# Exercise 1: Concurrent Programming

Dominique Blouin, Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

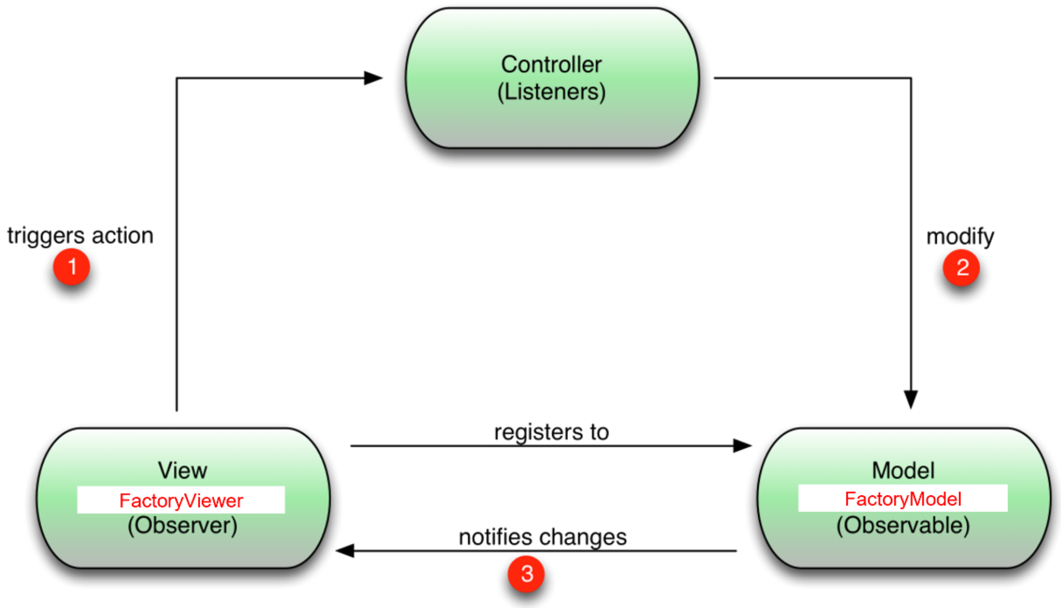
This first exercise consists of modifying the given Robotic Factory Simulator (RFS) that works sequentially so that each of its component (the robots for instance) can run in parallel. For this you will modify the code to execute the behavior of each component in a single thread as seen in the course. You will then need to synchronize the access to positions of the factory by the moving components to ensure that they can be accessed by only one component at a time.

## Introducing the Robotic Factory Simulator

In this section, we briefly introduce the RFS to explain its overall architecture, the detailed behavior of the robots, and how to run the simulator.

### Overall Architecture

The following diagram depicts the overall architecture of the simulator. It consists of three main components implementing the Model-View-Controller (MVC) design pattern.

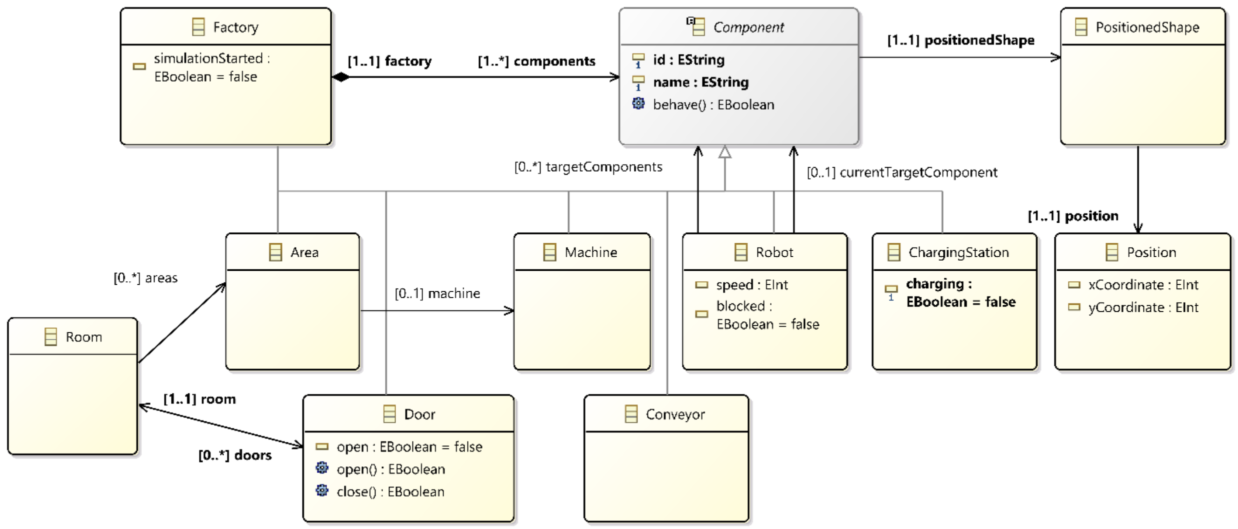


The model consists of all classes modeling the factory data that can be found in package *fr.tp.inf112.projects.robotsim.model*. The view consists of a canvas viewer Graphical User Interface (GUI) that is used by the simulator as a library. The viewer provides a set of Java interfaces specifying figures to be drawn in a canvas. The components of the factory were made to implement those interfaces so that the factory components can be view as figures and the factory as a canvas representing the factory floor. Finally, the controller class (named *SimulatorController*) can be found in package *fr.tp.inf112.projects.robotsim.app*.

As illustrated in the figure, the view registers itself to the model so that it can be notified when the model data is changed. In such case, the view refreshes itself so that it displayed the updated data. The controller listens to actions requested by the view such as starting r stopping the simulation. It modifies the model according to the received event.

## Factory Data Model

The following class diagram depicts the model of the factory.



An abstract class named *Component* is provided as the parent class of all different specific components of the factory such as *Room*, *Area*, *Machine*, *Door*, *Conveyor*, *Robot* and *ChargingStation*. The Factory itself is also considered as a component and therefore, its class also extends the *Component* class.

The *Factory* class contains all components of the factory via the *components* reference attribute. Other classes have references to components inside them such as *Room* that has *Doors* and *Areas*. An *Area* itself has a reference to a production *Machine*.

## Robot Behavior

The *Component* class declares a method named *behave()*, which returns a boolean stating whether the component did something or not. While the method does nothing and simply returns *false* for the abstract component, it can be redefined for coding the behavior of specific components like that of the *Robot* class.

The *Robot* class declares an attribute named *targetComponents* that contains the components to visit by the robot in the factory. It also declares an attribute named *currentTargetComponent* storing the current target component towards which the robot should move.

The class redefines the *behave()* method whose algorithm is given the following:

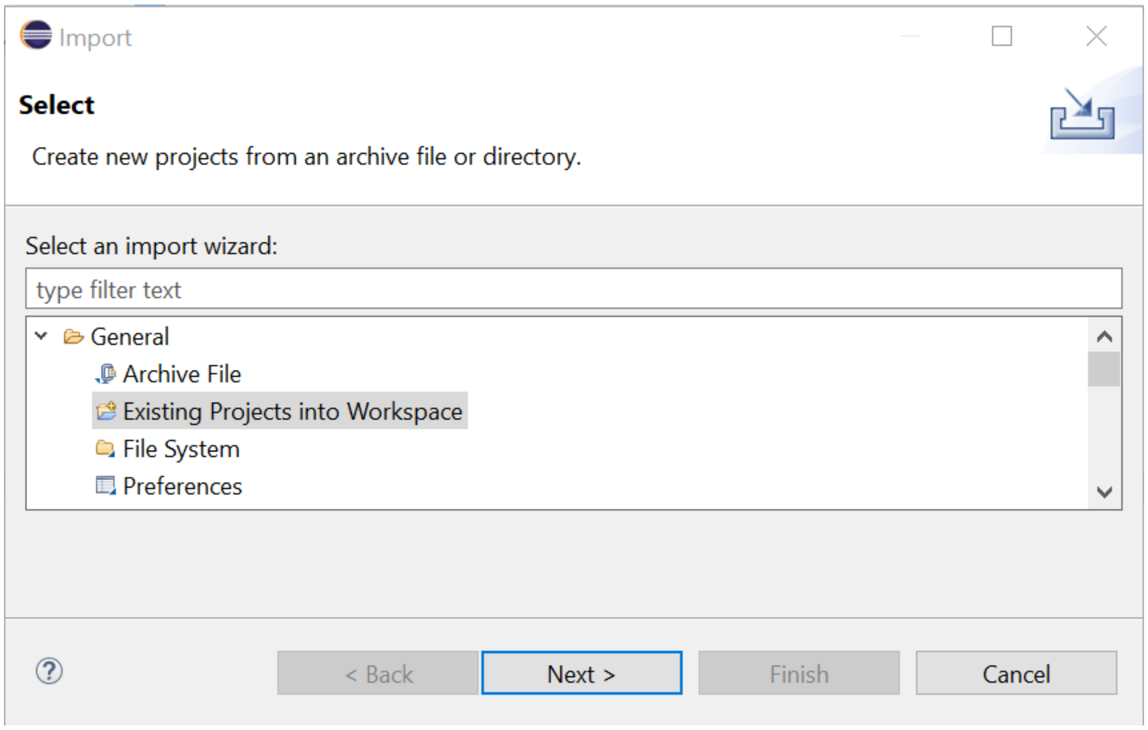
```
if targetComponents.isEmpty()
    return false
end if
```

```
if currTargetComponent = null or hasReachedCurrentTarget()
    currentTargetComponent = nextTargetComponentToVisit()

    computePathToCurrentTargetComponent()
end if

moveToNextPathPosition()
```

This algorithm first checks if the list of target components is empty and if so, *false* is immediately returned. Otherwise, if the current target component is null (it has not yet been initialized) or if the robot has reached the current target component, the next component to visit is obtained from the list of components to visit. Then, a list of positions in the factory constituting a path to reach the new target component is computed. Finally, the robot moves to the next position in the path.

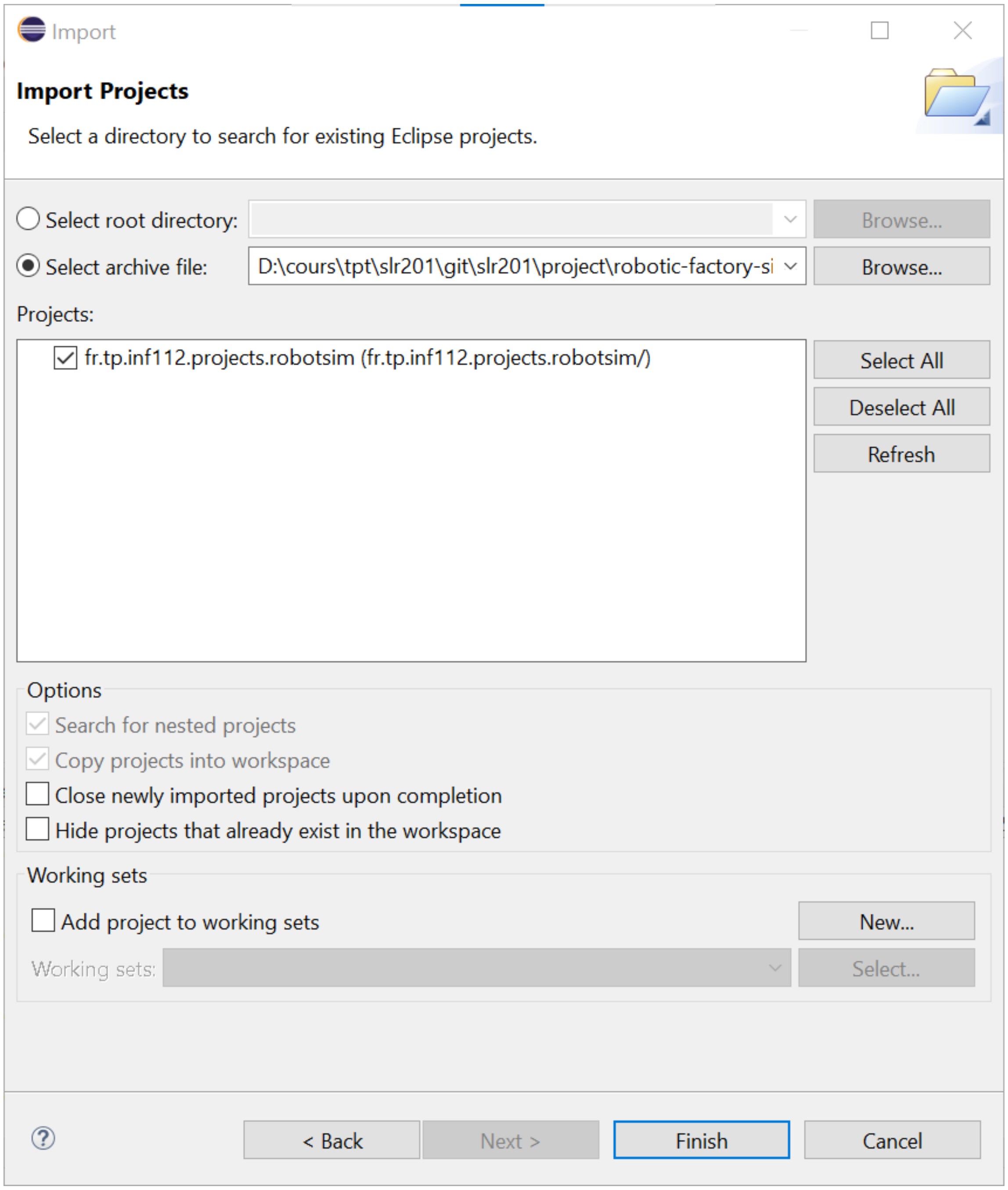
## Installing and Running the Robotic Factory Simulator

To install the simulator, first download its code from the eCampus course page. Then import the project into Eclipse by clicking menu *File>>Import*. From the dialog box that pops up, unfold the *General* category and select the *Existing Projects into Workspace* branch. Click *Next*.



In the dialog box that is displayed, check the *Select archive file* radio button and browse to the downloaded robotic factory simulator code archive. Check the *robotism* project in the project list and click *Finish*. The project should now be visible in the workspace.

The source code is located a project subfolder named *src*. The Javadoc of this code can be found under folder *doc*. Double-click the *index.html* file to navigate throughout the classes.


Import

**Import Projects**

Select a directory to search for existing Eclipse projects.

Select root directory:

[✓ Select archive file:] D:\cours\tpt\slr201\git\slr201\project\robotic-factory-si

Projects:

- [✓] fr.tp.inf112.projects.robotsim (fr.tp.inf112.projects.robotsim/)

Options

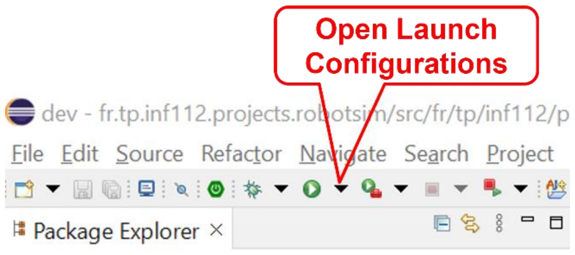
- [✓] Search for nested projects
- [✓] Copy projects into workspace
- [ ] Close newly imported projects upon completion
- [ ] Hide projects that already exist in the workspace

Working sets

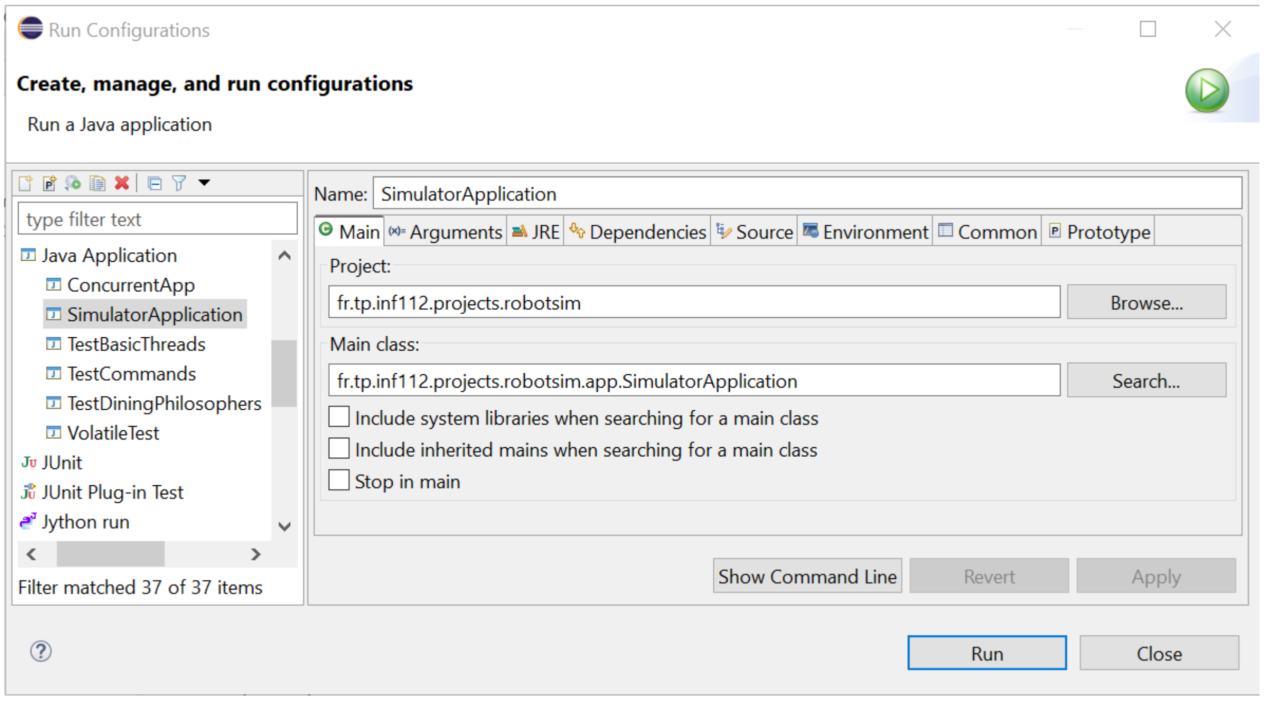
- [ ] Add project to working sets

Working sets:

The main class of the simulator, which contains the *main()* method entry point of the program, is named *SimulatorApplication*. To run the simulation, open the Eclipse launch configurations window by first clicking the downwards black arrow on the *Run* button from the upper Eclipse tool bar as illustrated in the screenshot below, and then selecting the *Run Configurations...* menu item.


Open Launch Configurations
dev - fr.tp.inf112.projects.robotsim/src/fr/tp/inf112/p
File Edit Source Refactor Navigate Search Project
Package Explorer

In the dialog box that is shown, select the *SimulatorApplication* launch configuration under the *Java Application* branch and click *Run*.



Name: SimulatorApplication

Project: fr.tp.inf112.projects.robotsim

Main class: fr.tp.inf112.projects.robotsim.app.SimulatorApplication

- [ ] Include system libraries when searching for a main class
- [ ] Include inherited mains when searching for a main class
- [ ] Stop in main

From the simulator window that is displayed, click the *Animation>>Start Animation* menu to start the simulation, and verify that robots are moving in the factory to visit some components.

You will notice that *Robot 2* immediately becomes red and does not move. This is because the first component that this robot is given to visit is a charging station named *Charging Station*. However, the door of the room that contains the charging station is closed and therefore *Robot 2* cannot compute a trajectory towards the charging station. *Robot 1* starts by visiting a machine named *Machine 1*, then a machine named *Machine 2*, then a conveyor named *Conveyor 1*, and finally the charging station. Again, because the door of the room that contains the charging station is closed, *Robot 1* stops after visiting the conveyor because it can no longer compute a trajectory towards the charging station.

The model of the factory that is displayed in the simulator window is first constructed by instantiating classes of the model in the main method of the *SimulatorApplication* class. This happens just before opening the simulator window.

The *SimulatorApplication* class can be found in package *fr.tp.inf112.projects.robotsim.app*. Open the class file to modify the code between lines 66 and 77 so that both robots must visit *Machine 1* first and then *Machine 2*. Run the simulator again. What do you observe?

## Removing the Livelock

Although the two robots are not yet executing in their own thread, the lock that you observed is like a livelock. As seen in class, a livelock is a special form of deadlock, but instead of waiting, the threads are continuously working, transferring states between one another.

In the case of the robots, each robot has calculated a path avoiding obstacles in the factory such as walls, closed doors and charging station. However, other robots were not considered in this calculation because they are moving. The naïve solution that was implemented to avoid collision between robots was then simply that each robot check if there is another robot located at its targeted position of the path before moving to the position. If so, the robot simply waits and will try to move again the next time the *behave()* method will be called. In our case, because each of the two robots wants to move to the position of the other robot, they end up waiting for each other indefinitely while being still active.

To remove the lock, observe the content of the *moveToNextPathPosition()* method in the *Robot* class. It first calls a *computeMotion()* method, which computes a *Motion* object, which later will be responsible to update the robot's position. Observe the code of the *computeMotion()* method and change it to remove the lock. Run the simulator again and check that one of the robots is now free to continue moving along its path.

## Making the Factory Components Execution Parallel

You probably observed that while you were able to remove the livelock, the consequence is that one of the robots now is free to move over the other one. We therefore need to fix this new problem. To do this, we will first make the execution of the robots parallel. Then, we will *synchronize* the access to positions in the factory to ensure that only one robot can be at a given position in the factory.

### Making the Factory Components Runnable

To make the execution of each component in parallel, we will use the Java *Runnable* interface as seen during the lecture. Therefore, first make the *Component* class implement the *Runnable* interface. Next, create a *run()* method in the *Component* class. In this method, create a *while* loop that will only terminate once the simulator has stopped. To determine if the simulator has stopped, you can use the *isSimulationStarted()* method of the *Component* class. In the loop, call the *behave()* method and then pause the thread for say 50 milliseconds.

## Making the Sequential Simulation Loop Parallel

The sequential simulation loop is defined in method *startSimulation()* of the *Factory* class. Open the class and examine the code of the method. A *while* loop iteratively calls the method *behave()* of the factory class, followed by a pause of 100 milliseconds of the main application thread. This iterates until the simulation has been stopped (simulation no longer started). Since this code is now handled in each component thread, modify the code of the *startSimulation()* method so that the *behave()* method of the factory class is only called once.

Now look at the code of the *behave()* method of the factory class. This method simply loops over all factory components and iteratively calls their individual *behave()* method. For most of the factory components, no specific behavior is defined. However, as we have seen for the *Robot* class, the *behave()* method is redefined so that the robot can move towards the components of the factory it has to visit.

Modify the code of the *behave()* method of the factory class so that instead of directly calling the *behave()* method of each component, a new *Thread* is instantiated passing the component as its *Runnable* to the constructor. Next, start the thread.

## Testing the Parallel Robotic Factory Simulator

Run the simulator again and check that the robots still execute as before.

## Synchronizing the Access to Positions in the Factory

The robots of the factory now run in parallel. However, they can still access the same position at the same time. To avoid this problem, we need to synchronize the accesses to positions of the factory.

One way to do so is to delegate the action of moving a component to the factory by creating a new method. This method will take as parameter the motion object and will displace the component after checking that the targeted position is free, otherwise it will simply return and not move. This method will also be declared to be synchronized so that only one thread at a time can actually move and can check that the targeted position if free.