TP06: Simulating the robotic factory model using MVC

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

In the previous practical session, you modified your robotic factory object model to implement the model interfaces of the graphical Canvas Viewer interface (`canvas-viewer.jar`) provided to you. This allowed visualizing your model as 2D figures with different colours, styles, and geometric shapes.

In this lab, you will define the **behavioural view** of your model to simulate it. Then, to visualize this simulation using the *Canvas Viewer* interface, you will need to create a class that implements the controller interface, enabling control over the display and simulation of your model.

1 Specifying the model behaviour

You will now model the **behavioural** part of the robotic factory. This involves defining *methods* in the various classes of factory *components* to describe how their *states*, represented by the values of their *attributes*, evolve over time and in response to events in the factory.

1. First, define a new method named `behave()` in the `Component` class of your model. Initially, this method will be empty. Each factory component **whose state can change over time** will *override* this method to specify its behaviour when the factory operates.

2. In the `Factory` class, override the `behave()` method inherited from the `Component` class. For each component in the factory, call the `behave()` method of its subcomponent. Thus, any method overridden by a subclass of `Component` will be executed, simulating the specific behaviour of each subclass.

Moving robots

3. Specify the behaviour of the `Robot` class. Initially, this behaviour involves moving from one component to another within the factory. To do this, add an attribute to the `Robot` class to hold a **list of components** the robot must visit as it moves through the factory.

4. In the same `Robot` class, override the `behave()` method inherited from `Component`.

- Within this `behave()` method, retrieve the first component from the list of components to be visited by the robot. Then, call a method named `move()` that you will define as described in the next section.
- Before calling `move()`, verify if the robot has reached the target component by checking if its position matches that of the component.
- If so, retrieve the next component in the list and direct the robot towards it. When the end of the list is reached, the robot must return to the first component in the list of components to visit.

5. In the `Robot` class, define the `move()` method.
   - In this method, increment (or decrement, as needed) the robot's `x` and `y` coordinates by a value defined by an attribute specifying the robot's `speed`, so that the robot moves closer to the component to visit. For now, assume there are no obstacles between the robot and the component it is heading toward.

2 Observers

6. Open the `canvas-viewer.jar` library located in the `libs` directory of your Eclipse project for the robotic factory simulator. Within the package `fr.tp.inf112.projects.canvas.controller`, you will find the `CanvasViewerController`, `Observable`, and `Observer` interfaces.

7. Open the `CanvasViewer` class from the `canvas-viewer.jar` library. You will notice that this class, which serves as the view (and thus the observer) in the MVC pattern, implements the `Observer` interface. This interface defines only one method, `modelChanged()`, which must be called by your model every time its data changes, so that the view(s) can refresh the data and keep the display consistent with the model's data.

8. Examine the body of the `modelChanged()` method in the `CanvasViewer` class. It simply calls the *repaint* methods of the menu bar and the panel that displays the shapes from your model. When displaying the menus in the menu bar, the `"Start Animation"` and `"Stop Animation"` menus will be enabled or disabled depending on whether your model is currently running a simulation, as determined by a call to the `isAnimationRunning()` method from the controller interface presented later in this document.

Implementing the Observable interface

To notify the *view* when its data changes, the *model* must implement the `Observable` interface:

```
public class Factory extends Component implements Canvas, Observable {
    ... }
```

9. This interface contains methods to *add* and *remove* observers. To implement them, declare an *attribute* in the `Factory` class to hold its observers.

Notify the view(s) when the model's data has changed

Since the simulator follows an MVC architecture, you will need to modify your model's code so that any change in its data (such as the coordinates of a moving robot, for example) can notify the views that have been registered as observers of the model. This way, the views can refresh and display the updated data from the model.

10. To achieve this, in the `Factory` class, you will need to create a `notifyObservers()` method that will call the `modelChanged()` method of each observer that has been registered with the model.

11. This `notifyObservers()` method should be called by the model whenever its data is **modified**, for all the classes in your model that need to be visualized by the graphical interface.

There are several ways to achieve this. A simple approach is to add an attribute of type `Factory` to the `Component` class, so that each component can call the `notifyObservers()` method of the `Factory` class whenever its data is changed through its *setter* methods.

3 Implementing the Controller interface

12. Open the `CanvasViewerController` interface from the `canvas-viewer.jar` library. Create a `SimulatorController` class implementing this interface. Pass an instance of this class to the `CanvasViewer` constructor for visualizing your model.

13. The `CanvasViewerController` interface inherits from the `Observable` interface. This means that the view registers with the model **indirectly** through the controller. Therefore, in your controller, you will need to maintain a reference to the model and implement the methods of the `Observable` interface by calling the corresponding methods of the model.

By reading the Javadoc of the methods in the `CanvasViewerController` interface, provide implementations for the required methods. The controller is responsible for starting and stopping the simulation using the `startAnimation()` and `stopAnimation()` methods, which are called by the view when the corresponding menus are selected. Additionally, a call to the `isAnimationRunning()` method allows the view to check whether the simulation is currently running, so it can manage the activation of the animation control menus.

14. To implement the `startAnimation()`, `stopAnimation()`, and `isAnimationRunning()` methods in your controller class, add an attribute to the `Factory` class to keep track of whether the simulation has started or not.

15. Also, add the `startSimulation()`, `stopSimulation()`, and `isSimulationStarted()` methods to the `Factory` class to manage this attribute. Do not forget to *notify* the observers when the value of this attribute changes.

16. In the `startAnimation()` method of the controller, copy the following code:

```
factoryModel.startSimulation();

while (factoryModel.isSimulationStarted()) {
    factoryModel.behave();

    try {
        Thread.sleep( 200 );
    }
    catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

The graphical interface will take care of launching the simulation of your model in a dedicated task (*thread* or *lightweight process*) to avoid interrupting its display, which runs in a specific JVM task called the *Event Dispatch Thread*.

The `sleep()` method is used to wait for 200 milliseconds between each execution of the `behave()` method of the factory, to prevent the animation from running too quickly and allowing the user to better visualize the simulation. This value can be changed if needed.

Note the `try-catch` instructions used to handle the `InterruptedException`, which will be thrown if another task interrupts the task in which the simulator is running.

17. Finally, implement the `stopAnimation()` and `isAnimationRunning()` methods in your controller class by delegating these operations to the model.

4 Launching the simulation

To test your simulation, follow these steps:

18. In a test class, instantiate a factory containing: **1 robot**, **2 machines**, and **1 charging station**.

19. Add these three components to the list of components to be visited by the robot.

20. Instantiate the `CanvasViewer` class from the graphical interface, this time using the constructor that takes a controller as a parameter.

21. Verify that your robot moves correctly, visiting the two machines and the charging station sequentially.