moveLift(0, fromFloor);
}
view.openDoors(fromFloor);
pass.enterLift();                   // step inside

view.closeDoors();
view.moveLift(fromFloor, toFloor);  // ride lift
view.openDoors(toFloor);

pass.exitLift();                    // leave lift
pass.end();                         // walk out (to the right)
```

2.1.2 Requirements

Your simulation is subject to the following requirements:

- The passenger thread is responsible for bringing the passenger safely to the destination. The thread then starts over, with a new delay and a new passenger.

  The passenger will usually not appear on screen immediately when your passenger thread calls `Passenger` method `begin()`. This method includes an initial delay in the range 0..45 seconds, to ensure passengers are dispersed in time.

- The lift thread is responsible for moving the lift moving up and down. It follows a simple rule: it moves continuously back and forth beween the bottom and top floors. On each floor, the doors are opened, and passengers are given the opportunity to walk on or off. (You will refine this algorithm later.)

- The lift doors must be opened before any passenger can enter, and must be closed before the lift moves.

- The lift thread and the person threads need to synchronize their actions. For example, a passenger can enter the lift only if it has arrived on the right floor. Once the lift has arrived, the passenger can only `enterLift` if the lift is not already fully occupied (i.e., has four passengers).

  Use Java's monitor facility for synchronization and shared data.

- We will initially assume the lift doors to be very narrow, allowing at most one passenger to enter or exit the lift at a time. (That is, at most one passenger thread may `enterLift` or `exitLift` at a time.)

  This is a temporary assumption, to help you get started. It will be revised for task I5.

2.1.3 Concurrency considerations

Ponder on how the synchronization of the lift and the passenger threads should work:

- What shared data will you need in your monitor?
- Which monitor attributes are accessed/modified by which threads?
- What condition must be true for a passenger to be allowed to enter the lift?
- What condition must be true for a passenger to leave the lift?
- What condition must be true for the lift to start moving to another floor?
- What monitor operations will you need? What is to be done in each monitor operation?
- Where should calls to `wait()` and `notifyAll()` be placed?

Note: the class `LiftView` is thread-safe. However, the method `moveLift` takes long time to execute, and **must not** be executed within a monitor method.

2.1.4 Design tips

- Start out with **long transactions** – that is, put as much of your application logic as possible in the monitor methods. You will have a few comprehensive monitor methods, rather than many small ones.

  Using such long transactions does not imply that the monitor will be held for a long time. (Time-consuming operations should not be executed within the monitor.) Remember that a thread that `wait()`s will temporarily release the monitor while waiting.

- In your monitor, you will need to keep track of how many passengers (if any) need to enter or exit on each floor. Attributes like the following can be useful:

  ```
  private int[] toEnter;     // number of passengers waiting to enter the lift at each floor
  private int[] toExit;      // number of passengers (in lift) waiting to exit at each floor
  ```

  Incidentally, arrays like these also fit nicely with our debugging facilities (section 2.1.5 below).

- Other possibly useful monitor attributes include:

  – the floor the lift is currently on,
  – the number of passengers currently in the lift (0 .. `MAX_PASSENGERS`),
  – whether the lift is currently moving,
  – the current direction of the lift (going up or down),

  These are **suggestions**, not requirements.

- A class can be a thread **or** a monitor. Resist the temptation to make a class both a thread **and** a monitor – that way madness lies.

2.1.5 Debugging support

If your monitor holds incorrect information about the number of passengers waiting to enter and exit on each floor, your simulation will fail in mysterious ways. To help you identify such situations, `LiftView` also includes the following utility method:

```
/**
 * Display the number of passengers waiting to enter and exit on each floor.
 *
 * @param nbrEntry  element [i] indicates the number of passengers
 *                  waiting to enter on floor i
 *
 * @param nbrExit   element [i] indicates the number of passengers
 *                  waiting to exit on floor i
 */
public void showDebugInfo(int[] nbrEntry, int[] nbrExit) { /* ... */ }
```

To debug such issues, call this method whenever your corresponding arrays (e.g., `toEnter`/`toExit`) have been updated. (This means when a passenger has arrived to the lift, entered the lift, or exited the lift.)

2.2 Implementation

I1. The example code from section 2.1.1 is available in the program `OnePersonRidesLift`. Run it. Does it work as you expected? (You can change the simulation rate using the slider on the right.)

I2. Implement your design. Use a lift with a seven-floor building (`NBR_FLOORS=7`), room for four passengers (`MAX_PASSENGERS=4`), and 20 passenger threads. (Later, in optional task X1, you will be encouraged to make your simulation more general.)

As before, break your work down into manageable steps that can be tested individually. If your monitor conditions don't seem to work, you may find the debugging support in section 2.1.5 above useful.

I3. Ensure the lift doors only open if there are any passengers waiting to enter or exit (on that floor).

I4. Ensure the lift halts when no passengers are waiting for the lift to arrive, and none are located within the lift itself. The lift should start again when a passenger arrives.

Note: you can choose yourself whether the lift halts with open or closed doors. The simplest solution is probably to halt with open doors. (This is also convenient to passengers, as they can walk right in.)

I5. Modify your solution to allow multiple passengers to enter and/or exit concurrently. (We no longer assume the doors to be narrow.) This also makes sense from a general design perspective: methods `enterLift` and `exitLift` take some time to execute, and should preferably be executed outside the monitor.

Ensure the lift does not move while passengers are entering or exiting.

Hint: add a monitor attribute for the number of currently entering/exiting passengers. You will likely also need to introduce new, shorter monitor methods, and to move some of your code to those methods.

2.2.1 Suggested enhancements

You have now completed the mandatory part of the lab, but you are encouraged to develop your simulation further. Make sure to first save a copy of the program, so you can show a working solution to the teacher at the lab session. Some possible enhancements include:

X1. Generalize your lift simulation to work with other building heights and lift capacities (other values than `NBR_FLOORS=7` and `MAX_PASSENGERS=4` used in the `OnePersonRidesLift` example).

You will probably see why tall buildings usually have more than one lift. A reasonable building height for our simulation is about 10 or so: in taller buildings, passengers will spend much of their time waiting. (This is a good example of something we can learn from a simulation.)

X2. The lift could provide better service by changing direction before reaching the top or bottom floors in some cases. Specifically, it could change direction if no passengers are waiting to enter or exit on the remaining floors.

X3. A passenger should avoid entering the lift if it is headed in the wrong direction. Passengers will then only occupy the lift when necessary.

2.3 Reflection

R1. What threads are there in your solution, and how do they communicate?

R2. Why do we always put `wait` in a `while` loop? Why wouldn't `if` work?

R3. Mathematician Augustus De Morgan (1806–1871) is known for having formulated two laws, which can be expressed in Java as

```
!(a && b)   ==   (!a || !b)          !(a || b)   ==   (!a && !b)
```

How can these laws be useful when implementing monitor methods? (Think about your `while` loops.)

R4. Why can't we call the `LiftView` method `moveLift()` in a monitor?

R5. Suppose a monitor includes a single attribute `x`. Also suppose the monitor includes the following method, waiting for `x` to change:

```
public class Example {
  private int x;

  // ...

  public synchronized int awaitX() throws InterruptedException {
    while (x == 0) {
      wait();
    }
    return x;
  }

  // ... other methods ...
```

The monitor also has other methods. Some methods (like `awaitX` above) read `x`, and some modify `x`. How can you decide which method(s) should call `notify`/`notifyAll`?

R6. Why can `wait` only be called in a `synchronized` method (or a method called by another `synchronized` method for the same object)?

LAB 3

Washing machine

In this lab, you will design and implement control software for a washing machine. Our simulator (figure 3.1) simulates some of the physical processes involved. You will implement two kinds of control:

regulation of water temperature, water level, and barrel spin, and

sequential control in washing programs.

You will use **message passing** for communication and synchronization. You will also use Java's **interruption** facility to stop running threads in a controlled manner.

Figure 3.1: Washing machine simulation.

3.1 Preparation

P1. Read the Java documentation about

`BlockingQueue`
`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html`

interruption
`https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html`

P2. Read section 3.1.1 and complete the class `ActorThread`. Ensure the JUnit test passes.

P3. Read section 3.1.2. Start the program `Wash`. A washing machine simulator appears, with a display showing water temperature, elapsed time, and room for a number of indicators. The indicators are initially off, but will light up when a corresponding control signal is activated (figure 3.2).

Press **M** to bring up manual controls for the hardware. (You will also see an additional display for state information, which may become useful later.) Get acquainted with the washing machine simulator by operating it manually, for example as follows:

- start program 1 (to get some laundry to work with),
- lock the hatch,
- fill water into the barrel (about half-way full), and
- heat the water up to about 39 degrees.
- Now try to keep the temperature between 38 and 40 degrees, while changing barrel rotation from left to right continually. Tricky, isn't it?

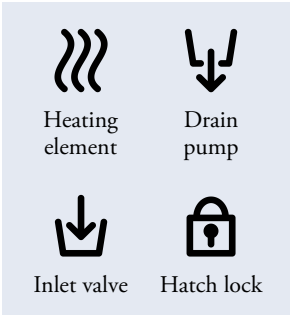


Figure 3.2: Display indicators.

P4. One washing program (`WashingProgram3`) is already implemented for you. Open `WashingProgram3` to get a general idea of how it works, and how it communicates with `SpinController`, `TemperatureController`, and `WaterController` (though these are not implemented yet).

P5. As you may have noted, you will use the class `WashingMessage` for communication between your threads. Open this class, examine it, and read section 3.1.3 to learn more about it.

P6. Read section 3.1.4. Choose $dt = 10s$ for your temperature regulator. What margins (m_u and m_l) will you need? Add an extra safety margin (say, 0.2°C) to accommodate sensor noise.

Make a note of these values — you will need them in your implementation.

P7. Read section 3.1.5–3.1.6 to learn more about how the washing machine is intended to work, and how the simulator can be used to verify some of these requirements.

3.1.1 Implementing message passing (class `ActorThread`)

In contrast to monitors, we're not relying on any built-in Java feature for message passing. Instead, you will implement a utility class `ActorThread`, which allows threads to send and receive messages easily.[2] Each such `ActorThread` thread will have its own **queue of incoming messages**.

Say, for example, that we use `Strings` for messages. `ActorThread` A could then send a message to `ActorThread` B as follows:

A	B
`ActorThread<String> b = ...` `b.send("hello");`	`String s = receive();` `System.out.println("got message: " + s);`

The method `send` puts a message in the receiving thread's message queue. The method `receive` fetches a message from a thread's own message queue. If the message queue is empty, `receive` will block until a message is available. There is also a method `receiveWithTimeout`, which only blocks for a limited time, and instead returns `null` if no message was received after the specified time.

The `ActorThread` class looks as follows:

```
public class ActorThread<M> extends Thread {

    /** Called by another thread, to send a message to this thread. */
    public void send(M message) { /* ... */ }

    /** Returns the first message in the queue, or blocks if none available. */
    protected M receive() throws InterruptedException { /* ... */ }

    /** Returns the first message in the queue, or blocks up to 'timeout'
        milliseconds if none available. Returns null if no message is obtained
        within 'timeout' milliseconds. */
    protected M receiveWithTimeout(long timeout) throws InterruptedException  { /* ... */ }
}
```

In the package `actor.test` you can find a few examples, as well as a JUnit test that runs the examples and checks for correct output. The examples don't work yet, and so the test fails.

[2] Here, the term "actor" refers to the *actor model*: a way of structuring concurrent programs, where actors communicate only by sending/receiving messages. Read more at, for example, `https://en.wikipedia.org/wiki/Actor_model`.

Your task

Complete the class `ActorThread`. In the provided project you will find the incomplete source code for `ActorThread` (in the package `actor`). Use a `LinkedBlockingQueue` for handling the message queue. You also need to implement the methods.

How to tell if your `ActorThread` works

If the JUnit test passes (all green), your `ActorThread` implementation is most likely fine. It is possible to implement `ActorThread` correctly by adding one line for an attribute, and one line of code for each of the methods. (Some alternative solutions may involve a few more lines.)

3.1.2 Hardware input/output

We will now turn to the washing machine hardware. It includes a number of input and output signals:

Inputs: sensor signals for **water level** and **temperature**, as well as **program buttons** (for the user to select washing programs).

Outputs: control signals for the **heating element**, the **inlet valve** (for letting water into the machine), the **drain pump**, and the **hatch lock**. There is also a signal for the motor (barrel rotation): **left**, **right**, **fast** (centrifuge), or **idle** (stopped).

These inputs and outputs are accessible to your software in the interface `WashingIO`:[3]

```
public interface WashingIO {

    /** @return water level, in range 0..20 liters */
    double getWaterLevel();

    /** @return temperature, in degrees Celsius */
    double getTemperature();

    /** Blocks until a program button (0, 1, 2, 3) is pressed */
    int awaitButton() throws InterruptedException;

    /** Turn heating element on (true) or off (false) */
    void heat(boolean on);

    /** Set inlet valve to open (true) or closed (false) */
    void fill(boolean on);

    /** Turn drain pump on (true) or off (false) */
    void drain(boolean on);

    /** Set hatch to locked (true) or unlocked (false) */
    void lock(boolean locked);

    /** @param mode  one of Spin.IDLE, Spin.LEFT, Spin.RIGHT, Spin.FAST */
    void setSpinMode(Spin mode);

    /** Values for setSpinMode */
    enum Spin { IDLE, LEFT, RIGHT, FAST };
}
```

[3] IO or I/O is sometimes used as an abbreviation for "input/output signals".

3.1.3 Messaging using `WashingMessage`

`WashingMessage` includes a number of constants for orders. You saw some of these in `WashingProgram3`, such as `SPIN_OFF` and `WATER_DRAIN`.

Make a list of these orders. For each order, note:

a) Which thread should send it?

b) Which thread should receive it?

c) Does the order need an acknowledgement?

For example, `WashingProgram3` awaits acknowledgment from `WaterController`, to make sure the barrel is empty of water before the hatch is unlocked.

Can you think of other similar situations, where an acknowledgment is appropriate?

Contents of `WashingMessage`

Messages can have attributes, and this makes message passing very useful. A `WashingMessage` contains two attributes (with associated getter methods):

- `command` indicates the intention of the message, such as `WATER_DRAIN` or `TEMP_SET`.
- `sender` makes `WashingMessages` keep track of the sender (the thread that originally sent the message). A thread can reply with an `ACKNOWLEDGMENT` message using something like

```
WashingMessage m = receiveWithTimeout(...);
// ... do stuff with m ...
WashingMessage ack = new WashingMessage(this, WashingMessage.Order.ACKNOWLEDGMENT);
ActorThread<WashingMessage> s = m.sender();
s.send(ack);
```

3.1.4 Washing machine physics

The washing machine simulator includes a simulation of physical processes (heating, cooling, water flow), based on the data in table 3.1.

Barrel volume:	20 L (normally filled up to about 10 L)
Heating power:	2 kW
Water input flow:	0.1 L/s
Water output flow:	0.2 L/s
Rotation speed:	15 rpm (slow), 800 rpm (fast)

Table 3.1: Physical data for washing machine.

Temperature control

You will implement a temperature controller, responsible for maintaining a given temperature (by turning the heater on/off). This means keeping the temperature between a **lower bound** and an **upper bound**.

When controlling the water temperature towards N degrees, N is the upper bound, and $N-2$ is the lower bound. For example, when washing in 60°C, the temperature T must be in the range $58 \leq T < 60$. Any temperature $T \geq 60$°C may otherwise damage the clothes.

Heating and cooling

The temperature controller will run periodically, say, every dt seconds. When switching the heater on, we must ensure the temperature will not exceed the upper bound until the next execution of the controller – that is, within the next dt seconds. We cannot, for example, switch the heating on at 59.99999°C: in the

next dt seconds, the temperature will then likely exceed 60°C. We can only heat until the temperature reaches $60 - m_u$, for some upper margin m_u (figure 3.3a).

This upper margin m_u must be larger than the temperature increase over dt seconds, if the heater is on:

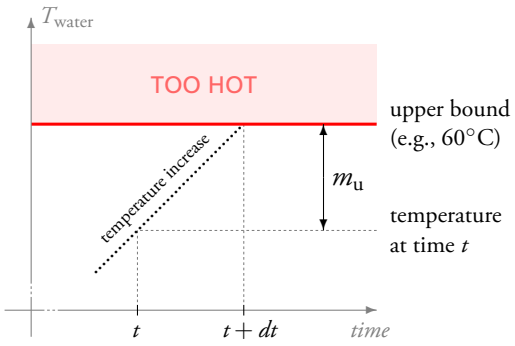
$$m_u > \frac{dt \cdot P}{V \cdot C} \approx dt \cdot 0.0478$$

where $P = 2\,\text{kW}$, $V = 10\,L$, and $C = 4.184 \cdot 10^3$ J/K (heat capacity of 1 litre of water).

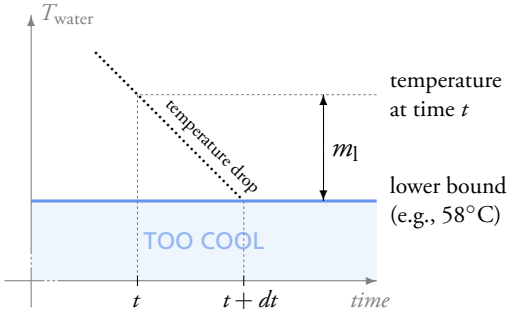
Similarly, we need to know a lower margin m_l, to ensure the temperature doesn't drop below the lower bound within the next dt seconds (figure 3.3b). The cooling depends on the machine's thermal isolation, and is proportional to the temperature difference from the environment:

$$m_l > dt \cdot k_{cool} \cdot (T - T_{amb}) \approx dt \cdot 9.52 \cdot 10^{-3}$$

where $k_{cool} \approx 2.38 \cdot 10^{-4}$ (measured experimentally), $T_{amb} = 20°\text{C}$ (ambient temperature), and $T = 60°\text{C}$ (worst case).



(a) heating for dt seconds



(b) cooling for dt seconds

Figure 3.3: Margins for keeping temperature between the lower and upper bound.

3.1.5 Requirements

The washing machine is subject to a number of requirements. These requirements are of two kinds: functional requirements (3.1.5.1) and safety requirements (3.1.5.2).

3.1.5.1 Functional requirements

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

Program 1 (color wash): Lock the hatch, let water into the machine, heat to 40°C, keep the temperature for 30 minutes, drain, rinse 5 times 2 minutes in cold water, centrifuge for 5 minutes and unlock the hatch.

While **washing** and **rinsing** the barrel should spin slowly, switching between left and right direction every minute. While **centrifuging**, the drain pump should run to evacuate excess water.

Program 2 (white wash): Like program 1, but with a 20 minute pre-wash in 40°C. The main wash (30 minutes) is to be performed in 60°C. Between the pre-wash and the main wash, the water in the barrel is drained and replaced with new, clean water.

Program 3 (draining): Turn off heating and rotation, drain the barrel of water, and unlock the hatch.

Note: the user is expected to select this program as soon as possible after interrupting an ongoing washing program with program 0 (stop).

Moreover, the machine should have a special program for immediately stopping it:

Program 0 (STOP): All motors, as well as the pump and the valve, should be turned off immediately.

The key difference between 3 and 0 is that 0 can used as an emergency stop. Program 3 can be used to finish a washing after interrupting it (with 0) or after a power failure. The mechanical design of the program selector ensures that the machine is always stopped (i.e., button 0 is pressed) before another program is selected.

Programs 1–2 are subject to the following additional requirements, which are verified by the simulator:

- The temperature must not exceed the ones stated above (40°C and 60°C) but may sporadically drop down to 2°C below. See also "Temperature control" in section 3.1.4.
- An electromechanical relay controls the power to the heater. To avoid wearing out the relay, it should not be switched on/off too often. When keeping the temperature at 40°C, the relay must not be activated more often than once every 200 seconds. At 60°C, it must not be activated more often than once every 100 seconds.
- While washing, the barrel should alternate between left and right rotation. The direction should be changed once every minute.

3.1.5.2 Safety requirements

The machine must not harm its users or the environment: it must not cause fire or overflow, and it must not consume unnecessary amounts of water. Moreover, the machine must not be physically damaged.

For these reasons, your software is subject to the following safety requirements:

SR1. The machine must only be heated if there is water in it.
SR2. The drain pump must not be running while the input valve is open.
SR3. There must not be any water in the machine when the hatch is unlocked.
SR4. The barrel must not spin when the hatch is unlocked.
SR5. Centrifuging must not be performed when there is any measurable amount of water in the machine.

These requirements are to be verified not only by simulation, but also by code review. In other words, you should be prepared to convince your lab teacher that these requirements are fulfilled by briefly showing a limited part of your code. The easier it is to verify the safety requirements, the better.

3.1.6 State supervision in the simulator

To assist you in your work, the simulator automatically supervises the state of the machine and your control software. It does this by monitoring your program's actions against the requirements, and it attempts to determine how far your washing program has progressed. The state supervision is there to guide you towards a complete solution, and it can provide (some) immediate feedback on your work.

For example, when program 1 is selected, the simulator's state supervision will expect your application to (as stated in section 3.1.5.1):

- fill the machine with water (to about half),
- raise temperature to between 38 and 40°C, and
- rotate the barrel slowly left and right.

Once these conditions have been met, they are expected to be kept for 30 minutes (again, as stated in section 3.1.5.1). If they are not, the simulator will inform you. After 30 minutes, your program is expected to continue with rinsing.

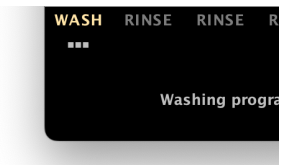
The results of the state supervision are given in the same panel as the manual controls (obtained by pressing **M**). Figures 3.4a–3.4c illustrate the progress of the first step (WASH) of program 1. In figure 3.4a, the WASH step is expected to begin, but hasn't done so yet. (The conditions above are not yet fulfilled.) In figure 3.4b, the WASH step has been active for about 17 minutes. In figure 3.4c, the WASH step has completed, and the next step (RINSE) is now expected to begin.

If your program doesn't seem to progress (according to the state supervision), and you wonder why that is, find out more by pressing the **S** key. Information about the current step (as estimated by state supervision) will then be printed to your Console window, including any conditions required for the step to begin or complete. For example, if you select program 1, then press **S** immediately (before the washing program has taken any action), something like this will be printed:

```
[0:00:02.80]: step WASH (40°C, 30 minutes) is WAITING because:
  machine must be filled to about half with water
  water too cool: temperature must be >= 38°C
  barrel must spin slowly
  barrel must change direction every minute
```

Figures 3.4b–3.4c also illustrate the simulator's monitoring of the heating relay. As explained in section 3.1.5.1, this relay must not be switched on/off too often. In figure 3.4b, the average time between relay activations in the WASH step is measured to 352 seconds (well above the lower bound of 200 seconds at 40°C). The relay is only monitored in this way when heating is involved — in WASH, but not in RINSE, for example.

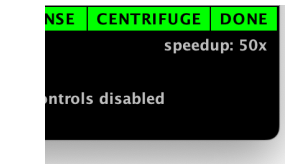
In figure 3.4d, the green DONE indicator corresponds to a successfully completed washing program. The laundry should now be sparkling clean.



(a) Expecting WASH

(b) During WASH

(c) Expecting RINSE



(d) Program completed

Figure 3.4: State supervision in the simulator.

3.2 Implementation

I1. Modify the program `Wash`, so `WashingProgram3` is started when button 3 is pressed.

Verify that button 3 now starts `WashingProgram3`. It will not reach the end, however: the final `println()` message is not printed. (Although the state supervision indicates the program to be complete, the washing program will not reach the end of the `run()` method.)

Why not? (You won't need to fix the problem yet – we will come to that below.)

I2. Modify `Wash` further so the current washing program (if any) is terminated when the STOP button (0) is pressed. Once the current program has been terminated, it should be possible to start a new one.

The button panel automatically requires STOP to be pressed before a new washing program is started, and your modifications will now ensure that at most one program can be running at a time.

Hint: by inspecting `WashingProgram3`, you can get a clue as to how washing programs can be terminated in a controlled manner.

I3. Make a copy of `WashingProgram3`, and call it `WashingProgram1`. Let the body of `WashingProgram1` (the section between `try` and `catch`) initially be just the following:

```
// Lock the hatch
io.lock(true);

// Instruct SpinController to rotate barrel slowly, back and forth
// Expect an acknowledgment in response.
System.out.println("setting SPIN_SLOW...");
spin.send(new WashingMessage(this, SPIN_SLOW));
WashingMessage ack1 = receive();
System.out.println("washing program 1 got " + ack1);

// Spin for five simulated minutes (one minute == 60000 milliseconds)
Thread.sleep(5 * 60000 / Settings.SPEEDUP);

// Instruct SpinController to stop spin barrel spin.
// Expect an acknowledgment in response.
System.out.println("setting SPIN_OFF...");
spin.send(new WashingMessage(this, SPIN_OFF));
WashingMessage ack2 = receive();
System.out.println("washing program 1 got " + ack2);

// Now that the barrel has stopped, it is safe to open the hatch.
io.lock(false);
```

Modify `Wash` so `WashingProgram1` can be started with button 1. Although `SpinController` is not yet fully implemented, you should be able to see that it receives the messages. (How?)

I4. Implement `SpinController`.

- When `SpinController` receives a `SPIN_SLOW` message, the barrel should alternate between slow left rotation and slow right rotation, changing direction every minute.
- When `SpinController` receives a `SPIN_FAST` message, the barrel should rotate fast (centrifuge).
- When `SpinController` receives a `SPIN_OFF` message, barrel rotation should stop.
- In all three cases above, an acknowledgment should be sent back (a `WashingMessage` with command `ACKNOWLEDGMENT`).

Run `WashingProgram1` (as above) and verify that the barrel indeed rotates back and forth for five simulated minutes. (Note that at `SPEEDUP=50`, five minutes are simulated in six seconds.)

Also verify that the washing program receives an acknowledgment from `SpinController`.

I5. Implement `TemperatureController` and `WaterController`. Refer to section 3.1.4 and the values for m_u and m_l you calculated earlier.

Don't forget the extra safety margin for temperature (0.2°, as suggested in preparation item P6). This margin should be added to both m_u and m_l.

These controllers are **periodic threads**: that is, they are to wake up periodically, examine the system state (temperature/water level), and possibly take action. Any received message should be handled immediately, however, just like in `SpinController`.

Hints:

- During preparation (item P6), we decided on a period of $dt = 10\,s$ for the temperature controller. Choose a shorter period for the water controller. (Water fills/drains fast.)

- While implementing these controllers, use one of the washing programs (such as `WashingProgram1`) as a test bed, just as you did in steps I3–I4 above.
- When the temperature T is in range ($38 \leq T < 40$ or $58 \leq T < 60$, depending on program and state), the temperature bar turns green. When the temperature is too high, the bar turns red.
- If the heating relay is activated too often, it will be damaged. The simulator will give an error in this case. Use sections 3.1.4–3.1.5 to ensure that your algorithm starts and stops heating properly. Remember to keep the period for the temperature controller at $dt = 10s$.

I6. Once you are happy with your implementation of the controllers, complete the implementation of `WashingProgram1` and `WashingProgram2`.

Use the state supervision to verify that your washing machine software works according to the requirements in section 3.1.5. (Note, however, that some of the requirements are to be verified by code review.)

You can assume that users will **not** access the hardware controls (the 'H' key) while a program is running.

3.3 Reflection

R1. Which threads exist in your solution?

R2. How do the threads communicate? Is there any shared data?

R3. `WashingMessages` are *immutable*. What does this mean? Can you think of how it could matter here?

R4. For `TemperatureController`, we selected a period of 10 seconds.
What could the downside of a too long or too short period be?

R5. What period did you select for `WaterController`?
What could the downside of a too long or too short period be?

R6. Do you use any `BlockingQueue` in your solution? How?

R7. How do you use Java's interruption facility (`interrupt()`, `InterruptedException`)?

R8. How do you ensure that the machine never heats unless there is water in it?

R9. Suppose a washing program ends by turning the heat off and draining the machine of water. The heat is turned off by sending a `WashingMessage` to `TemperatureController`.

How can you ensure that the heat has indeed been turned off before the washing program continues (and starts the drain pump)?

APPENDIX A
Troubleshooting graphics issues

The labs in this course are somewhat graphically intensive. They work best with hardware-accelerated graphics, and so hardware acceleration is enabled automatically in our lab code.

If you run the labs on your own computer, and the graphics look strange (such as distortions, unexpectedly blank windows, or anomalous flickering), you may find these tips helpful. Start with section 1 below.

1. Finding out whether your computer has hardware-accelerated Java graphics

Different hardware acceleration technologies are used for Windows, Mac, and Linux:

Windows and Macs have their own graphics acceleration, enabled automatically. We haven't seen any issues with either. In case of inadequate graphics performance, try reducing the graphics frame rate (section 3).

On **Linux**, we use OpenGL acceleration.[4] When you start one of our labs on an OpenGL-capable Linux machine, you should see the following message printed in the Console window:

```
OpenGL pipeline enabled for default config on screen 0
```

If you use Linux and **don't** see this message, it probably means that your computer does not have OpenGL support. Getting OpenGL to work just for our labs is probably not worth the hassle: our labs will likely run adequately anyway. If they don't, see section 3 below about reducing the frame rate.

2. Disabling OpenGL acceleration (Linux)

If the graphics on your Linux machine look distorted, try disabling OpenGL acceleration, by putting this Java line **first in your `main` method**:

```
System.setProperty("sun.java2d.opengl", "False");               // disable OpenGL acceleration
```

This will likely result in choppier graphics. However, on many computers (such as the LTH Linux machines) the labs work fine anyway, albeit somewhat less aesthetically pleasing.

If you find the graphics *too* choppy, you can try using a lower frame rate – see below.

3. Reducing the frame rate (Windows, Mac, or Linux)

In our labs, the frame rate will be automatically reduced if the graphics lag behind. However, the reduction is not immediate, as the rate control relies on measured graphics performance (over a few seconds).

To select a lower frame rate from the start, put this Java line **first in your `main` method** (or possibly after any other `System.setProperty` calls):

```
System.setProperty("se.lth.cs.lower_frame_rate", "True");       // request lower frame rate
```

You can also resize the window to a smaller size, as a window with fewer pixels can be updated faster.

[4] `https://docs.oracle.com/javase/8/docs/technotes/guides/2d/new_features.html`. The property `sun.java2d.opengl` is set automatically in our labs; you don't need to set it yourself unless you want to disable acceleration (section 2).

LUND UNIVERSITY

LTH

FACULTY OF ENGINEERING

cs.lth.se/edaf85

LUND UNIVERSITY
Department of Computer Science

Box 118
S-221 00 Lund
cs.lth.se