

An Introduction to Metamodelling and Graph Transformations

with eMoflon



Part I: Installation and Setup

For eMoflon Version 2.16.0

File built on 21st September, 2016

Copyright © 2011–2016 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

For further information contact us at contact@emoflon.org.

The eMoflon team

Darmstadt, Germany (September 2016)

Contents

1	Getting started	1
2	Get a simple demo running	4
3	Validate your installation with JUnit	10
4	Project setup	11
5	Generated code vs. hand-written code	19
6	Conclusion and next steps	20

Part I:

Installation and Setup

This part provides a very simple example and a JUnit test to check the installation and configuration of eMoflon. It can be considered *mandatory* if you are new to eMoflon, but we recommend working through it anyway.

After working through this part, you should have an installed and tested eMoflon working for a trivial example. We also explain the general workflow, the different workspaces involved.

URL of this document: <https://emoflon.github.io/eclipse-plugin/beta/handbook/part1.pdf>

1 Getting started

If, however, you're finding that the screenshots we've taken aren't matching your screen and you *ARE* in the right place, please send us an email at contact@emoflon.org and let us know. They get outdated so fast! They just grow up, move on, start doing their own thing and ...uh, wait a second. We're talking about pictures here.

If you have problems while using Moflon you can also have a look at our FAQs: <https://github.com/eMoflon/emoflon-docu/wiki/eMoflon-FAQ>.

1.1 Install our plugin for Eclipse

- ▶ Make sure that you have **Java 1.8** installed.
- ▶ Download and install Eclipse Mars for modelling, which is called “**Eclipse Modeling Tools**” from <http://www.eclipse.org/downloads/index.php> (Figure 1.1).¹
- ▶ Install our Eclipse Plugin from the following update site²:

¹Please note that you *have to* install Eclipse Modelling Tools, or else some features won't work! Although different versions may support eMoflon, our tool is currently tested only for Mars and Java 1.8.

²For a detailed tutorial on how to install Eclipse and Eclipse Plugins please refer to <http://www.vogella.de/articles/Eclipse/article.html>

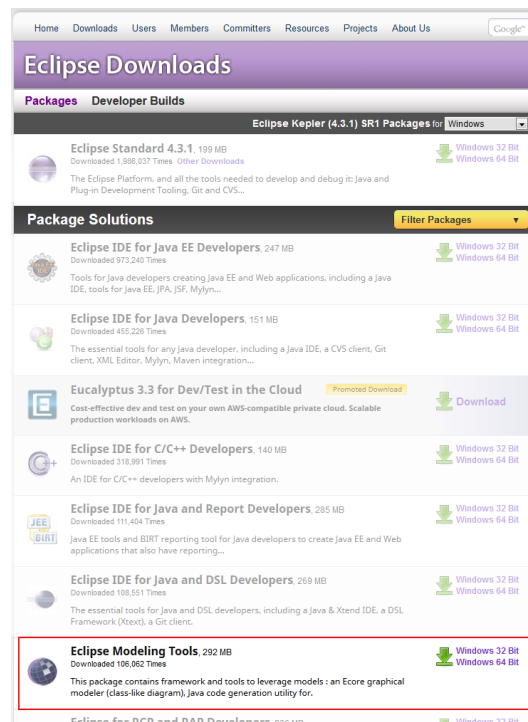


Figure 1.1: Download **Eclipse Modeling Tools**

<https://emoflon.github.io/eclipse-plugin/beta/update-site2/>

Please note: Calculating requirements and dependencies when installing the plugin might take quite a while depending on your internet connection.

Hint: To inform you about new updates, we provide some mailing lists for you: <http://www.emoflon.org/emoflon/mailling-lists/>

1.2 Install our extension for Enterprise Architect

Enterprise Architect (EA) is a visual modelling tool that supports UML³ and a host of other modelling languages. EA is not only affordable but also quite flexible, and can be extended via *extensions* to support new modelling tools – such as eMoflon!

- Download EA for Windows from <http://www.sparxsystems.com/> to get a free 30 day trial and follow installation instructions (Figure 1.2).



Figure 1.2: Download Enterprise Architect

- Install our EA extension (Figure 1.3) to add support for our modelling languages. Download <https://emoflon.github.io/eclipse-plugin/beta/update-site2/ea-ecore-addin.zip>, unpack, and run `eMoflonAddinInstaller.msi`.

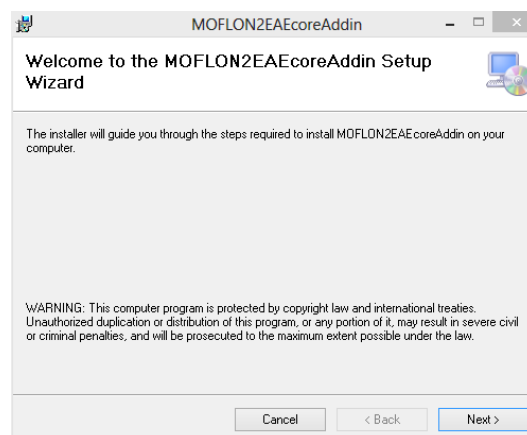


Figure 1.3: Install our extension for EA

³Unified Modelling Language

2 Get a simple demo running

- Open Eclipse to a clean, fresh workspace. Go to *Window* → *Open Perspective* → *Other...*⁴ and choose **eMoflon** (Figure 2.1).

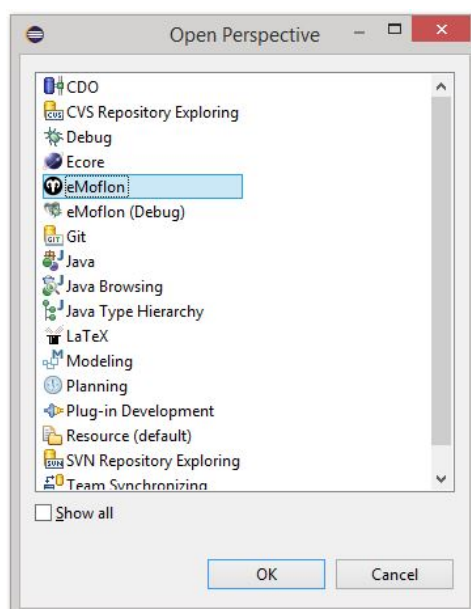


Figure 2.1: Choose the eMoflon perspective

- At either the far right or center of the toolbar, a new action set should have appeared. Navigate to “*eMoflon Cloud*” → *Install Workspace* (Figure 2.2).
- In this menu you can check out different workspaces for Eclipse. Here you can also check out workspaces for the handbook tutorials (*eMoflon Examples*). How you can create your own projects is described in Part I. As the first tutorial select *eMoflon Examples* → *Demo (Double-Linked List)*.

All our handbook examples are provided via Git and are hosted on GitHub⁵. If you encounter problems when fetching some handbook example, the reason may be that your checked out working copy is in a “dirty” state. In this case, it is safe to remove your whole working copy as follows: Navigate to the *Git* perspective in Eclipse (*Window*

⁴A path given as *foo* → *bar* indicates how to navigate in a series of menus and toolbars. New definitions or concepts will be *italicized*, and any data you’re required to enter, open, or select will be given as **command**.

⁵See <https://github.com/eMoflon/emoflon-examples>.

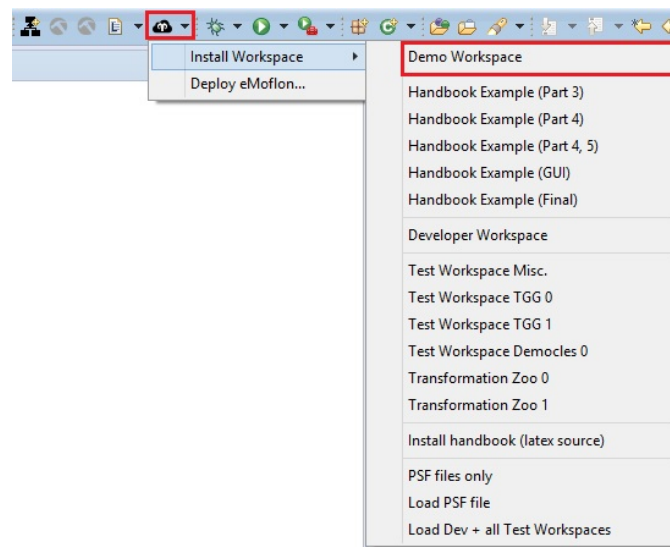


Figure 2.2: Invoking “Install Workspace → eMoflon Examples → Demo (Double-Linked List)”

→*Perspective* →*Open Perspective* →*Other...*), right-click your working copy `emoflon-examples` and choose *Delete Repository...*. Make sure to fully delete the repository by ticking all boxes.

- Another button in the new action set is *View and configure logging* represented by an L (Figure 2.3). Clicking this icon will open a `log4jConfig.properties` file where you can silence certain loggers, set the level of loggers, or configure other settings.⁶ All of eMoflon’s messages appear in our console window, just below your main editor. This is automatically opened when you selected the `eMoflon` perspective and contains important information for us if something goes wrong!

⁶If you’re not sure how to do this, check out a short Log4j tutorial a <http://logging.apache.org/log4j/1.2/manual.html>

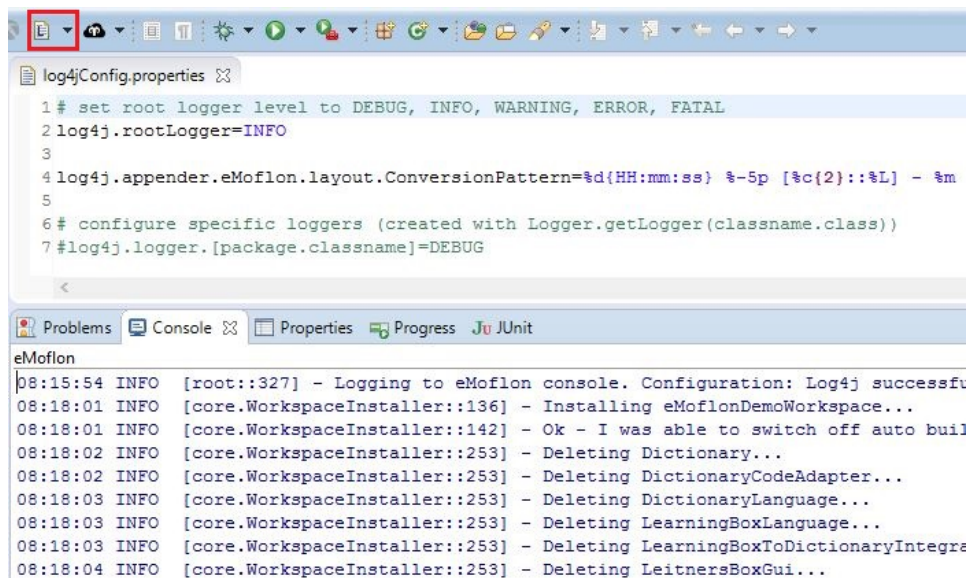


Figure 2.3: The eMoflon console with log messages

2.1 A first look at EA

- Can you locate the new `Demo.eap` file in your package explorer? This is the EA project file you'll be modelling in. Don't worry about any other folders at the moment - all problems will be resolved by the end of this section.

In the meantime, do not rename, move, or delete anything.

- Double-click `Demo.eap` to start EA, and choose **Ultimate** when starting EA for the first time.
- In EA, select *Extensions* → *Add-In Windows* (Figure 2.4). This will activate our tool's full control panel. If nothing happens the installation was probably not successful. Work again through the installation or have a look at the following site:
<https://github.com/eMoflon/emoflon-docu/wiki/Howto-on-Using-EA-With-A-Second-Windows-Account>.

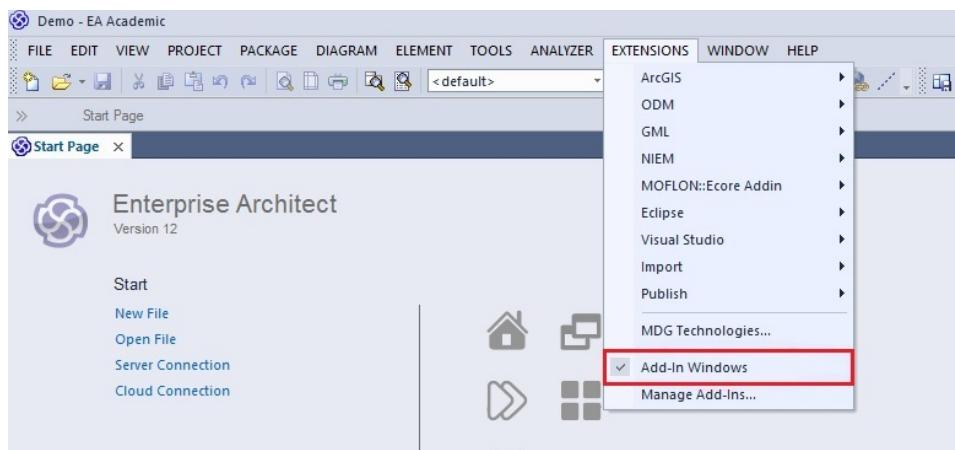


Figure 2.4: Export from EA

- This tabbed control panel provides access to all of eMoflon's functionality. This is where you can validate and export your complete project to Eclipse by pressing **A11** (Figure 2.5).
- Now try exploring the EA project browser! Try to navigate to the packages, classes, and diagrams. Don't worry if you don't understand that much—we'll get to explaining everything in a moment. Just make sure not to change anything!
- Switch back to Eclipse, choose your metamodel project, and press **F5** to refresh. The export from EA places all required files in a hidden folder (`.temp`) in the project. A new, third project named

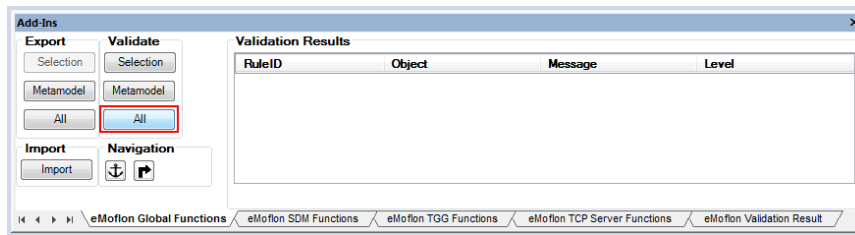


Figure 2.5: eMoflon's control panel in EA

`org.moflon.demo.doublelinkedlist` is now being created. Do not worry about the problem markers.

- The three asterisks signal that the project still needs to be built (Figure 2.6).

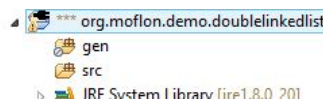


Figure 2.6: Dirty projects are marked with ***

- Now, right-click `org.moflon.demo.doublelinkedlist` and choose *eMoflon* → *Build* (or use the shortcut `Alt+Shift+E,B`⁷).

eMoflon now generates the Java code in your repository project. You should be able to monitor the progress with the green bar in the lower right corner (Figure 2.7). Pressing the symbol opens a monitor view that gives more details of the build process. You don't need to worry about any of these details, just remember to (i.) refresh your Eclipse workspace after an export, and (ii.) rebuild projects that bear a “dirty marker” (***).

- If you're ever worried about forgetting to refresh your workspace, or if you just don't want to bother with having to do this, Eclipse does offer an option to do it for you automatically. To activate this, go to *Window* → *Preferences* → *General* → *Workspace* and select **Refresh on access**.

⁷First press `Alt+Shift+E`, release, and press `B`. By default, most shortcuts eMoflon start with `Alt+Shift+E`.

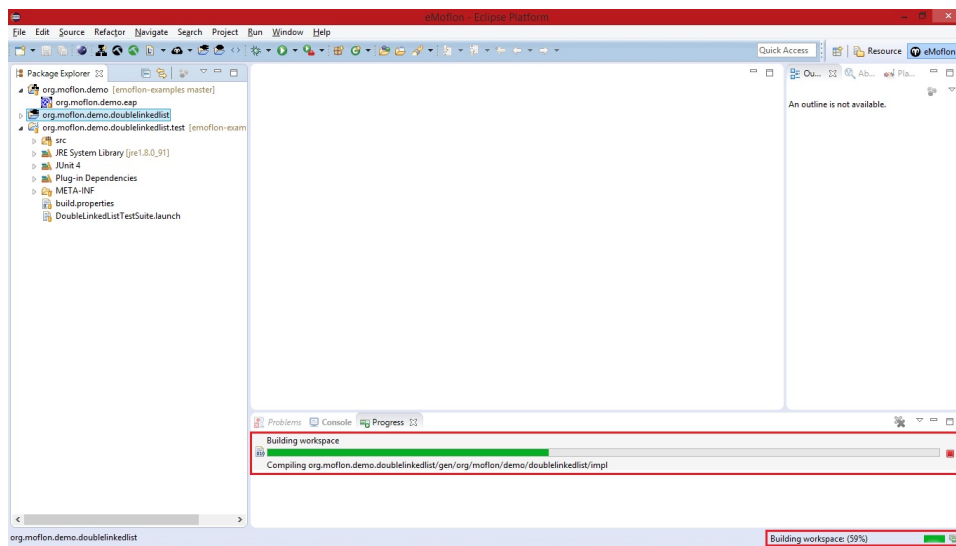


Figure 2.7: Eclipse workspace while building the demo project

3 Validate your installation with JUnit

- In Eclipse, choose **Working Sets** as your top level element in the package explorer (Figure 3.1), as we use them to structure the workspace.

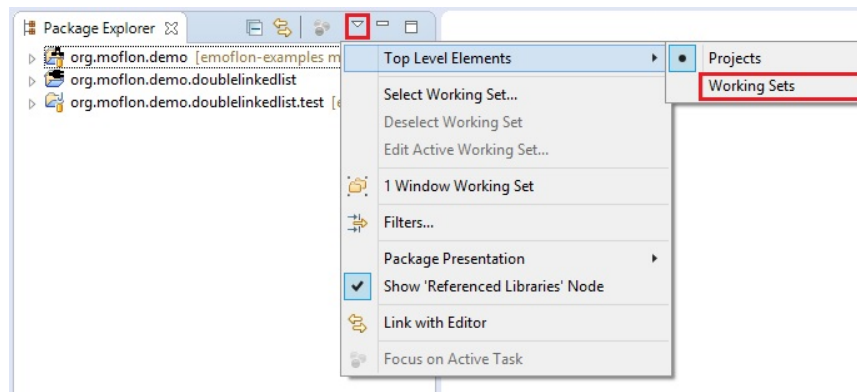


Figure 3.1: Top level elements in Eclipse

- Locate *Other Projects* → *org.moflon.demo.doublelinkedlist.test*. This is the testsuite imported with the demo files to make sure everything has been installed and set up correctly. Right click on the project to bring up the context menu and go to *Run As* → *JUnit Test*. If anything goes wrong, try refreshing by choosing your metamodel project and pressing **F5**, or right-clicking and selecting **Refresh**.

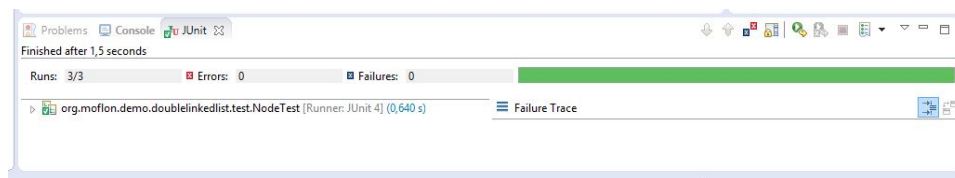


Figure 3.2: All's well that ends well...

Congratulations! If you see a green bar (Figure 3.2), then everything has been set up correctly and you are now ready to start metamodeling!

4 Project setup

4.1 Your Enterprise Architect Workspace

Now that everything is installed and setup properly, let's take a closer look at the different workspaces and our workflow. Before we continue, please make a few slight adjustments to Enterprise Architect (EA) so you can easily compare your current workspace to our screenshots. These settings are advisable but you are, of course, free to choose your own colour schema.

- Select *Tools* → *Options* → *Themes* in EA, and set Diagram Theme to **Enterprise Architect 10**.
- Next, proceed to *Gradients and Background* and set *Gradient* and *Fill* to **White** (Figure 4.1).

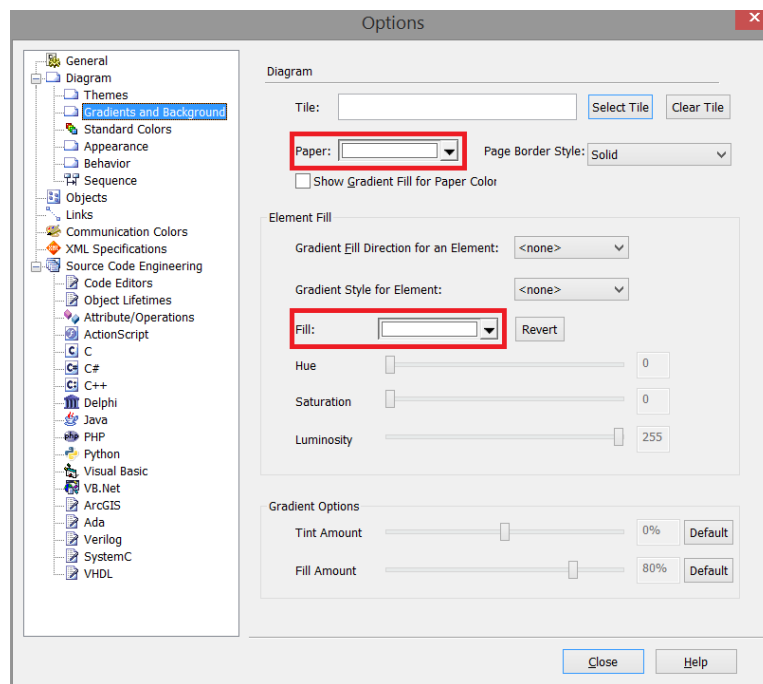


Figure 4.1: Suggested paper background and element fill

- In the *Standard Colors* tab, and set your colours to reflect Figure 4.2.
- In the same dialogue, go to *Diagram* → *Appearance* and reflect the settings in Figure 4.3. Again, this is just a suggestion and not mandatory.

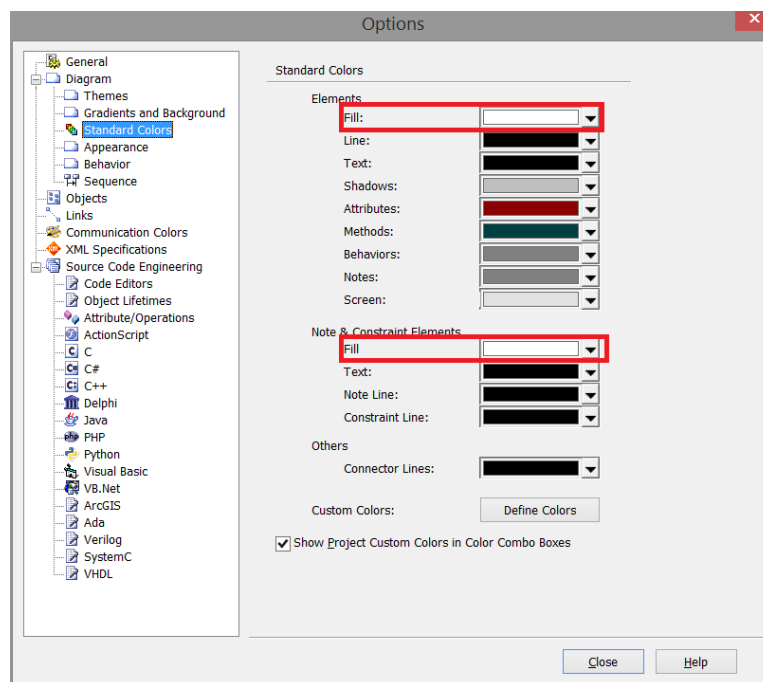


Figure 4.2: Our choice of standard colours for diagrams in EA

- Last but not least open the *Code Engineering* toolbar (Figure 4.4) and choose **Ecore** as the default language (Figure 4.5). **This setting is mandatory, and very important.**

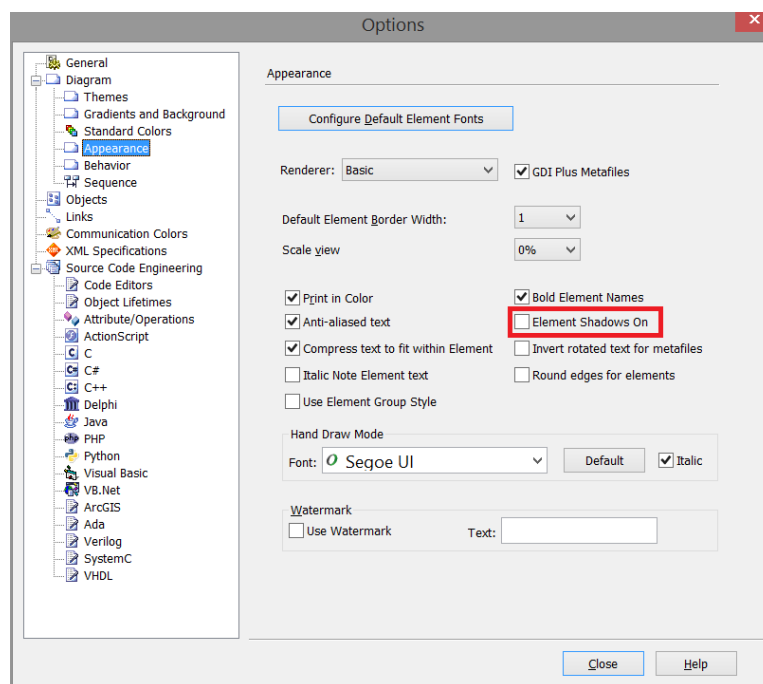


Figure 4.3: Our choice of the standard appearance for model elements

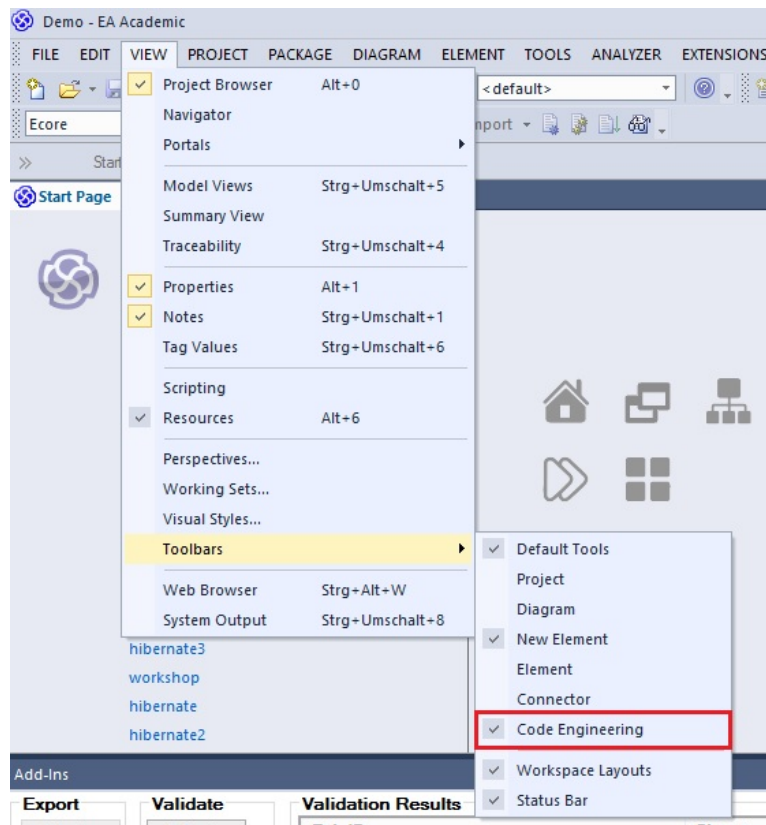


Figure 4.4: Open the *Code Engineering* toolbar

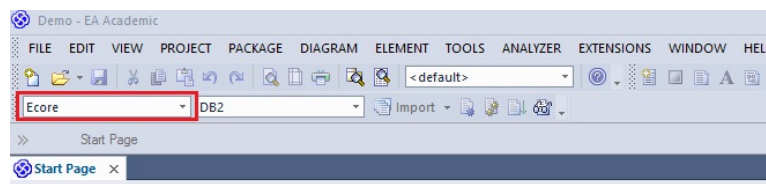


Figure 4.5: Make sure you set the standard language to Ecore

In your EA “workspace” (actually referred to as an *EA project*), take a careful look at the project browser: The root node **Demo** is called a *model* in EA lingo, and is used as a container to group a set of related *packages*. In our case, **Demo** contains a single package `org.moflon.demo.doublelinkedlist`. An EA project however, can consist of numerous models that in turn, group numerous packages.

Now switch back to your Eclipse workspace and note the two nodes named **Specifications** and `org.moflon.demo` (Figure 4.6).

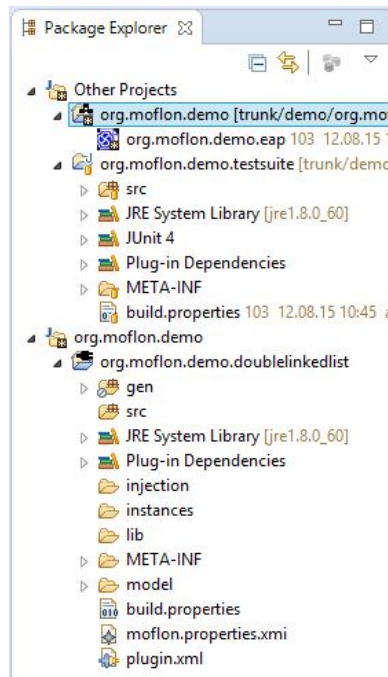


Figure 4.6: Project structure

These nodes, used to group related *Eclipse projects* in an Eclipse workspace, are called *working sets*. The working set **Specifications** contains all *metamodel projects* in a workspace. Your metamodel project contains a single EAP (EA project) file and is used to communicate with EA and initiate code generation by simply pressing F5 or choosing **Refresh** from the context menu. In our case, **Specifications** should contain a single metamodel project `org.moflon.demo` containing our EA project file `org.moflon.demo.eap`.

Figure 4.7 depicts how the Eclipse working set `org.moflon.demo` and its contents were generated from the EA model `org.moflon.demo`. Every model in EA is mapped to a working set in Eclipse with the same name. From every package in the EA model, an Eclipse project is generated, also with

the same name.

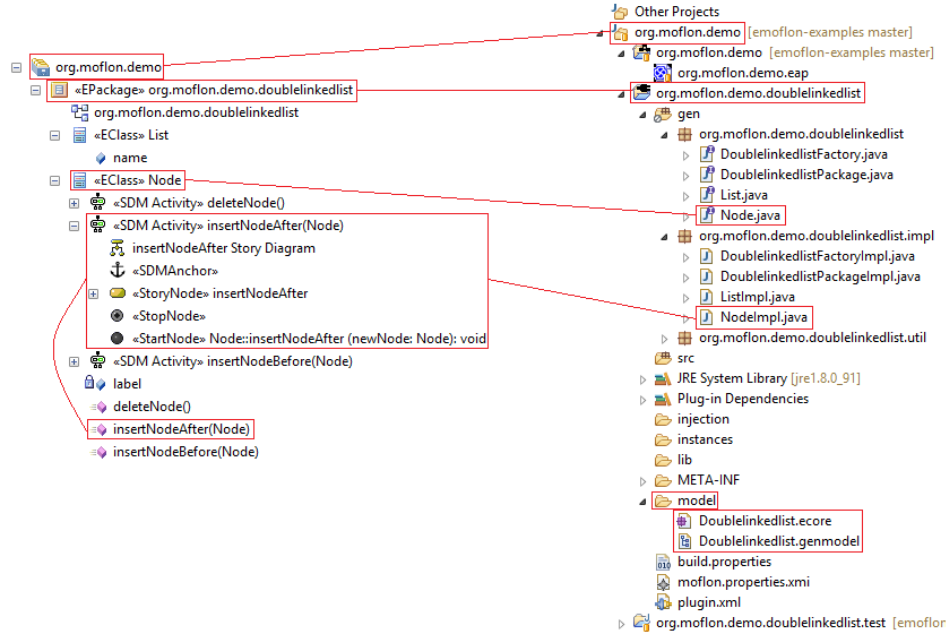


Figure 4.7: Mapping between artefacts in EA and Eclipse

These projects, however, are of a different nature than, for example, meta-model projects or normal Java projects. These are called *repository projects*. A *nature* is Eclipse lingo for “project type” and is visually indicated by a corresponding nature icon on the project folder. Our metamodel projects sport a neat little class diagram symbol. Repository projects are generated automatically with a certain project structure according to our conventions.

The `model` subfolder in the Eclipse package explorer is probably the most important as it contains the *Ecore model* for the project. Ecore is a metamodeling language that provides building blocks such as *classes* and *references* for defining the static structure (concepts and relations between concepts) of a system. This folder also contains a *genmodel*, the second model required by the Eclipse Modeling Framework (EMF) to generate Java code.

Looking back to Figure 4.7, realize that it also depicts how the class `Node` in the EA model is mapped to the Java interface `Node`. Double-click `Node.java` and take a look at the methods declared in the interface. These correspond directly to the methods declared in the modeled `Node` class.

As indicated by the source folders `src`, `injection`, and `gen`, we advocate a clean separation of hand-written (should be placed in `src` and `injection`) and generated code (automatically in `gen`). As we shall see later in the handbook, hand-written code can be integrated in generated classes via

injections. This is sometimes necessary for small helper functions.

Have you noticed the methods of the `Node` class in our EA model? Now hold on tight—each method can be *modeled* completely in EA and the corresponding implementation in Java is generated automatically and placed in `NodeImpl.java`. Just in case you didn’t get it: The behavioural or dynamic aspects of a system can be completely modeled in an abstract, platform-/programming language-independent fashion using a blend of activity diagrams and a “graph pattern” language called *Story Driven Modelling* (SDM). In our EA project, these *Story Diagrams* or simply *SDMs*, are placed in *SDM Containers* named according to the method they implement. For instance, «SDM Activity» `insertNodeAfter` SDM for the method `insertNodeAfter(Node)` as depicted in Figure 4.7. We’ll dedicate Part III of the handbook to understanding why SDMs are so **crazily** cool!

To recap all we’ve discussed, let’s consider the complete workflow as depicted in Figure 4.8. We started with a concise model in EA, simple and independent of any platform specific details (1). Our EA model consists not only of static aspects modelled as a class diagram (2), but also of dynamic aspects modelled using SDM (3). After exporting the model and code generation (4), we basically switch from *modelling* to *programming* in a specific general purpose programming language (Java). On this lower *level of abstraction*, we can flesh out the generated repository (5) if necessary, and mix as appropriate with hand-written code and libraries. Our abstract specification of behaviour (methods) in SDM is translated to a series of method calls that form the body of the corresponding Java method (6).

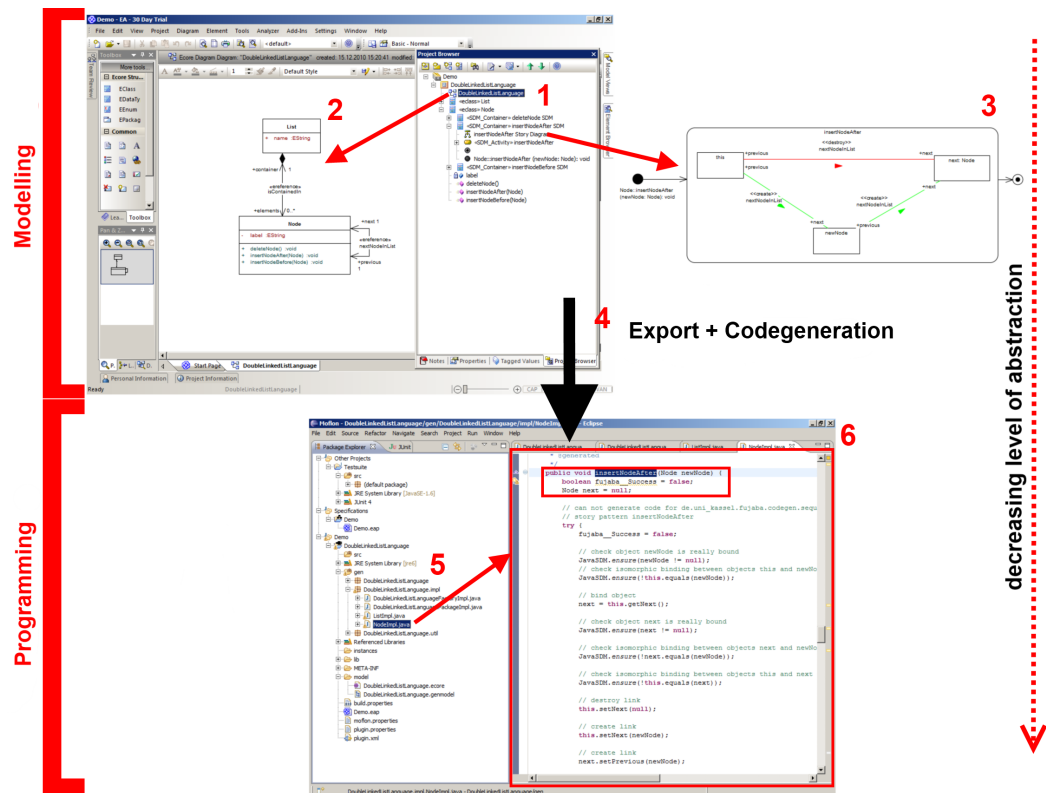


Figure 4.8: Overview

5 Generated code vs. hand-written code

Now that you've worked through the specifics of your syntax, let's have a brief discussion on code generation.

The Ecore model is used to drive a code generator that maps the model to Java interfaces and classes. The generated Java code that represents the model is often referred to as a repository. This is the reason why we refer to such projects as repository projects. A repository can be viewed as an adapter that enables building and manipulating concrete instances of a specific model via a programming language such as Java. This is why we indicate repository projects using a cute adapter/plug symbol on the project folder.

If you take a careful look at the code structure in `gen` (Figure 5.1), you'll find a `FooImpl.java` for every `Foo.java`. Indeed, the subpackage `.impl` contains Java classes that implement the interfaces in the parent package. Although this might strike you as unnecessary (why not merge interface and implementation for simple classes?), this consequent separation in interfaces and implementation allows for a clean and relatively simple mapping of Ecore to Java, even in tricky cases such as multiple inheritance (allowed and very common in Ecore models). A further package `.util` contains some auxiliary classes such as a factory for creating instances of the model.

If this is your first time of seeing generated code, you might be shocked at the sheer amount of classes and code generated from our relatively simple model. You might be thinking: "Hey – if I did this by hand, I wouldn't need half of all this stuff!" Well, you're right and you're wrong. The point is that an automatic mapping to Java via a code generator scales quite well.

This means for simple, trivial examples (like our double linked list), it might be possible to come up with a leaner and simpler Java representation. For complex, large models with lots of mean pitfalls however, this becomes a daunting task. The code generator provides you with years and *years* of experience of professional programmers who have thought up clever ways of handling multiple inheritance, an efficient event mechanism, reflection, consistency between bidirectionally linked objects, and much more.

A point to note here is that the mapping to Java is obviously not unique. Indeed there exist different standards of how to map a modelling language to a general purpose programming language such as Java. As previously mentioned, we use a mapping defined and implemented by the Eclipse Modelling Framework (EMF), which tends to favour efficiency and simplicity over expressiveness and advanced features.

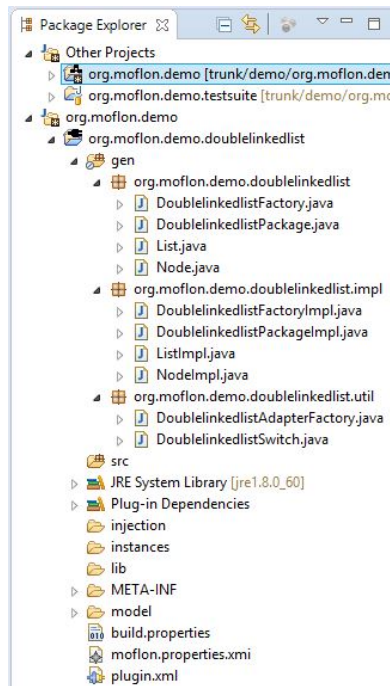


Figure 5.1: Package structure of generated code (**gen**)

6 Conclusion and next steps

Congratulations – you’ve finished Part I! If you feel a bit lost at the moment, please be patient. This first part of the handbook has been a lot about installation and tool support, and only aims to give a very brief glimpse at the big picture of what is actually going on.

If you enjoyed this section and wish to get started on the key features of eMoflon, Check out Part II⁸! There we will work through a hands-on, step-by-step example and cover the core features of eMoflon.

We shall also introduce clear and simple definitions for the most important metamodeling and graph transformation concepts, always referring to the concrete example and providing references for further reading.

If you’re already familiar with the tool, feel free to pick and choose individual parts that are most interesting to you. Check out Story Driven Modeling (SDMs) in Part III⁹, or Triple Graph Grammars (TGGs) in Part IV¹⁰. We’ll provide instructions on how to easily download all the required resources so

⁸Download: <https://emoflon.github.io/eclipse-plugin/beta/handbook/part2.pdf>

⁹Download: <https://emoflon.github.io/eclipse-plugin/beta/handbook/part3.pdf>

¹⁰Download: <https://emoflon.github.io/eclipse-plugin/beta/handbook/part4.pdf>

you can jump right in. For further details on each part, refer to Part 0¹¹.

Cheers!

¹¹Download: <https://emoflon.github.io/eclipse-plugin/beta/handbook/part0.pdf>