TP05: Visualizing the robotized factory model through interfaces

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

In the previous practical session, we modelled the **structure** of the robotized factory for which we want to develop a simulator. We created an **object model** of the factory. In this session, we will modify this model to make it visualizable through a graphical interface provided to you. To do so, your model must implement the **Java interfaces** provided by this graphical interface.

The graphical interface is very simple. The only thing it can do is display two-dimensional (2D) geometric shapes of *rectangular*, *oval*, or *polygonal* forms, in various colours, with different line styles (dashed and thickness) on a drawing *canvas*.

As seen in class, an interface defines a **perspective** or a **property descriptor** for the classes that implement it. Thus, the provided graphical interface provides Java interfaces specifying a simple **2D shape** viewpoint. Each component of the robotized factory can indeed be represented as a 2D shape on a canvas. Therefore, the classes in your model need only **implement** the provided 2D shape interfaces for the graphical interface to display them.

This set of interfaces provided by the graphical interface serves as a **contract** between the model and the graphical interface for displaying the model. This way of integrating different software components is a common programming practice.

1 The graphical interface for shape visualization

1. Download the graphical interface library for shape visualization (canvas viewer), which can be found here. This library, named `canvas-viewer.jar`, is essentially a set of compiled Java classes (`.class` files) that can be used by other Java applications.

2. Open the `canvas-viewer.jar` file with a file compression utility like 7-Zip. You will see that it is simply an archive containing `.class` files arranged in the same directory structure as the class packages. You will also find `.java` files, allowing you to view the source code of the graphical interface and even set breakpoints to understand what happens using a debugger.

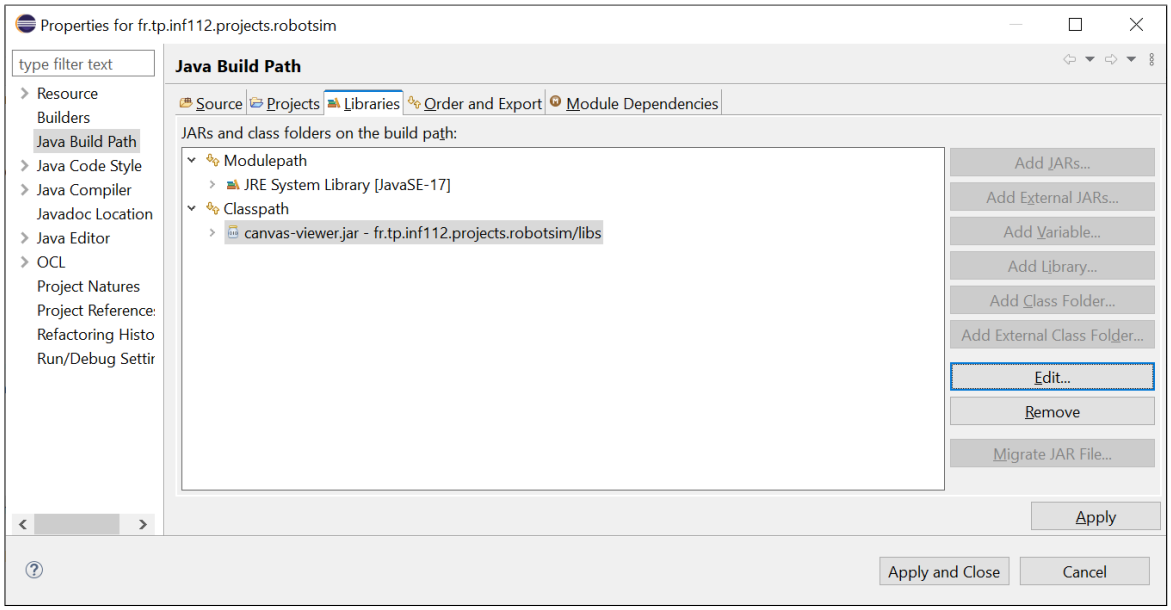
Configuring the simulator project to use the graphical interface library

3. In the project or package browser in Eclipse, select your simulator project, right-click, and click `New` >> `Folder` to create a directory named `libs`.

4. Move the `canvas-viewer.jar` file to this directory (you can also directly drag and drop the file from your operating system's file browser into the project browser in Eclipse).

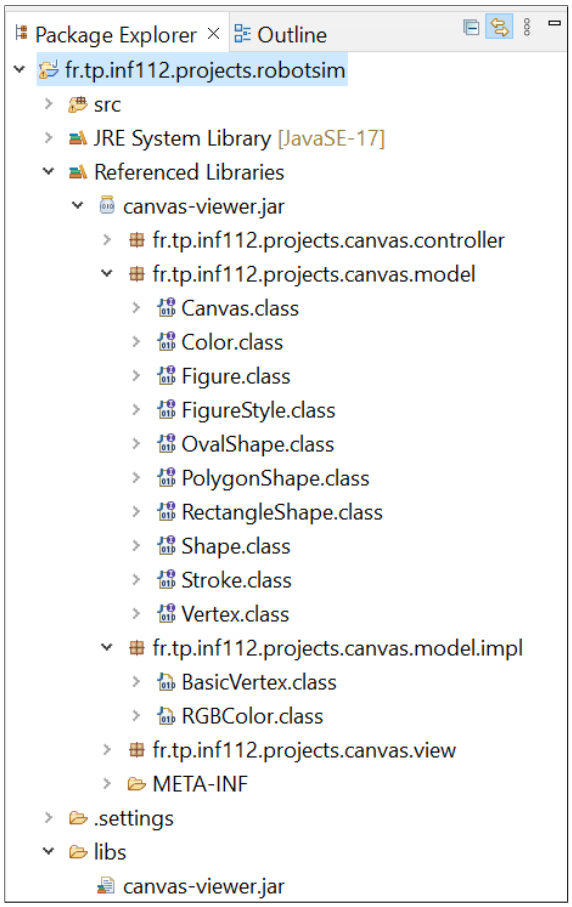
5. Select your simulator project, right-click, and click `Properties`.

6. In the dialogue box that appears, select the `Java Build Path` section, then the `Libraries` tab on the right side of the window. Select `Classpath`, then click the `Add Jars...` button and navigate to the `canvas-viewer.jar` file you added in the `libs` directory.



7. In the package explorer (or project browser), expand the `Referenced Libraries` branch. You will see the `canvas-viewer.jar` library you just added to the project.

8. You will also see the packages provided by this library. Open the packages `fr.tp.inf112.projects.canvas.model` and `fr.tp.inf112.projects.canvas.model.impl`.



9. Examine the contents of these packages. In `fr.tp.inf112.projects.canvas.model` you will find interfaces such as `Canvas`, `Figure`, `Style`, and `Shape`.

   - The `Canvas` interface describes canvas properties such as its name, width, height, and a collection of shapes to display on the canvas.
   - The `Figure` interface describes 2D shape properties such as its name, position on the canvas (x and y coordinates), style (characterized by colour and contour), and shape, as described by the `Shape` interface.
   - The graphical interface can display **only three types of shapes**, whose properties are described by the sub-interfaces `RectangleShape`, `OvalShape`, and `PolygonShape`.

   To make them easier to use, all these interfaces include Javadoc documentation explaining the various methods required by the interfaces.

Note 1: In `fr.tp.inf112.projects.canvas.view` you will find the classes that implement the graphical interface used to visualize your robotized factory model. Understanding these classes is unnecessary, as graphical interfaces are not part of this course curriculum.

Implementing the canvas interfaces with the robotized factory model

To display your model in the graphical interface, you must determine which interfaces the classes of your robotized factory model should implement. For example, the factory, which consists of a layout where the components of the factory are positioned, can be seen as a `Canvas`. Similarly, any component of the robotized factory can be represented by a 2D geometric shape (or *figure)*.

10. Thus, your abstract `Component` class must implement the `Figure` interface:

```
public abstract class Component implements Figure
```

Name and position of the shape on the canvas

11. The `Figure` interface requires a *name* (`getName()` method) for the shape, displayed on the top-left corner of the shape on the canvas. It also requires the *coordinates* of the shape to know where to draw it on the canvas.

Style of the shape

12. The `Figure` interface also requires knowing the display style of the shape. This style is characterized by a *background colour* for the shape (`getBackgroundColor()` method) and the contour characteristics (*thickness, dash, colour*, as described by the `Stroke` interface).

Shape of the shape

As mentioned earlier, the graphical interface can only draw shapes of rectangular (`RectangleShape`), oval (`OvalShape`), or polygonal (`PolygonShape`) forms.

13. For each concrete subclass of the `Component` class, you must determine the desired shape and define implementation classes for these shapes.

14. Then, each component must define the `getShape()` method of the `Figure` interface to return an instance of the implementation corresponding to the chosen shape for the component.
`For example:` Since the robots in the factory are *circular*, your `Robot` class will return an instance of an implementation class of `OvalShape`, while the `Room` or `Area` classes, corresponding to *rectangular* components, will return instances of an implementation class of `RectangleShape`.

15. Do the same for each class component of your model that needs to be visualized by the graphical interface, except for the class representing the robotized factory, which will be considered rectangular.

The canvas

Indeed, this class should instead implement the `Canvas` interface, which represents the container for shapes. Note that the `Canvas` interface specifies a method `Collection<Figure> getFigures()`. Since all the factory components implement the `Figure` interface, you can directly return the attribute of your `Factory` class containing the components.

16. However, you may need to cast this reference as follows:

```
public Collection<Figure> getFigures() {
    return (Collection) components;
}
```

2 Launching the graphical interface

17. Examine the `CanvasViewer` class in the `fr.tp.inf112.projects.canvas.view` package of the graphical interface library (file `canvas-viewer.jar`). One of the constructors of this class takes a single parameter: an instance of a class implementing the `Canvas` interface. T*his is the constructor you will use to launch the graphical interface*.

18. To this end, create a class named `SimulatorApplication` to represent the simulator application linking the model and the graphical interface.

19. In this class, create a `main()` method.

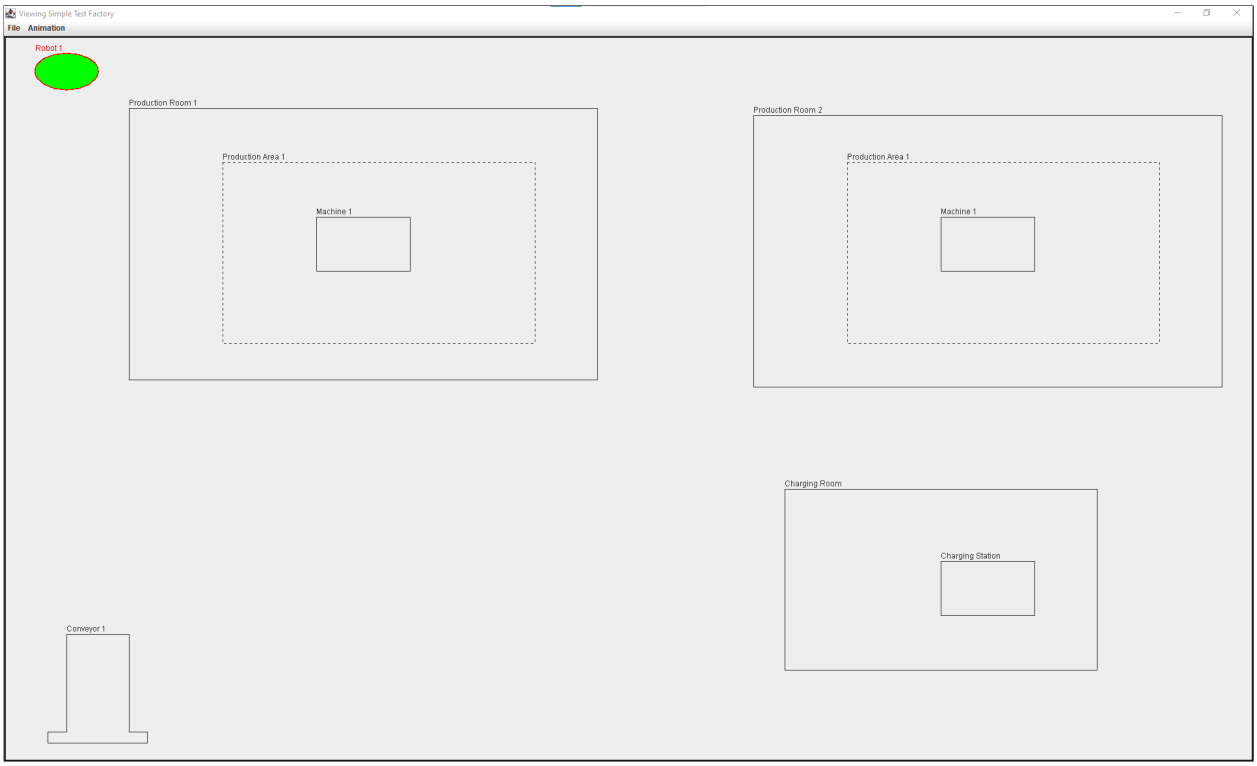
20. In this method, instantiate a factory model as in the previous session, which contains:

    - **3 rooms**, each containing a workspace with **a production machine**,
    - **3 robots**, and
    - **1 charging station**.

21. Then, instantiate the `CanvasViewer` class provided by the graphical interface, providing it with the factory you just created.

22. Run the `main()` method.

23. Verify that your model is correctly displayed in the graphical interface, as illustrated by the following figure.



Note 2: It should be noted that none of the actions defined by the graphical interface menus will work (they will be disabled) except for the action of **quitting** the application. These actions will be defined in the next practical session when you implement the controller interface of the *model-view-controller design pattern*, which we will explain in class first.