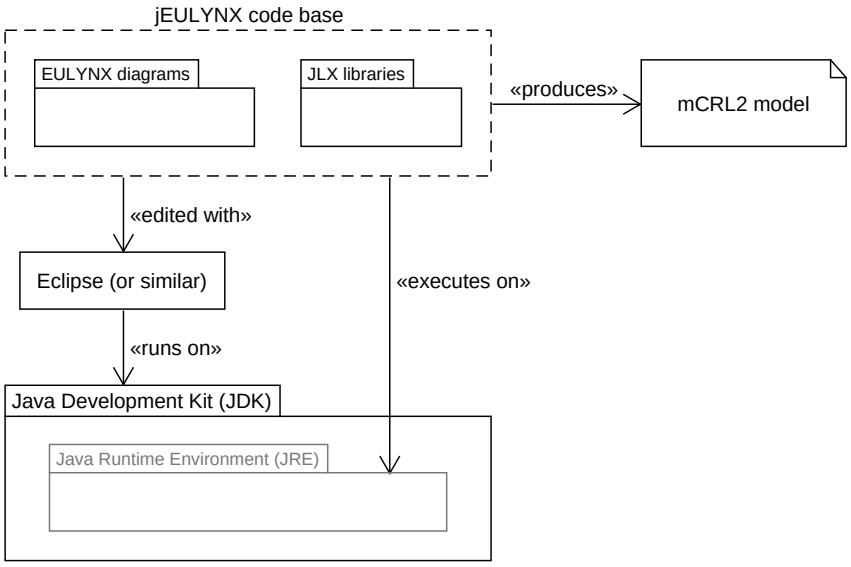
jEULYNX Installation Guide

Djurre van der Wal

May 2023

1 Architecture

We developed our software framework, 'jEULYNX' (i.e. "Java-EULYNX"), in *Java*, a popular object-oriented programming language. We often abbreviate the name to 'jlx' or 'JLX'.
As is standard for Java software, jEULYNX is executed in a Java Runtime Environment. More unusual is that we use Java also for inputting EULYNX diagrams. This means that EULYNX diagrams can be created/edited with the same Java development tools that we use for the functional features of jEULYNX. In the workshop, we use *Eclipse IDE* as the development tool, but any other contemporary Java development tool could be used instead.
Below, we give an overview of the architecture of the jEULYNX software:



The Eclipse IDE requires a *Java Development Kit (JDK)* to run. A JDK is accompanied by a *Java Runtime Environment (JRE)*, on which Java (read: jEULYNX) code can be executed. Among several other functions, jEULYNX can translate EULYNX diagrams into an mCRL2 model.

2 Software setup

To get started with jEULYNX, three pieces of software must be set up:

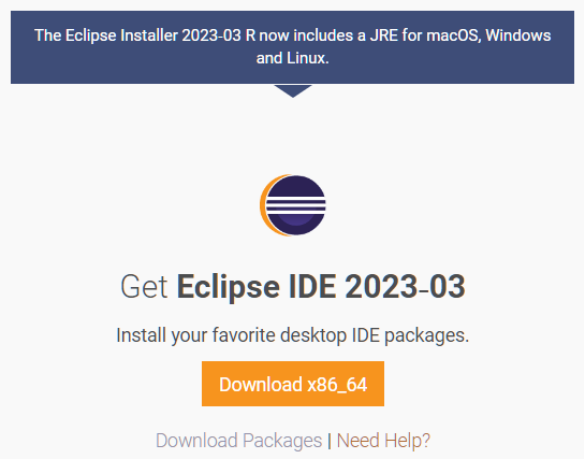
1. Eclipse (or similar);
2. the Java Development Kit (which actually comes automatically with Eclipse); and
3. the jEULYNX code base.

2.1 Installing Eclipse

Eclipse installers exist (see 2.1.1), but portable Eclipse files are also available (see 2.1.2).

2.1.1 Eclipse installer

1. Go to `https://www.eclipse.org/downloads/` and download the Eclipse installer:



2. Run the installer. Eclipse can be used for different purposes, which you must specify during installation. Choose the 'Eclipse IDE for Java Developers' option.

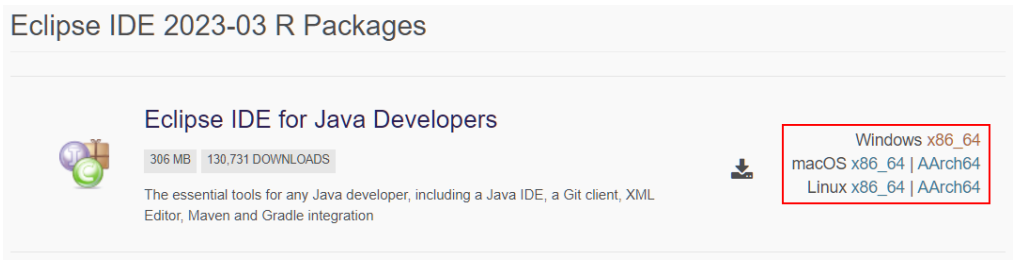
   A few further comments:

   - The installer will ask you to give certain permissions and to trust certain resources. You will need to take these steps.
   - The installer may comment that the process is "taking longer than usual". In our experience, no action is needed when this happens.

   See `https://www.eclipse.org/downloads/packages/installer` for more detailed steps.

2.1.2 Portable Eclipse files

1. Go to `https://www.eclipse.org/downloads/packages`:

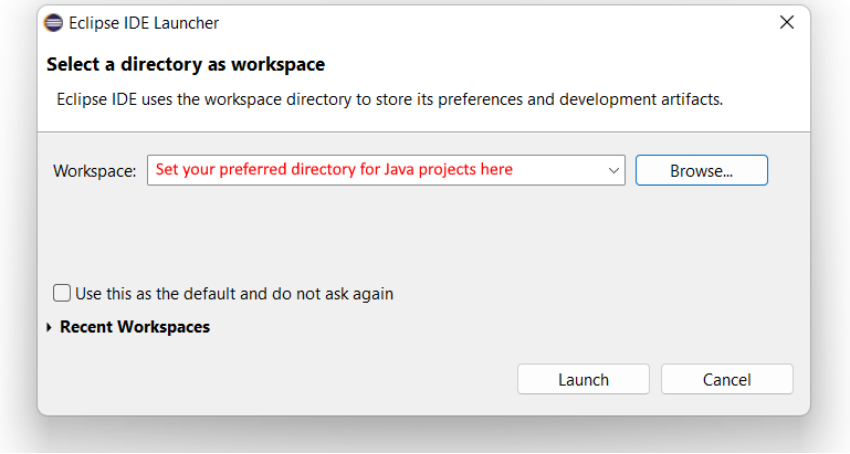


2. Locate the ‘Eclipse for Java developers’ item. Click the appropriate link on the right side. This brings you to a site similar to

3. Click the link to download the archive to a location of your choice.
4. Extract the archive to a location of your choice.
5. The extracted contents of the archive should contain a folder named ‘eclipse’, which should contain some type of executable (‘eclipse.exe’ on Windows). Double-click this executable to launch Eclipse.

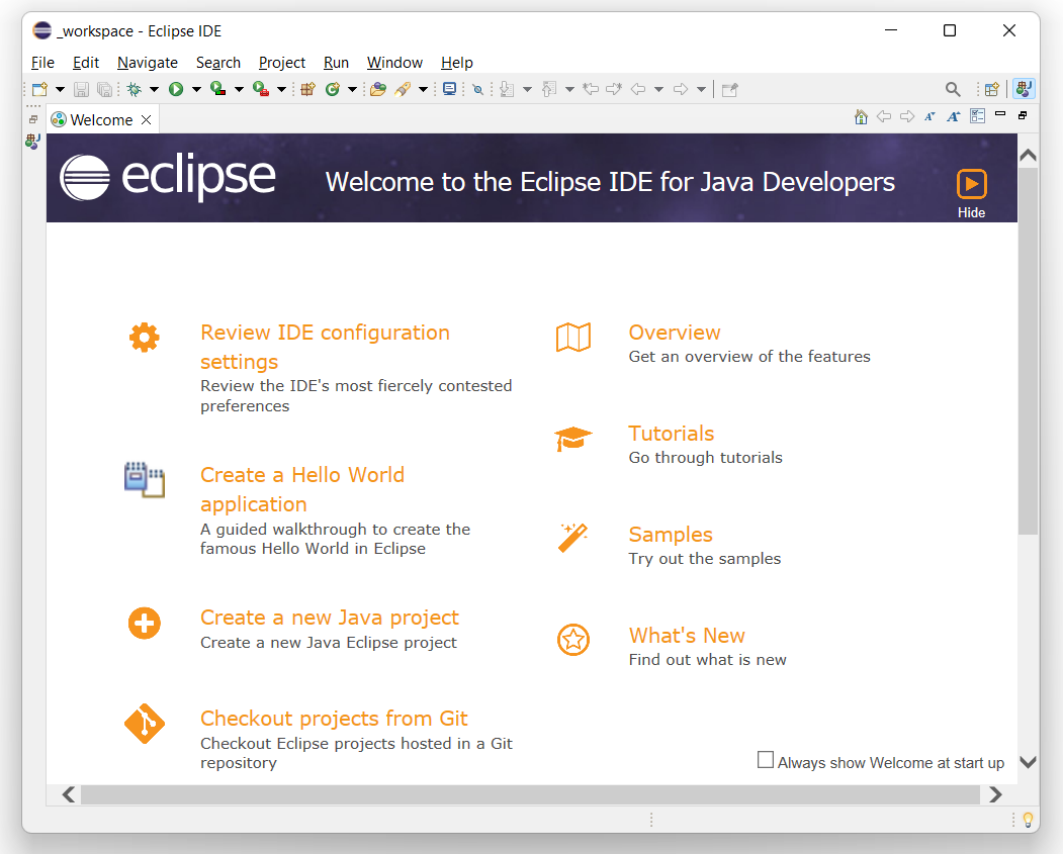
2.2 Importing jEULYNX

1. Start Eclipse. You should see the launcher window:



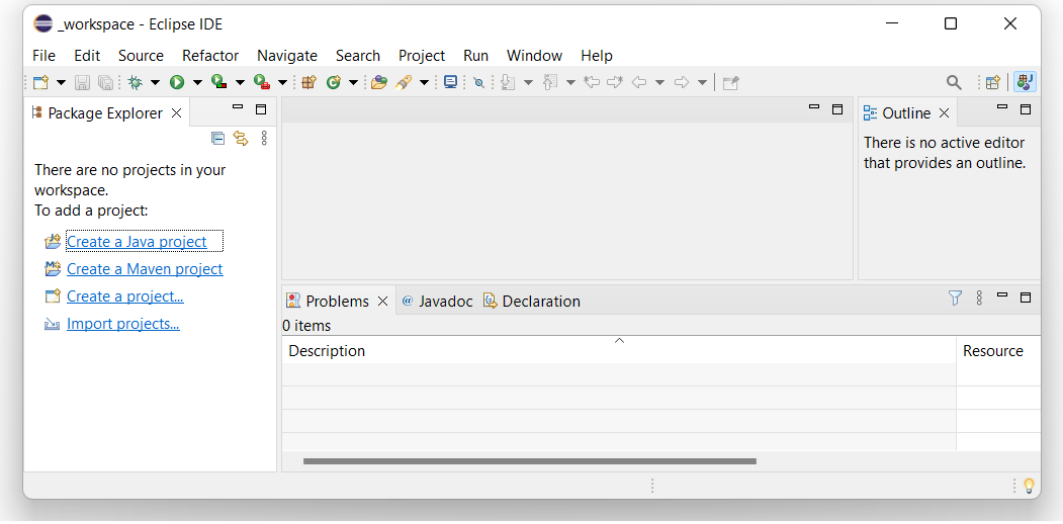
Choose the directory in which you want to place the files for Java projects (i.e. the jEULYNX code base).

2. Click ‘Launch’. After Eclipse has loaded, you should see a window similar to



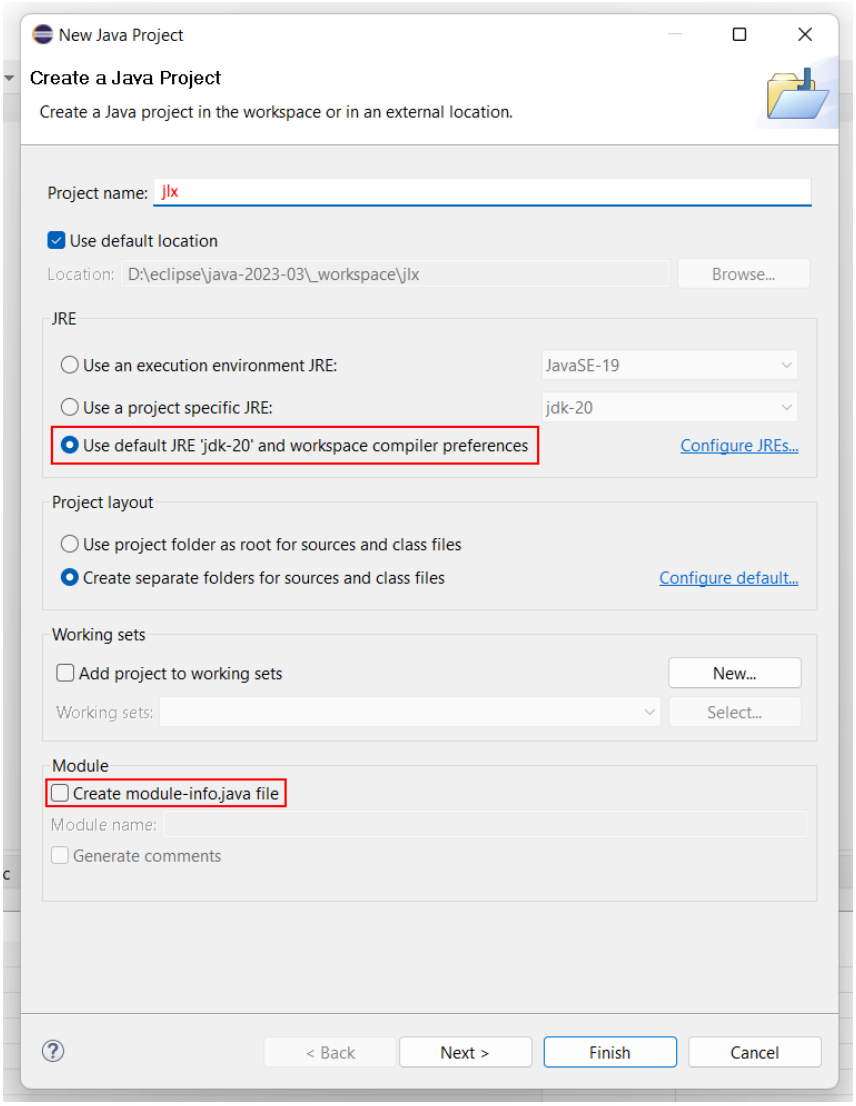
The main area of the windows presents links to several useful resources. Ignore these for now.

3. Click the cross symbol next to ‘Welcome’ at the top of the main area. You should see an empty workspace:



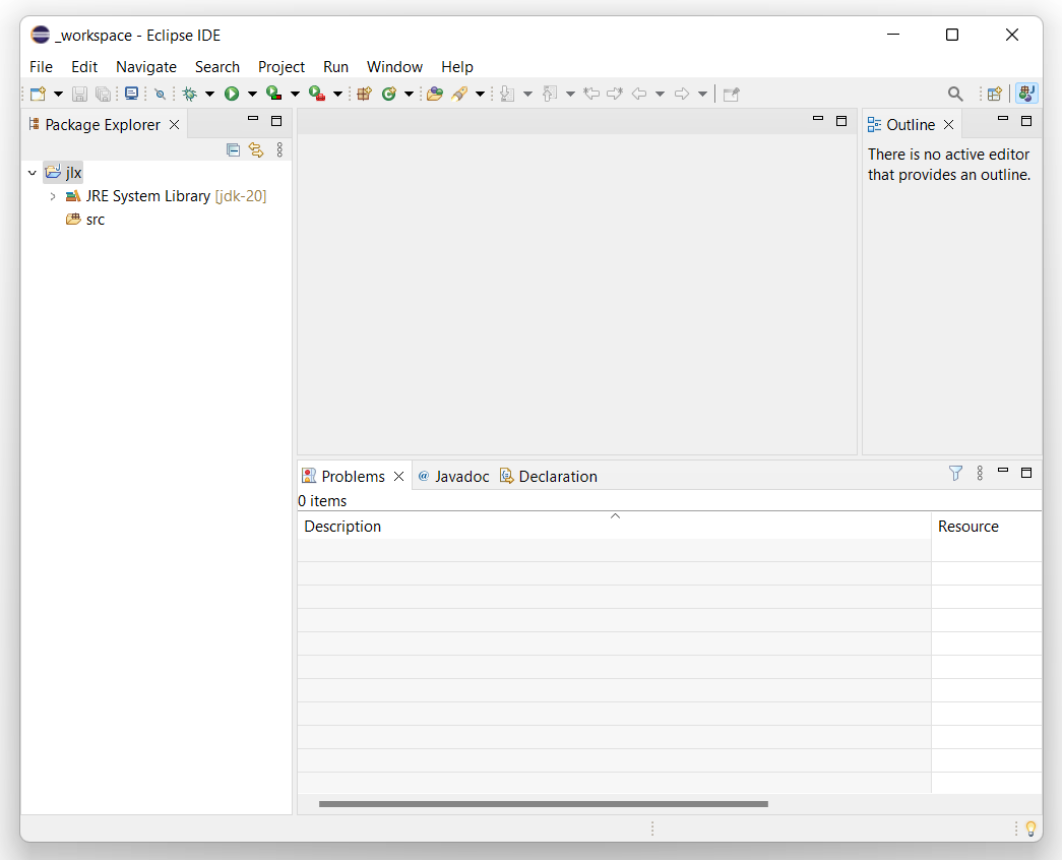
The window shows the ‘Package Explorer’ view on the left. Because you have no Java projects yet, it is currently empty.

4. Click the ‘Create a Java project’ action to add a project. A window for creating a new project pops up:

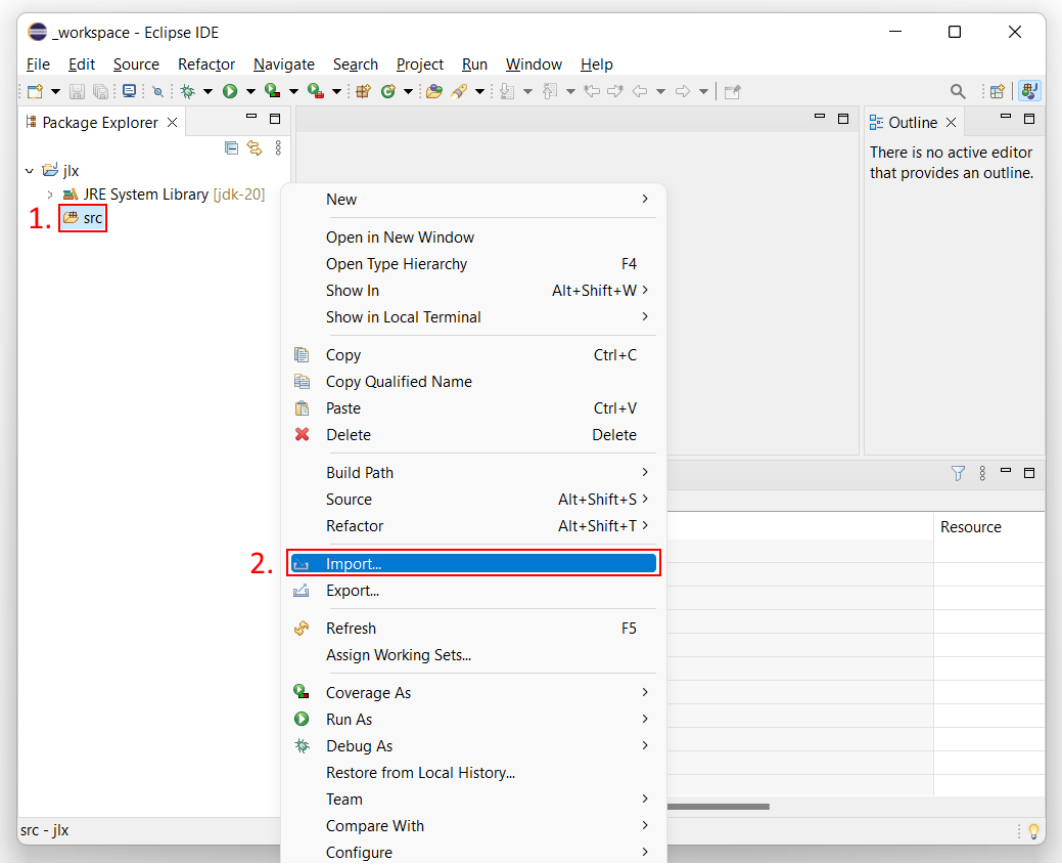


Set the name of the new project; set the Java Runtime Engine (the third option typically works, but not always); and **disable the automatic creation of a module file**.

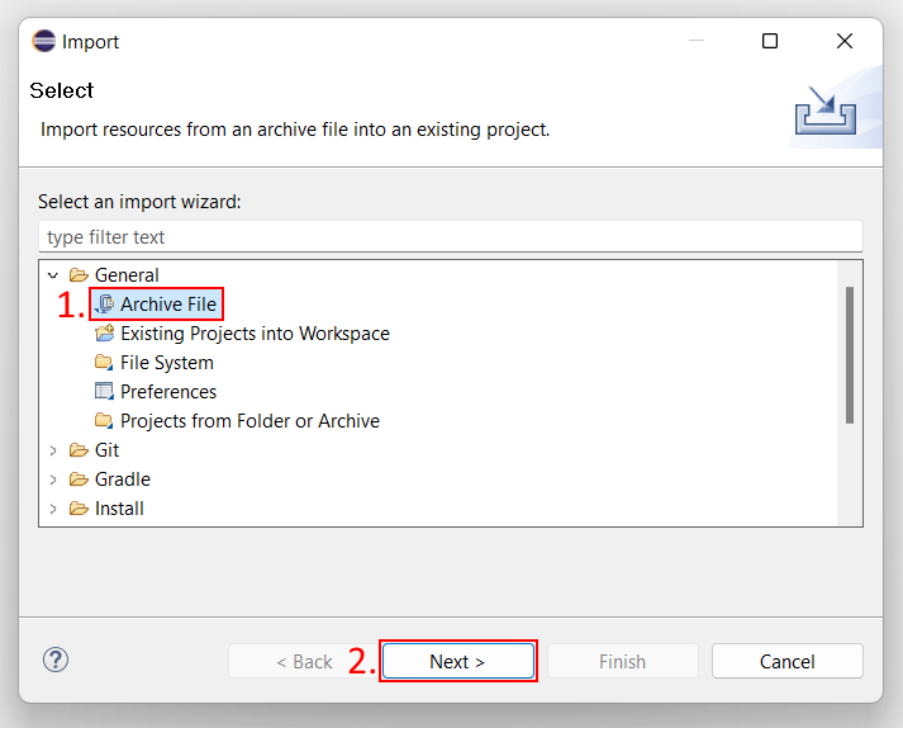
5. Click ‘Finish’. You should see the following window:



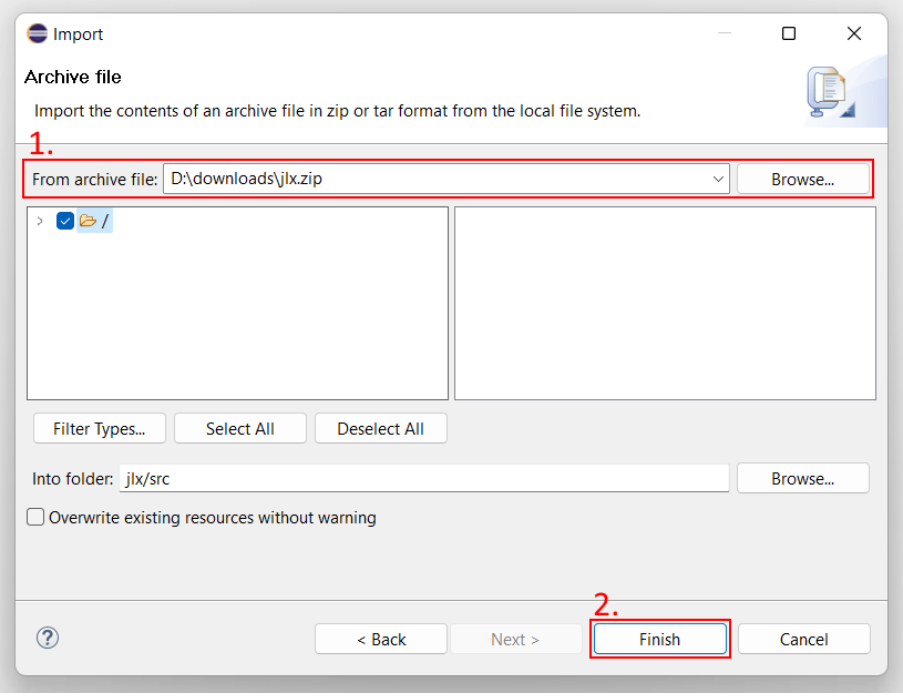
6. **Right-click the 'src' package** in the new project, and then left-click the 'Import...' option in the context menu:



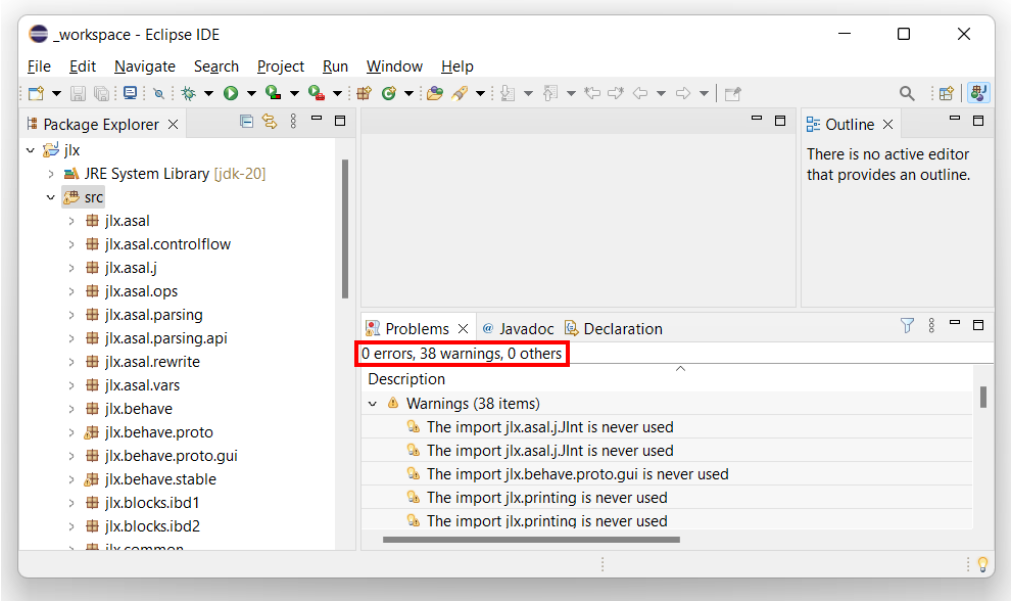
7. Choose to import resources from an ‘Archive File’, and then click ‘Next’:



8. Locate the archive from the ‘jlx’ repository, namely ‘jlx.zip’, and click ‘Finish’:



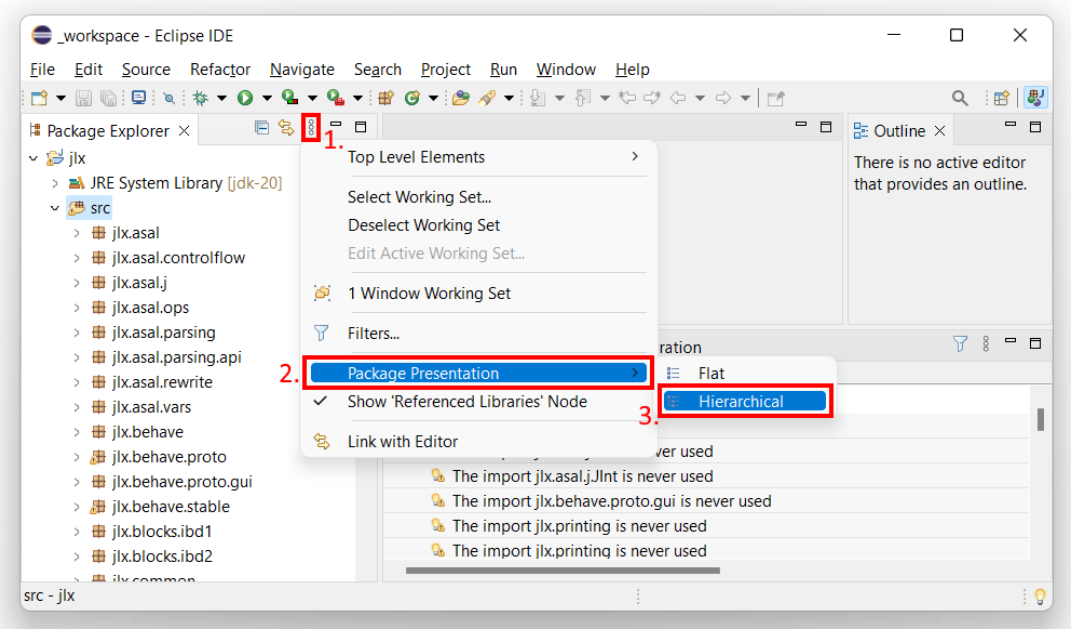
9. Check that there are no errors in the ‘Problems’ panel at the bottom (it is OK if there are warnings). If there are errors, please contact us:



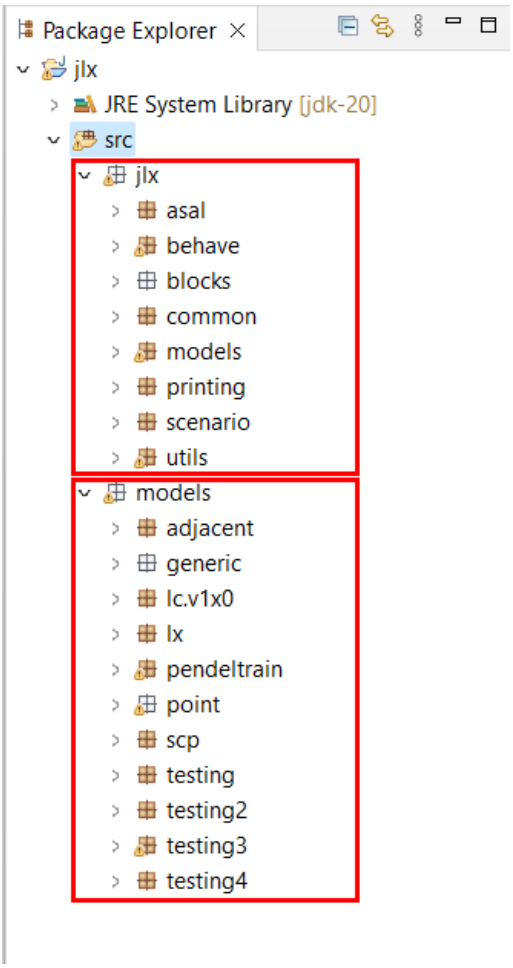
3 jEULYNX code base overview

3.1 Packages

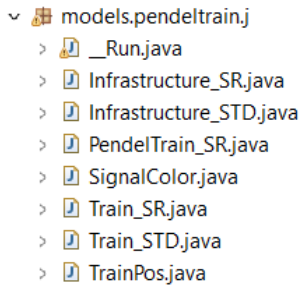
Java code is organized with *packages*. Packages are essentially directories: they can contain other packages, and/or files with Java code (i.e. files ending with '.java').
Packages are explored in Eclipse with the 'Package Explorer' (on the left-hand side). By default, they are displayed as "flat", meaning that the name of a package is combined with the name of its parent packages. For example, 'jlx.asal' refers to the 'asal' package, which is contained by the 'jlx' package.
Perhaps you prefer an "hierarchical" view of the packages. You can change these as depicted below:



In the hierarchical view, it becomes more clear that the jEULYNX code base is distributed over two root packages, ‘jlx’ and ‘models’. The ‘jlx’ root package contains the functional features of the jEULYNX software. The ‘models’ root package contains the diagrams of various EULYNX specifications (albeit in textual form).



Among others, ‘models’ contains the running example from a workshop, namely in the ‘pendeltrain.j’ sub-package:



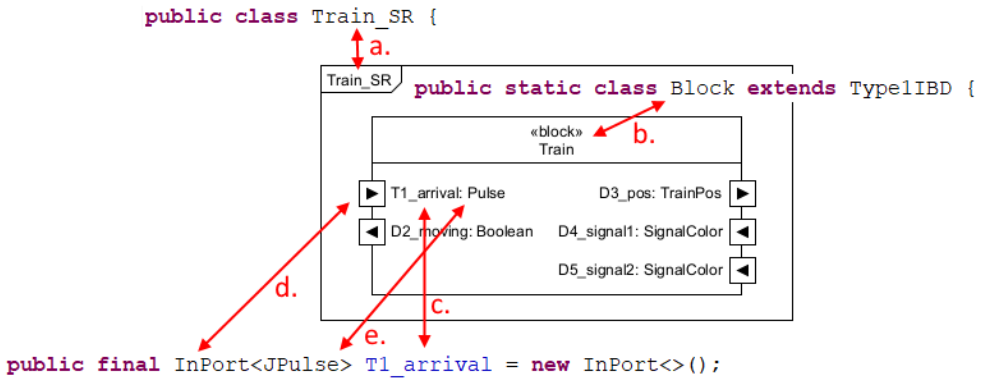
3.2 EULYNX diagram files

Open the file ‘models.pendeltrain.j.Train_SR.java’:

Train_SR.java

```
1 package models.pendeltrain.j;
2
3 import jlx.asal.j.*;
4 import jlx.blocks.ibd1.*;
5
6 public class Train_SR {
7 »   public static class Block extends Type1IBD {
8 »   »   public final InPort<JPulse> T1_arrival = new InPort<>();
9 »   »   public final OutPort<JBool> D2_moving = new OutPort<>();
10 »  »   public final OutPort<TrainPos> D3_pos = new OutPort<>();
11 »  »   public final InPort<SignalColor> D4_signal1 = new InPort<>();
12 »  »   public final InPort<SignalColor> D5_signal2 = new InPort<>();
13 »  }
14 }
15
16
```

It captures one of the internal block diagrams from the running example. Consider:



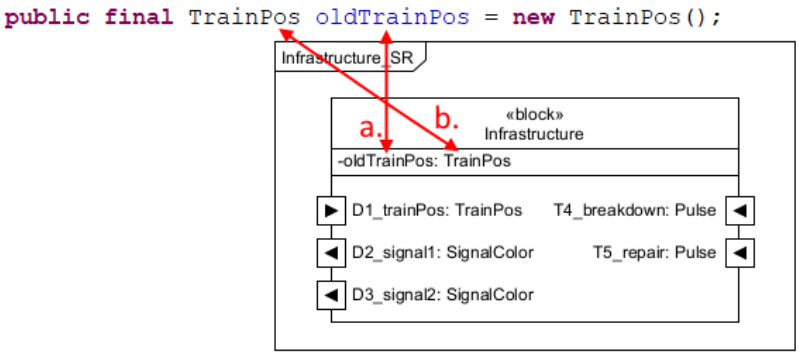
We use a class to capture the diagram “frame” (see a), and we use a nested class to capture the block inside of the diagram (see b). We use class fields to capture input ports and output ports. The name of the port is the same as the name of the class field (see c); the direction of the port is indicated by ‘InPort’ or ‘OutPort’ (see d); and the type of the port is the type parameter of ‘InPort’ or ‘OutPort’ (see e).

Open the file ‘models.pendeltrain.j.Infrastructure_SR.java’:

Infrastructure_SR.java

```
1 package models.pendeltrain.j;
2
3 import jlx.asal.j.*;
4 import jlx.blocks.ibd1.*;
5
6 public class Infrastructure_SR {
7     public static class Block extends Type1IBD {
8         public final TrainPos oldTrainPos = new TrainPos();
9         public final InPort<TrainPos> D1_trainPos = new InPort<>();
10        public final OutPort<SignalColor> D2_signal1 = new OutPort<>();
11        public final OutPort<SignalColor> D3_signal2 = new OutPort<>();
12        public final InPort<JPulse> T4_breakdown = new InPort<>();
13        public final InPort<JPulse> T5_repair = new InPort<>();
14    }
15 }
16
17
```

It includes a class field that represents a property, ‘oldTrainPos’, in addition to class fields that represent input and output ports:



The name of the class field is the same as the name of the property (see a). The type of the class field is the same as the type of the property (see b). The accessibility of the property (meaning that ‘oldTrainPos’ is only available to the instance of ‘Infrastructure’ that owns it) is not captured.

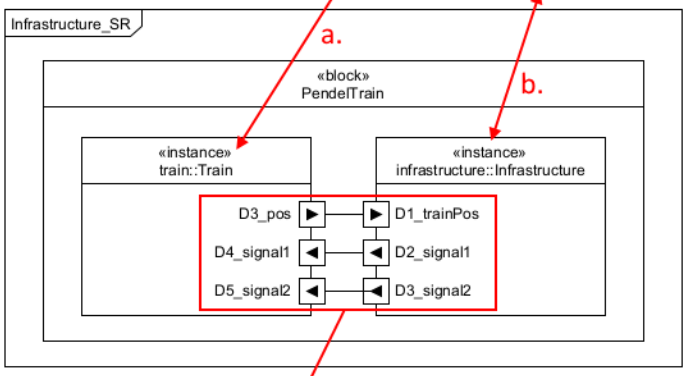
Open the file ‘models.pendeltrain.j.PendelTrain_SR.java’:

PendelTrain_SR.java

```
1 package models.pendeltrain.j;
2
3 import jlx.blocks.ibd2.*;
4
5 public class PendelTrain_SR {
6 »   public static class Block extends Type2IBD {
7 »   »   public final Train_SR.Block train = new Train_SR.Block();
8 »   »   public final Infrastructure_SR.Block infrastructure = new Infrastructure_SR.Block();
9 »   »
10 »  »   @Override
11 »  »   public void connectFlows() {
12 »  »   »   train.D3_pos.connect(infrastructure.D1_trainPos);
13 »  »   »   train.D4_signal1.connect(infrastructure.D2_signal1);
14 »  »   »   train.D5_signal2.connect(infrastructure.D3_signal2);
15 »  »   }
16 »  }
17 }
18
19
```

The file captures from the running example an internal block diagram that has *aggregate parts* (see a and b) that are connected by *flows* (see c):

```
public final Train_SR.Block train = new Train_SR.Block();
public final Infrastructure_SR.Block infrastructure = new Infrastructure_SR.Block();
```




```
@Override
public void connectFlows() {
»   train.D3_pos.connect(infrastructure.D1_trainPos);
»   train.D4_signal1.connect(infrastructure.D2_signal1);
»   train.D5_signal2.connect(infrastructure.D3_signal2);
}
```

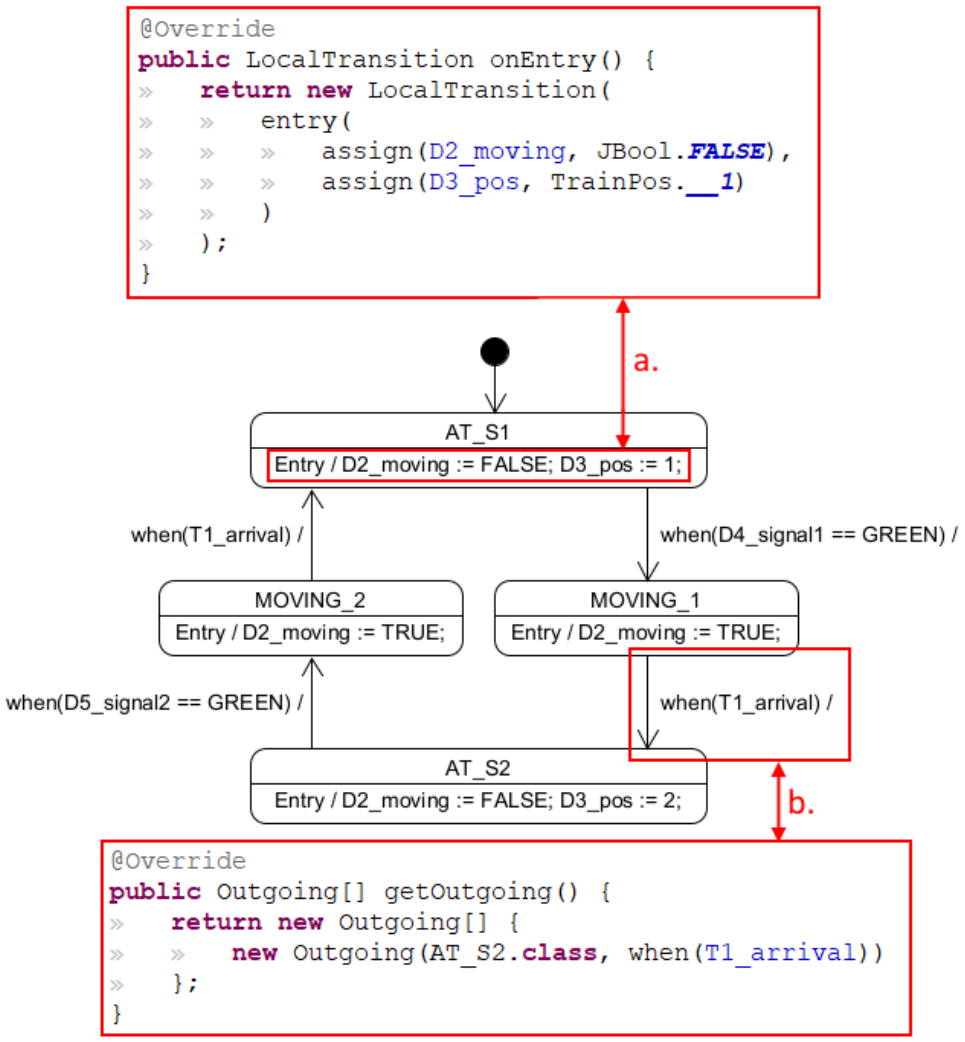
Open the file ‘models.pendeltrain.j.Train_STD.java’. When restricted to two of the nested classes in the file, namely ‘AT_S1’ and ‘MOVING_1’, it looks like:

Train_STD.java

```
1  package models.pendeltrain.j;
2
3  import jlx.asal.j.*;
4  import jlx.behave.*;
5
6  public class Train_STD extends Train_SR.Block implements StateMachine {
7      public class Initial0 extends InitialState {
15
16     public class AT_S1 extends State {
17         @Override
18         public LocalTransition onEntry() {
19             return new LocalTransition(
20                 entry(assign(D2_moving, JBool.FALSE), assign(D3_pos, TrainPos.__1))
21             );
22         }
23
24         @Override
25         public Outgoing[] getOutgoing() {
26             return new Outgoing[] {
27                 new Outgoing(MOVING_1.class, when(eq(D4_signal1, SignalColor.GREEN)))
28             };
29         }
30     }
31
32     public class AT_S2 extends State {
45
46     public class MOVING_1 extends State {
47         @Override
48         public LocalTransition onEntry() {
49             return new LocalTransition(entry(assign(D2_moving, JBool.TRUE)));
50         }
51
52         @Override
53         public Outgoing[] getOutgoing() {
54             return new Outgoing[] {
55                 new Outgoing(AT_S2.class, when(T1_arrival))
56             };
57         }
58     }
59
60     public class MOVING_2 extends State {
73 }
74
```

The outer class, ‘Train_STD’, represents a state machine diagrams. We can tell that it is a state machine diagram because it *implements* ‘StateMachine’ (see line 6). It *extends* ‘Train_SR.Block’, the class that represents the ‘Train’ block in the file ‘models.pendeltrain.j.Train_SR.java’. This gives the state machine diagram access to the properties, operations, and ports that ‘Train_SR.Block’ defines.
Each class represents a state (or pseudo-state, in case of ‘Initial0’). The *super-class* of a class denotes the type of the state that the class represents (e.g. ‘State’ and ‘InitialState’). Classes can be organized in a hierarchy, just like states.
Behaviors are attached to states with *methods*. For example, ‘onEntry’ methods define the entry behavior of a state; and ‘getOutgoing’ methods define the transitions (including behavior) that leave a state.

The file above captures one of the state machine diagrams from the running example. Below, we map two code fragments back to that diagram:



The entry behavior of the ‘AT_S1’ state is captured with an ‘onEntry’ method (see a); and the transition from the ‘MOVING_1’ state to the ‘AT_S2’ state is captured with an ‘getOutgoing’ method (see b).

3.3 __Run.java files

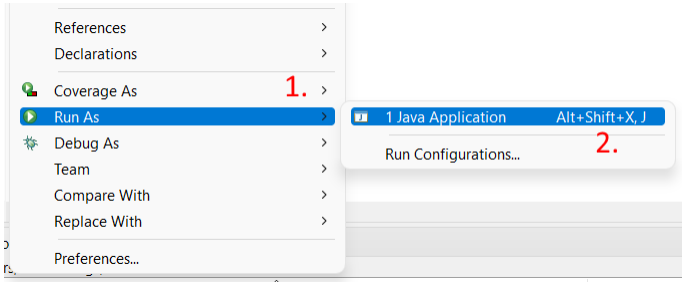
In jEULYNX software, diagrams must be combined in a separate file. Below, we find 'models.pendeltrain.j.__Run.java', which combines the diagrams of the running example:

```
_Run.java ×
 1 package models.pendeltrain.j;
 2
 3 import jlx.behave.proto.gui.DecaFourSimulatorGUI;
 9
10 public class __Run {
11     public static void main(String[] args) throws ReflectionException {
12         Model m = new Model();
13         m.add("pendelj", new PendelTrain_SR.Block());
14         m.add("train", new Train_STD());
15         m.add("infrastructure", new Infrastructure_STD());
16
17         UnifyingBlock ub = new UnifyingBlock("pendeltrain", m, false, false);
18
19         new MCRL2Printer(ub, new PrintingOptions()).printAndPop("pendeltrain");
20
21         new DecaFourSimulatorGUI(ub.sms4);
22     }
23 }
24
25
```

We give some brief descriptions below:

- Line 12 creates a 'Model' entity, which manages a collection of diagrams.
- Line 13 adds the 'PendelTrain_SR' diagram to the 'Model' entity. Indirectly, the 'Train_SR' and 'Infrastructure_SR' diagrams are also added.
- Line 14 and 15 add the state machine diagrams, 'Train_STD' and 'Infrastructure_STD', respectively.
- Line 17 transforms the diagrams that are now part of the 'Model' entity into an intermediate format.
- Line 19 generates an mCRL2 file from the intermediate format that is created in line 17.
- Line 21 starts a simulator from the intermediate format that is created in line 17.

To run the code, right-click in area with the code. A context menu appears:



Navigate to the 'Run As' sub-menu, open it, and click the 'Java Application' option.