Robotic Factory 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 components (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. You will then need to synchronize the access to positions of the factory by moving components to ensure that a position can only be accessed by one component at a time.

Introducing the Robotic Factory Simulator

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

The RFS is a simulator inspired by the cyber-physical systems lab of the Hasso Plattner Institute. Its description can be found here. It consists of a set of robots in a building that move across different rooms to bring pucks from one place to another. Such places can be production machines or conveyors. Additionally, charging points exist where a robot can go if it detects that its battery level is too low.

Overall Architecture

The diagram of Figure 1 depicts the overall architecture of the simulator. It consists of three main components implementing the Model-View-Controller (MVC) architectural design pattern.

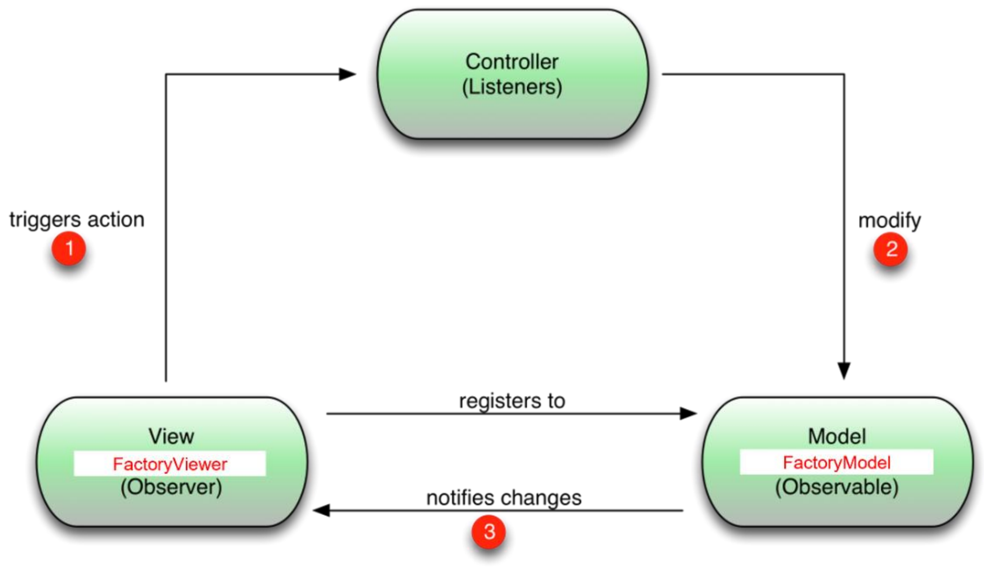


Figure 1: A diagram of the Model-View-Controller (MVC) pattern

The Model component consists of several classes representing the robotic factory data. Such classes can be found under package *fr.tp.inf112.projects.robotsim.model*.

The View consists of a Graphical User Interface (GUI) packaged as a library named *CanvasViewer.jar*. It is used by the simulator to display the factory data model. The viewer can only display two dimensional geometrical figures on a canvas. It provides a set of Java interfaces specifying those figures so that it can draw them. To be viewed by the viewer, the components of the factory model were made to implement those interfaces so that they can be displayed as figures, while the factory is displayed as a canvas representing the factory floor and containing the components.

Finally, the controller plays the role of receiving actions such as starting or stopping the simulation from the view and dispatching them to modify the model accordingly. It is implemented as a class named *SimulatorController* that can be found in package *fr.tp.inf112.projects.robotsim.app*. As illustrated in the figure, the view registers itself as an Observer of the Observable data model so that it can be notified when the model data is changed. In such case, the view simply refreshes itself so that the updated data is displayed.

Factory Data Model

The class diagram of Figure 2 depicts the model of the factory.

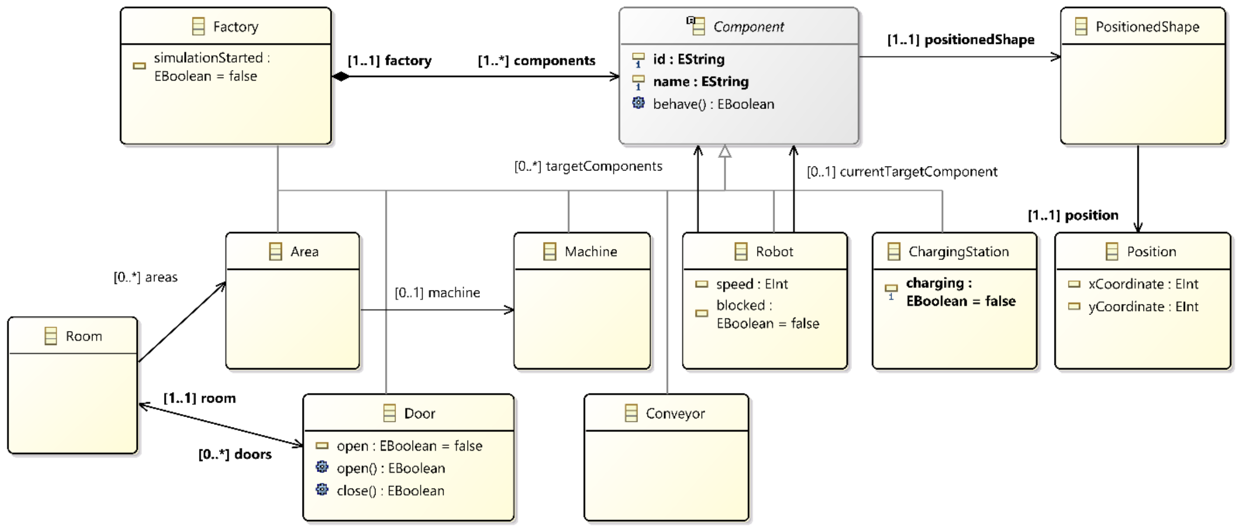


Figure 2: Class diagram of the robotic factory data model

An abstract class named *Component* is the parent class of all specific concrete components of the factory such as *Room*, *Area*, *Machine*, *Door*, *Conveyor*, *Robot* and *ChargingStation*. These subclasses have additional attributes specific to their nature such as components that they need to know. For instance, a *Room* needs to know its *Doors* and *Areas* via dedicated references, and an *Area* itself has a reference to an optional production *Machine* it may contain.

The Factory, which is also considered a component, also extends the *Component* class. In addition, it contains all components of the factory via its *components* reference attribute.

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* at this abstract component level, it can be redefined for coding the behavior of specific components like that of the *Robot* component.

The simplified behavior of the robots of the RFS consists of sequentially visiting a set of other components in the factory. For this, the *Robot* class declares an attribute named *targetComponents* that contains the components to be visited by the robot. It also declares an attribute named *currentTargetComponent* storing the current target component towards which the robot is moving. The class redefines the *behave()* method of the *Component* class whose code is given in the following:

```
@Override
public boolean behave() {
    if (getTargetComponents().isEmpty()) {
        return false;
    }

    if (currTargetComponent == null || hasReachedCurrentTarget()) {
        currentTargetComponent = nextTargetComponentToVisit();

        computePathToCurrentTargetComponent();
    }

    moveToNextPathPosition();

}
```

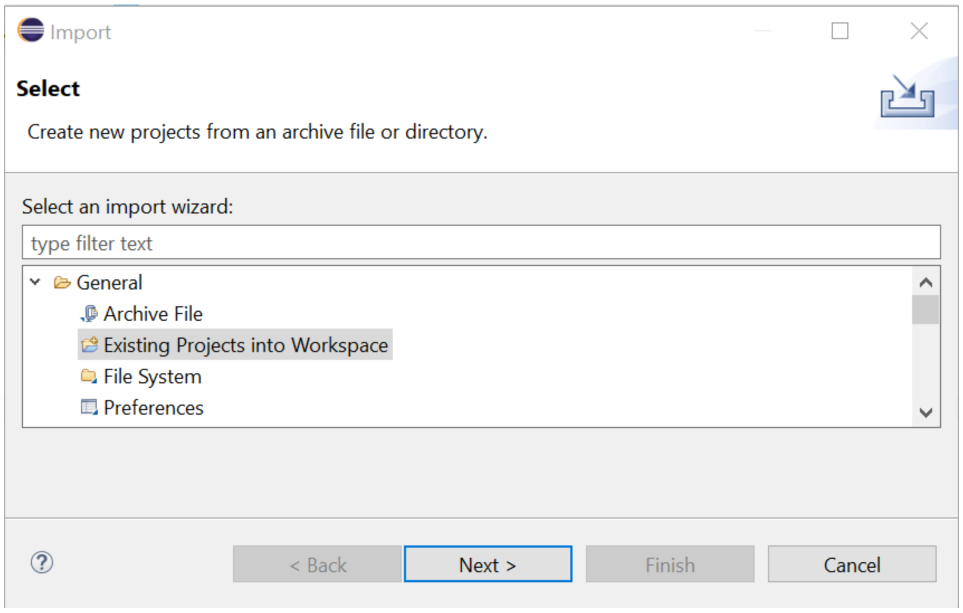
This method first checks if the list of target components is empty and if so, *false* is immediately returned to indicate that no behavior was performed. Otherwise, if the current target component is null (it has not yet been initialized) or if the robot has reached the current target component, the 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 here. 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 as shown in Figure 3. Click *Next*.

In the dialog box that is displayed, check the *Select archive file* radio button as shown in Figure 4 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 on a subfolder of the project 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

Select

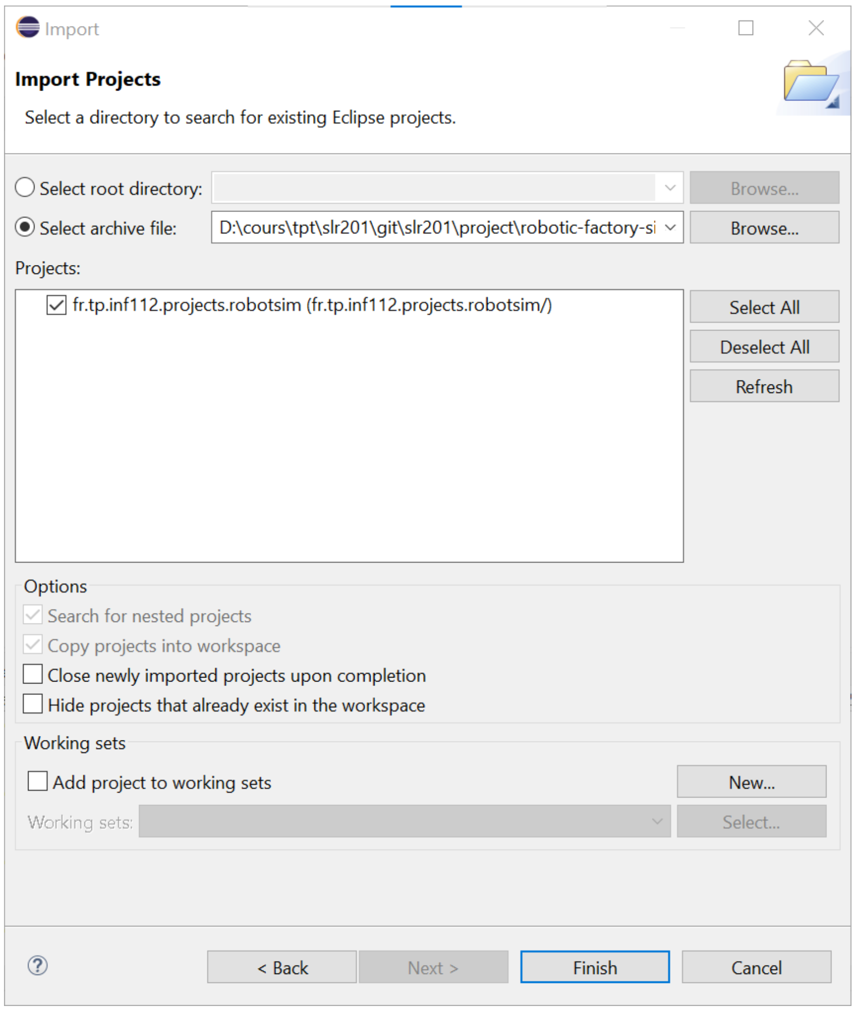
Create new projects from an archive file or directory.

Select an import wizard:

type filter text

- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Preferences

Figure 3: Importing an existing project into the workspace


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:

Figure 4: Importing an archived project

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 Figure 5, and then selecting the *Run Configurations…* menu item.

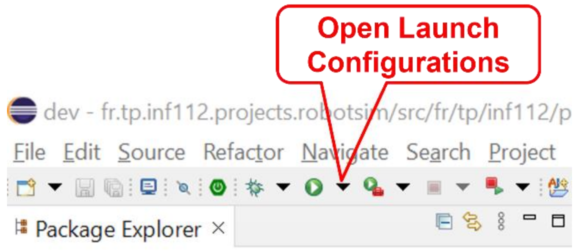
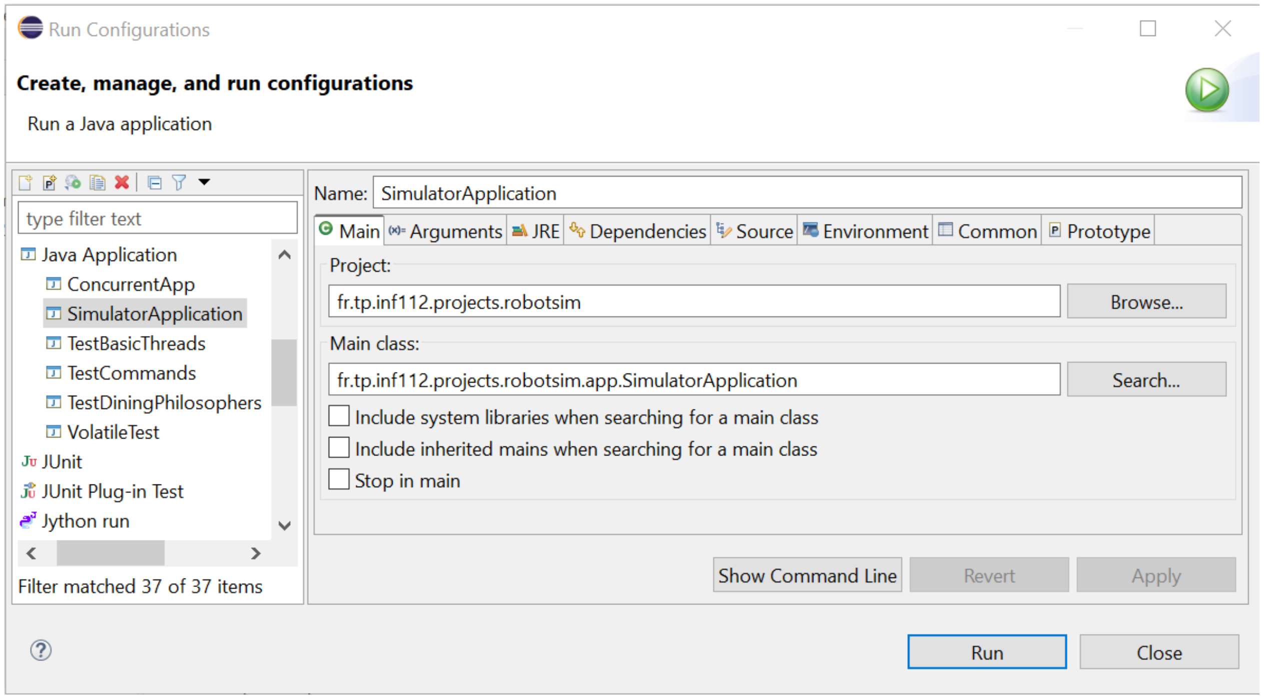

Open Launch Configurations
dev - fr.tp.inf112.projects.robotsim/src/fr/tp/inf112/p
File Edit Source Refactor Navigate Search Project
Package Explorer

Figure 5: Opening launch configurations

In the dialog box of Figure 6, select the *SimulatorApplication* launch configuration under the *Java Application* branch and click *Run*.


Run Configurations

Create, manage, and run configurations

Run a Java application

type filter text

- Java Application
  - ConcurrentApp
  - SimulatorApplication
  - TestBasicThreads
  - TestCommands
  - TestDiningPhilosophers
  - VolatileTest
- JUnit
- JUnit Plug-in Test
- Jython run

Filter matched 37 of 37 items

Name: SimulatorApplication

Main | Arguments | JRE | Dependencies | Source | Environment | Common | Prototype

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

Figure 6: The simulator launch configuration

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 *Machine 2*, then *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 robots, each robot has calculated a path to avoid obstacles in the factory. However, other robots were not considered in this calculation because their position is not constant. The naïve solution that was implemented to avoid collisions between robots was then 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 its *behave()* method is called.

In this implementation, a livelock occurs when two robots are traveling in opposite directions. For example, Robot 1 wants to move to the position occupied by Robot 2, and Robot 2 wants to move to the position occupied by Robot 1, and both robots end up waiting for each other indefinitely. While still active, each robot constantly checks for the presence of the other robot at the targeted position.

To remove the lock, the first step is to create a method in the *Robot* class that can be called to detect if the robot is in a livelock state. When a robot is blocked, it will memorize its next target position on the path in an attribute named *memorizedTargetPosition*. To do determine if there is a live lock, the method should first get the value of the *memorizedTargetPosition* attribute. If it is null, it means that the robot is not blocked and that the robot is therefore not in a live lock state. Otherwise, the robot will need to obtain the component (robot) that is located at its *memorizedTargetPosition*. This can be achieved by calling a method on the robotic factory class that you can create looking at the implementation of the *hasMobileComponentAt()* method for help. Then, if the *memorizedTargetPosition* attribute of the other robot is the same as the current position of the robot, then there is a live lock. The code below implements this algorithm.

```
public boolean isLivelyLocked() {
    final Position memorizedTargetPosition = getMemorizedPosition();

    if (memorizedTargetPosition == null) {
        return false;
    }
    final Component otherRobot = getFactory().
        getMobileComponentAt(memorizedTargetPosition, this);

    return otherRobot != null &&
           getPosition().equals(((Robot) otherRobot).getMemorizedTargetPosition());
}
```

Now we need to call the *isLivelyLocked()* method and solve the lock in case of live lock. For this, observe the content of the *moveToNextPathPosition()* method of the *Robot* class. It first calls a *computeMotion()* method, which returns a *Motion* object. Observe the code of the *computeMotion()* method. If it finds that there is another mobile component (or robot) at the next position to be reached by the robot, it sets the *nextPosition* attribute to memorize the same next position to target the next time the behave method will be called and returns a null motion object.

Modify the *moveToNextPathPosition()* method of the *Robot* class to solve the live lock if it occurs. Check if the displacement of the robot is zero. If so, it means that the robot has not been able to move to its next position. In such case, call the *isLivelyLocked()* method to check if the robot is in a live lock state. If so, create and call a new method to compute a neighboring position for the robot to move to that is free of obstacles. Move the robot to this new next position and recompute the path to the current target component from this new position. This algorithm is illustrated by the code below:

```
private int moveToNextPathPosition() {
    final Motion motion = computeMotion();

    final int displacement = motion == null ? 0 : motion.moveToTarget();

    if (displacement != 0) {
        notifyObservers();
    }
    else if (isLivelyLocked()) {
        final Position freeNeighbouringPosition = findFreeNeighbouringPosition();

        if (freeNeighbouringPosition != null) {
            nextPosition = freeNeighbouringPosition;
            displacement = moveToNextPathPosition();
            computePathToCurrentTargetComponent();
        }
    }

    return displacement;
}
```

Run the simulator again and ensure that the livelock does not occur anymore. In other words, when a live lock occurs, one of the robots should decide to move one position away from the other robot's path to let it pass and then recompute a new path towards the current component to visit in the factory.

Making the Factory Components Execution Parallel

The next step is to make the execution of the factory components parallel. Then, we will *synchronize* the access to positions in the factory to ensure that robots move to their target positions in a synchronized way to avoid inconsistent data.

Making the Factory Components Runnable

To make the execution of each component in parallel, use the Java *Runnable* interface as seen during the lecture. 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 been stopped. To determine if the simulator has been stopped, you can use the *isSimulationStarted()* method of the *Component* class. In the loop, call the *behave()* method of the component and then pause the thread for some time like 50 milliseconds.

Making the 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 some time. The loop ends when the simulation has been stopped (simulation no longer started). Since this loop is now handled in each component thread, modify the code of the *startSimulation()* method to remove the loop 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. Modify the code 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 so that the component *behave()* method is called from the component's *run()* method.

Synchronizing the Access to Positions in the Factory

The robots of the factory now run in parallel. However, they can still potentially access the same position at the same time. This is because when a robot is checking if there is another robot at the position it wants to reach, another robot may be moving to this position at the same time in parallel.

To avoid this problem, access to positions of the factory must be synchronized. One way to do so is to delegate the action of moving a component to the factory object by creating a new synchronized method in its class like the one below.

```
public synchronized int moveComponent(final Motion motion,
                                      final Component componentToMove) {
    ...
}
```

This method will take as parameter the motion object and call its *moveToTarget()* method to displace the component after checking that the targeted position is free. Otherwise, the method will simply return a displacement of zero and not move.

Testing the Parallel Robotic Factory Simulator

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