TP07: Save the robotic factory model

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

In the previous practical session, you modified your object-oriented model of the robotic factory so that the robots could move from one component of the factory to another. You also implemented the MVC interfaces provided by the `Canvas Viewer` graphical interface so that the movements of these robots could be automatically visualized.

In this practical session, you will develop the **data access layer** (or persistence layer) for your simulator in order to store the data of the robotic factory model on permanent storage (SSD or database). This data will then be able to be re-read when the simulator is restarted.

This persistence layer will rely on various Java classes used to handle **input and output operations**, such as *reading* and *writing* data to files in either text or binary format.

1 Introduction to input / output in Java and data persistence

1. Read the presentation on input and output in Java available here.
2. Then read the presentation on data persistence here.

Update the `Canvas Viewer` library

3. First, download the latest version of the `Canvas Viewer` graphical interface, which can be found here.
4. Then replace your current version with this new one. You will notice differences in the canvas model interfaces, which will be explained later in this document.

The controller interface

The `CanvasViewerController` interface has a new method called `getPersistenceManager()`, as declared in the following code:

```
/**
 * Returns the persistence manager to be used to persist this canvas
 * model into a data store.
 * @return A non {@code null} {@code CanvasPersistenceManager}
 * implementation for the desired data storage kind
 * (file, database, etc.).
 */
CanvasPersistenceManager getPersistenceManager();
```

This method should return an object of a class that implements the `CanvasPersistenceManager` interface.

2 Implement the persistence interface

Examine the documentation for the `CanvasPersistenceManager` interface. As seen in the presentation on data persistence, this interface specifies method signatures for reading, writing (persisting), and deleting a canvas model.

For this simulator, we will implement a data persistence layer that relies on the **file system** of the computer running the program.

5. To simplify the implementation of the `CanvasPersistenceManager` interface, create a class that extends the abstract class `AbstractCanvasPersistenceManager`. This class is provided by the `Canvas Viewer` library.

The `CanvasChooser` interface

The class `AbstractCanvasPersistenceManager` contains an attribute of type `CanvasChooser`, and its constructor takes an object of this type as a parameter. Examine the `CanvasChooser` interface and its documentation. It specifies a method `chooseCanvas()`, which returns a string that uniquely identifies a canvas.

It is not necessary to provide a class that implements the `CanvasChooser` interface. Since the models will be stored in the computer's file system, you can simply instantiate the `FileCanvasChooser` class, which implements `CanvasChooser`, and is provided by the `Canvas Viewer` library.

The `FileCanvasChooser` class allows browsing the computer's file system and selecting a file whose expected extension can be specified by a constructor parameter. When a user of the `Canvas Viewer` graphical interface selects the menu `File` >> `Open Canvas`, the `FileCanvasChooser` object will be used to present a list of files and folders to the user, who can then browse them to choose a model to visualize.

Model a unique identifier for the model

To store your models in the file system (which acts as a database here), each instance of your factory model should provide a string that will serve as its unique identifier in the file system. To this end, the `getId()` and `setId()` methods have been added to the `Canvas` interface, so that your class representing the robotic factory (`Factory`), which implements the `Canvas` interface, will need to provide an attribute for the identifier as well as accessors for this attribute.

When saving the model via the `Canvas Viewer` graphical interface, the `FileCanvasChooser` object will also be used so that the user can choose an existing file name from the file system or enter a new file name. This identifier retrieved from the `FileCanvasChooser` object will then be assigned to the factory model by calling the `setId()` method of your class. It will be composed of the file name and its path within the folder hierarchy.

Make the model classes serializable

To simplify the representation of your model in a file, you will use data serialization, as seen in the course presentation on data input and output.

6. To do this, any class in your model that needs to be saved must implement the `Serializable` interface. Modify the classes in your model to implement this interface. Note that this interface is a **marker** interface, meaning it declares no methods and only serves to mark classes.

However, some data in the classes may not need to be preserved / saved between executions of the program. This is the case for the model's **observers**. It is better not to store these objects, especially since they are instances of Java graphical interface classes, which are not always serializable.

7. Therefore, use the `transient` keyword to declare that the observer attribute should not be serialized.

8. Do not forget to modify the instantiation of the attribute to use **lazy** instantiation, as no class constructor will be called during the deserialisation of the object, as discussed in the presentation on input-output.

9. Then, do the same for any other attributes of your model classes that should not be serialized.

Implement persistence methods

The abstract class `AbstractCanvasPersistenceManager` does not provide the body for the `read()`, `persist()`, and `delete()` methods of the `CanvasPersistenceManager` interface.

10. You need to implement these methods in your persistence class using object serialization streams `ObjectInputStream` and `ObjectOutputStream`, as introduced in the presentation on input and output operations.

11. For the `delete()` method, a simple approach is to instantiate an object of the `java.io.File` class and call the `delete()` method on this object.

3 Test the model persistence

1. To test the persistence of your model, run the simulator as you did in the previous lab session.

2. Use the `Open Canvas` and `Save Canvas` submenus from the `File` menu in the `Canvas Viewer` interface and verify that you can save the model to disk and then correctly reload it in the graphical interface.

4 Set up application logging

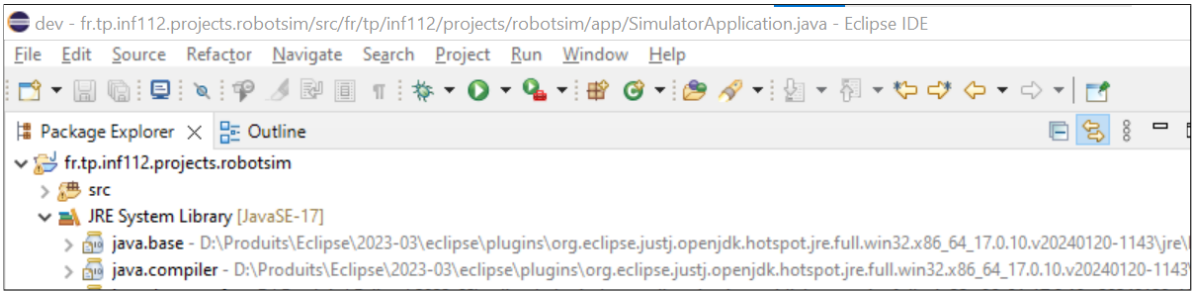
As we saw in class, it is preferable to use a logging library to display the program's execution traces instead of directly using the system console output. This allows storing the traces in other media, such as files.

In this part of the exercise, you will set up logging both to the *console* and in a *log file* using and configuring the JDK's logger.

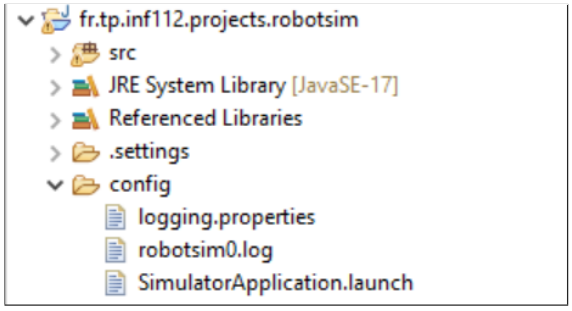
Configure the logger

As seen in class, logging can be configured in Java code or via a configuration file. For any JRE installation, a default configuration file is provided in the installation directory of the JRE. You will copy this file into a directory in your simulator project, then modify it to adjust the logger configuration to your needs.

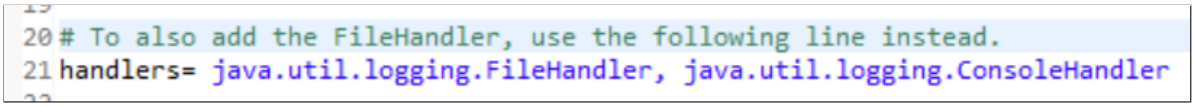
1. As shown in the following screenshot, open the `JRE System Library` directory of your simulator project to locate the installation directory of the JRE in your computer's file system. From this directory, navigate to a subdirectory named `conf` and copy the `logging.properties` file.



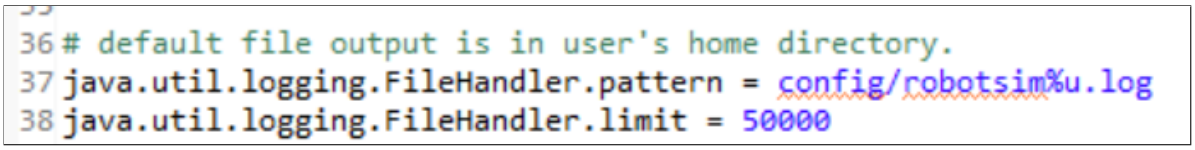
2. Create a directory named `config` in your project and place the previously retrieved `logging.properties` file in it, as shown in the following screenshot.



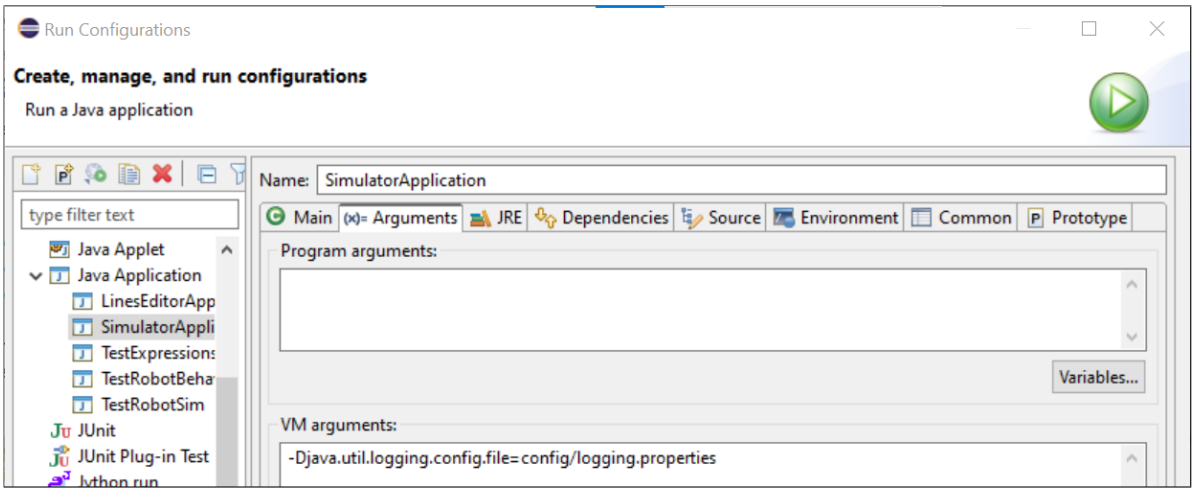
3. Open the `logging.properties` file in Eclipse and modify it so that two handlers are used: a first handler that writes to the console and a second one that writes to a file, as illustrated in the following screenshot.



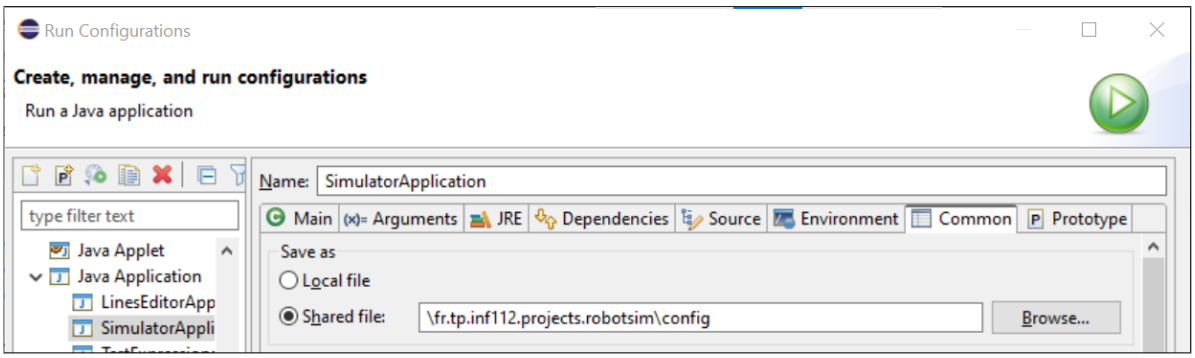
4. Next, as shown in the screenshot below, specify the path of the log file so that it is created in the `config` directory of the simulator project, then save the logger configuration file.



5. Then, specify the location of the logging configuration file to be used by your simulator. This can be done by setting the value of the `java.util.logging.config.file` property, as shown in the following screenshot.



6. Finally, also save your simulator launch configuration in your project. This will make it easier for teachers to evaluate your project, as they can directly use your launch configuration to run your simulator and check its proper functioning. This is done by selecting the `Common` tab and specifying the `config` directory in the text field of the `Shared file` option, as shown in the following screenshot.



Define a logger object to display messages

7. In your simulator application's class (`SimulatorApplication`), create a variable for the logger like the one in the following line.

```
private static final Logger LOGGER = Logger.getLogger(Main.class.
    getName());
```

8. Then, display two information messages notifying the start of your simulator, with two different levels of detail: `info` and `config`, as shown in the following code.

```
LOGGER.info("Starting the robot simulator...");
LOGGER.config("With parameters " + Arrays.toString(args) + ".");
```

5 Verify the logger's proper functioning

1. Run your simulator and observe the messages that appear in the console.
2. In the Eclipse project explorer, refresh the `config` folder. The log file generated by the logger should appear.
3. Open it and verify the messages written in it. Are all the messages properly displayed in both the console and the log file? If not, why?
4. Examine the contents of the `logging.properties` file and change the logging level to `config`. Verify that both messages are correctly displayed in the log file.

5. Then, set a finer logging level (such as `fine`) and restart the simulator. Do the logging messages still appear? You will notice that several messages from the graphical interface now appear at this level of detail. If the simulator's messages no longer appear, check the value of the `java.util.logging.FileHandler.limit` property and refer to its documentation to fix the problem.

Note 1: Note that the log file is written in XML format, while the data is written in a different format in the console. How do you explain this difference?