Robotic Factory Exercise 2
Data Persistence and Client-Server Architecture

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

This second exercise is composed of three parts. First, you will introduce proper logging into the simulator code. Next, you will implement and test a simple web server. Finally, you will modify the current data persistence layer of the simulator for persisting the robotic factory model on the web server that you have created.

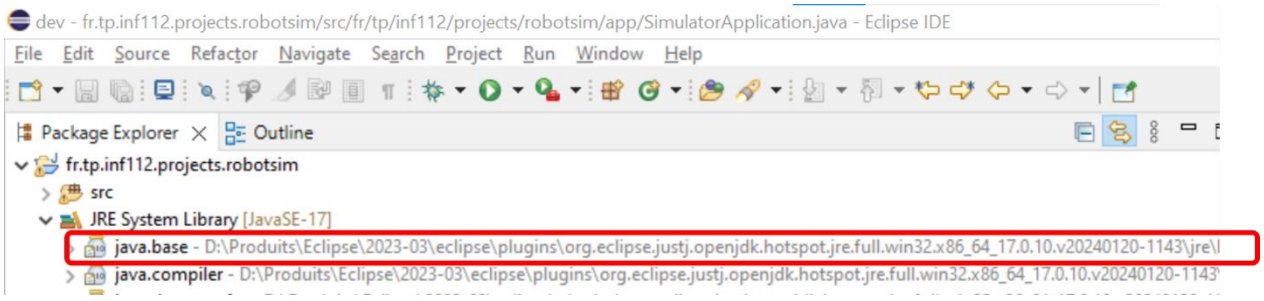
Using Java Loggers to Log the Simulator Execution Traces

We have seen during the lectures that it is better to use a logging library to manage the program execution traces rather than directly using the console output of the system. The purpose is to be able to store the traces in various media such as files or databases for example, by only modifying a logger configuration file without having to modify the program code. In this part of the exercise, you will set up logging both to the console and in a log file by using and configuring the JDK logger.

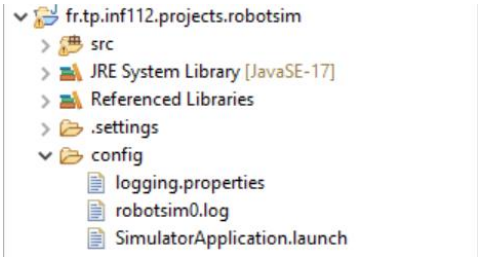
Configuring the Logger

As seen during the lectures, logging can be configured in Java code or via a configuration file. The latter is recommended to avoid the need to modify the code when the level of logging must be changed, to obtain more details of the execution traces for example. For any JRE installation, a default configuration file is provided in the JRE installation directory. You will copy this file into a directory of your simulator project, then modify it to adapt the logger configuration to your needs.

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



Create a directory named *config* in your simulator project and paste the *logging.properties* file there as shown in the screenshot below:



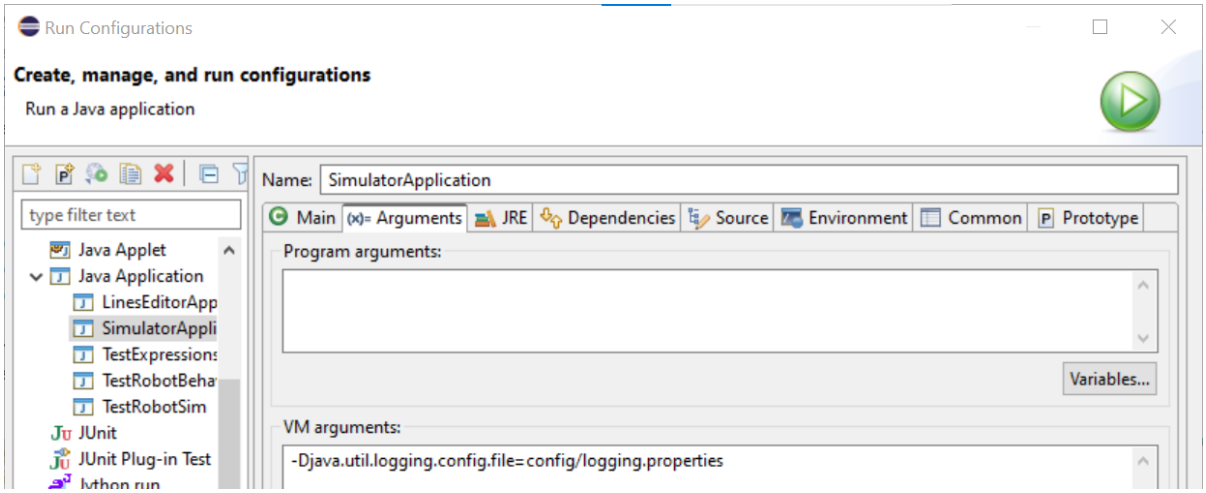
Open the *logging.properties* file and as illustrated by the screenshot below and modify its content so that two handlers are used; a first handler will output the traces to the console, and a second one will write the traces to a file.

```
# To also add the FileHandler, use the following line instead.
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

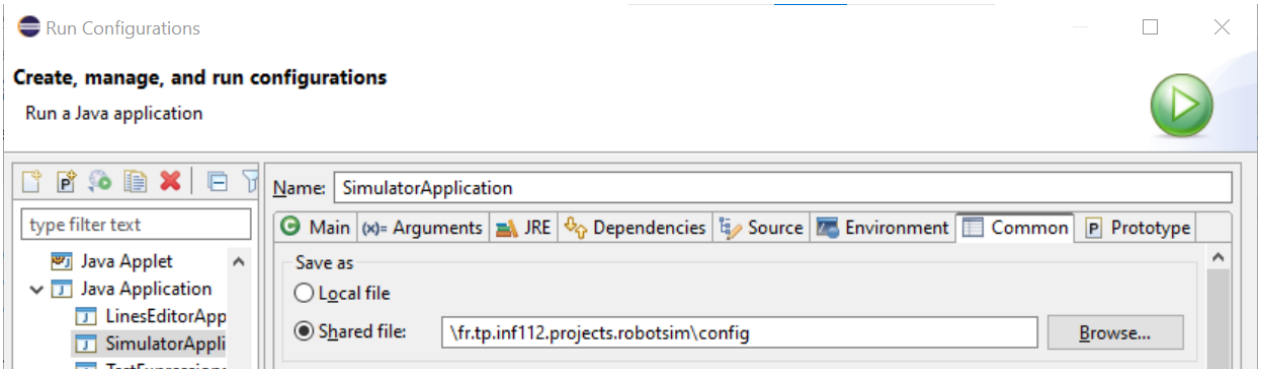
Next, as shown from the screenshot below, specify the path of the log file so that it is stored in the *config* directory of the simulator project and save the logging configuration file.

```
# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = config/robotsim%u.log
java.util.logging.FileHandler.limit = 50000
```

Then specify to the simulator program which logging configuration file to use. This can be done by setting the value of the Java Virtual Machine property *java.util.logging.config.file* to be the desired file as depicted in the screenshot of the launch configuration below.



Finally, ensure that this launch configuration is saved in the *config* folder of the simulator 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 verify that it works properly. This is achieved by selecting the tab named *Common* from the launch configuration window and then specifying the *config* directory in the *Shared file* option as shown in the screenshot below.



Creating Logger Objects and Use them to Log the Execution Traces

In the *SimulatorApplication* class, create a variable for a logger object as shown in the following line:

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

Then replace the calls to the system console with calls to the logger object to log messages of different logging levels as shown in the code snippet below.

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

Search for all system console calls throughout the simulator code to replace them with appropriate logging code.

Checking that Logging Works

Run the simulator and observe the messages that appear in the console. In the Eclipse project explorer, refresh the *config* folder. The log file generated by the logger should appear. Open it and check the messages written there. Are all the messages displayed in the console and in the log file? If not, why?

Examine the contents of the *logging.properties* file and change the logging level to *CONFIG* and verify that both messages are indeed displayed in the log file.

Then specify a finer logging level (FINE for example) and restart the simulator. Are the simulator logging messages still displayed? You will notice that several messages from the GUI are now displayed at this level of details.

If the simulator messages no longer appear, examine the value of the *java.util.logging.FileHandler.limit* property in the logging configuration file and search for its documentation to correct the problem.

Note that the log file is written in xml format, while the logging messages are written in plain text in the console. How do you explain this difference?

Implementing a Simple Distributed "Hello World!" Software Application

As a practice, based on the slides of the lecture, implement a client that sends message *my message* to a server. The server should read the message and respond message **`"I received " + my message + "!"`** to the client. Use sockets to communicate between the two JVMs.

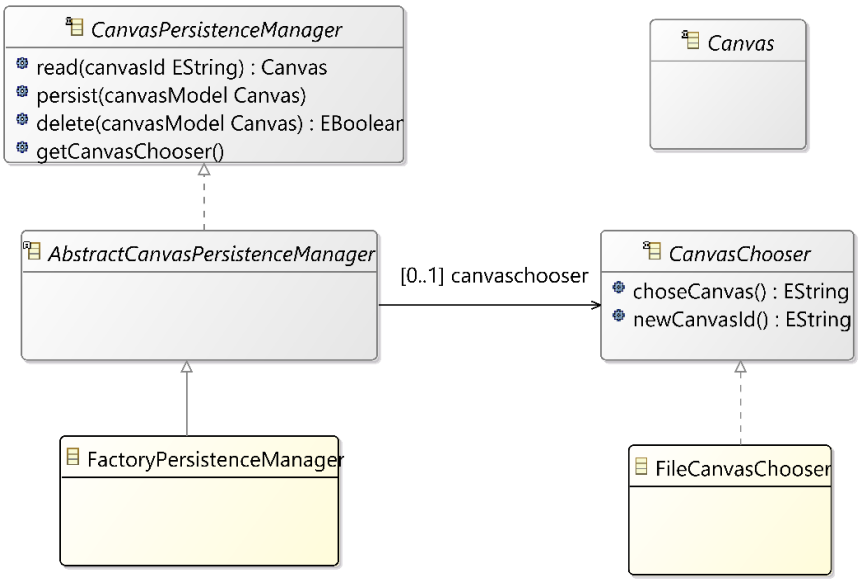
Test your simple "Hello World" distributed application and make sure it works fine.

Migrating the Robotic Factory Simulator so that the Model is Persisted on a Server

In this part of the exercise, you will replace the current persistence layer of the robotic factory simulator so that the data model is stored on a remote server. You will reuse and adapt the server classes that you wrote in the previous section for this.

How the Current Persistence Layer Works

The following class diagram depicts the interfaces and classes of the current persistence layer of the simulator.



The canvas viewer library provides the interface *CanvasPersistenceManager* that declares CRUD (Create, Read, Update, Delete) operations that the viewer calls when users click the *Open / Save Canvas* items of the *File* menu in the simulator user interface. As you can see from the diagram, the CRUD methods of the *CanvasPersistenceManager* interface manipulate *Canvas* objects, and not objects of the *Factory* class, which is the actual robotic factory simulator model class. This is because the canvas viewer is more abstract than the actual robotic factory model. It only knows about a canvas into which it displays figures. Since the factory model classes implement those interfaces, its components can be manipulated by the canvas viewer user interface without knowing the irrelevant details of the factory model.

The canvas viewer library also provides the *CanvasChooser* interface, which is used by the canvas viewer to let users choose a file on the file system into which the model is saved. When users click the *Open Canvas* menu in the simulator user interface, the method *choseCanvas* the *CanvasChooser* interface is called so that users can browse the file system and choose an existing model file to be opened. The method then returns the absolute file path and name of the file that was selected by the user, which is used as the identifier for the model. Then the canvas viewer simply calls the *read()* method of the *CanvasPersistenceManager* interface passing the identifier so that the model can be read from the data store (file system) and displayed by the user interface.

As seen in the class diagram, the class *FactoryPersistenceManager*, which is declared in the simulator model package, is the actual persistence class that is used by the simulator. It extends the *AbstractCanvasPersistenceManager* class, which implements the *CanvasPersistenceManager* interface and declares a reference attribute to the *CanvasChooser* object mentioned above.

Looking at the code of the CRUD methods of the *FactoryPersistenceManager* class, you will notice that it simply persists the model by serializing its objects and saving their binary representation into a file of the computer file system.

Developing a Remote Persistence Layer

The new persistence layer that you will implement will be distributed. A new persistence manager class will be created to play the role of the client that will call the server that you have developed during the previous section. It will still serialize the data, but instead of being directly written to a file, the data will be first sent to the server who will then write it to a file of its file system.

The server will employ a very simple scheme to determine what to do with requests from the client. It will first deserialize the received data into an object and then check for its type. If it is a *String*, the server will assume that the *read()* method was called and that the received *String* is the identifier (or file name) of the factory model to be read and sent back to the client. If the deserialized object received from the client is an instance of the *Factory* class of the simulator model, the server will assume that this data should be persisted to a file, using the *id* attribute of the factory object as file name. Note that the *delete()* method of the persistence manager class does not need to be implemented since it is never called by the simulator user interface.

Implementing a Simulator Persistence Server

First modify the "Hello World" server that you developed in the previous section so that it can save and read factory models on the server file system, as already explained in the previous section. To do so, the server only needs to deserialize the objects received from the client and then simply reuse the original *FactoryPersistenceManager* class, calling the *read()* method if a String object was received, and the *persist()* method if a *Factory* object was received from the client.

Implementing a Remote Persistence Manager

To develop a remote persistence layer, first create a new *RemoteFactoryPersistenceManager* class, and implement the *persist()* and *read()* methods so that sockets are used to send and receive the data to and from the persistence server you created before.

Implementing a Remote Model Chooser (optional)

Since the model files to be read can potentially be distributed on another machine than that of the client, the *FileCanvasChooser* can no longer be used as it can only choose files to be opened from the local file system. Therefore, create a new *RemoteFileCanvasChooser* class extending the *FileCanvasChooser* class. In your class, redefine the *browseCanvases()* method.

When the value of the Boolean parameter of the *browseCanvases()* method named *open* is true, a list of model files should be displayed to the user so that he can choose which model

to open. For this, ask the persistence server for a list of existing model file names on its file system. For simplicity, you can only return the files that are directly located on the working directory of the server, without searching for files in subdirectories. Next, use the *JOPtionPane* class of the JDK, calling its static *showInputDialog()* method and providing an array of file names received from the server to its *selectionValues* parameter. The opened dialog box will automatically display a combo box containing the list of files, and the user can select which file he wants to open from the combo's drop-down menu.

When the *open* parameter is false, it means that the user should be prompted to provide a file name for the model. In such case, again use the *JOptionPane* class of the JDK to simply open a dialog box with only one input text field for the user to enter the name of the file.

Modifying the Simulator Code so that the Remote Persistence Layer is Used

The next step is to modify the simulator application class so that it uses your new remote persistence layer class and its remote file chooser class. For this, open the *SimulatorApplication* class and locate where the *FileCanvasChooser* and the *FactoryPersistenceManager* classes are instantiated. Replace those with your own persistence layer classes.

Finally, check that the saving and opening of factory models works as if the files were stored on the client machine. Optionally, to further test your work, change the server host to the computer of a friend and check that you can open and save models on his computer.