# An Introduction to Metamodelling and Graph Transformations

*with eMoflon*


## Part I: Installation and Setup

For eMoflon Version 2.2.1

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

*The eMoflon team*
Darmstadt, Germany (September 2015)

# Contents

# 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, and general usage of both our visual and textual syntax.

`Approximate time to complete:  Just a few minutes...`

`URL of this document:  http://tiny.cc/emoflon-rel-handbook/part1.pdf`

## 1 Getting started

Here's how we've organized our handbooks; Black, red, and blue headers are used to separate common, visual, and textual syntax instructions (Fig 1.1).

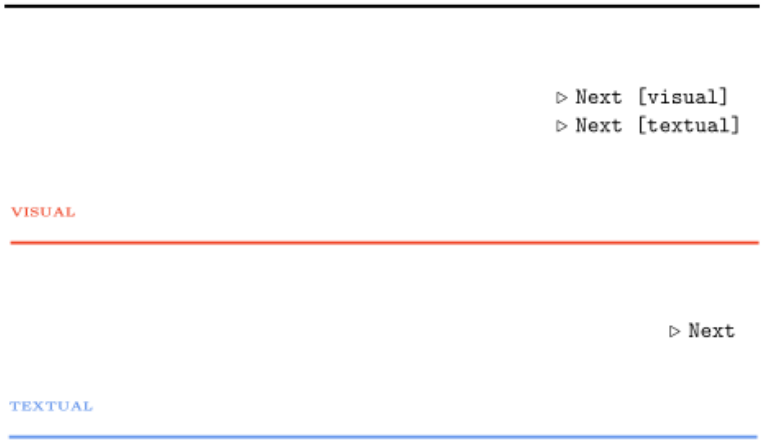


Figure 1.1: Page headers and links

You'll find a ▷ `link` at the bottom of some pages. These will take you to the next appropriate place for your syntax. You are still welcome to go through

the entire handbook page by page. In fact, we encourage it and hope you'll compare the differences and similarities between the two specifications. But be warned! If what you're doing isn't matching what you see, you may be reading the wrong instructions.

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 They get outdated so fast!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.

Here are some guidelines to help you decide which syntax to use:

- If you have used a UML tool before and feel comfortable with the standard UML diagrams, then you might prefer our visual syntax.
- If you do not like switching tools, and want everything integrated completely in Eclipse with zero installation, then you should stick to our textual syntax.
- If you use a Mac (or some other *Nix system) then you will need virtualisation software to run Windows and install the required UML tool for our visual syntax. Most maconians find this insulting and of course choose our textual syntax.
- As a final remark, consider that graph transformations obviously have something to do with graphs, which are inherently two dimensional structures. A visual syntax thus has some obvious advantages and is what we (currently) prefer and use internally (eMoflon is built with eMoflon).

## 1.1 Install our plugin for Eclipse

- Make sure that you have **Java 1.8** installed.
- Download and install Eclipse Marsfor modelling, which is called "**Eclipse Modeling Tools**" from `http://www.eclipse.org/downloads/index.php`.[1] (Fig. 1.2).

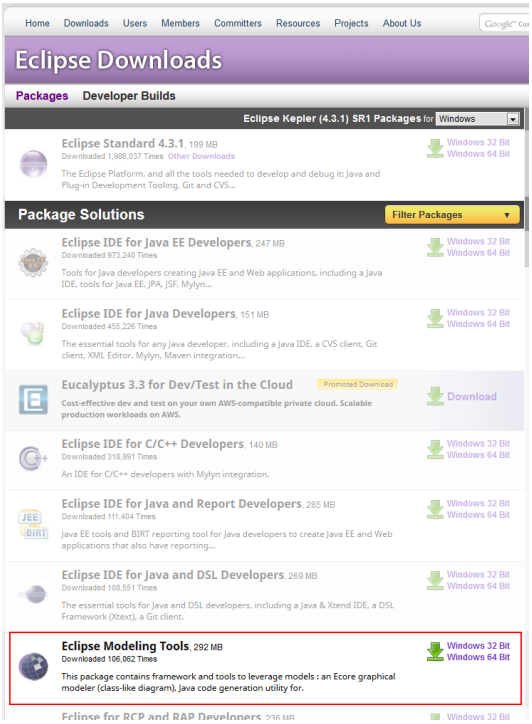


Figure 1.2: Download **Eclipse Modeling Tools**

- Install our Eclipse Plugin from the following update site[2]: `http://tiny.cc/emoflon-rel-update-site`

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

[1] Please 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.

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

▷ Next [visual]
▷ Next [textual]

## 1.2 Install our extension for Enterprise Architect

Enterprise Architect (EA) is a visual modelling tool that supports UML[3] 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 (Fig. 1.3).


Figure 1.3: Download Enterprise Architect

- Install our EA extension (Fig. 1.4) to add support for our modelling languages. Download `http://tiny.cc/emoflon-rel-eaaddin`, unpack, and run `eMoflonAddinInstaller.msi`.

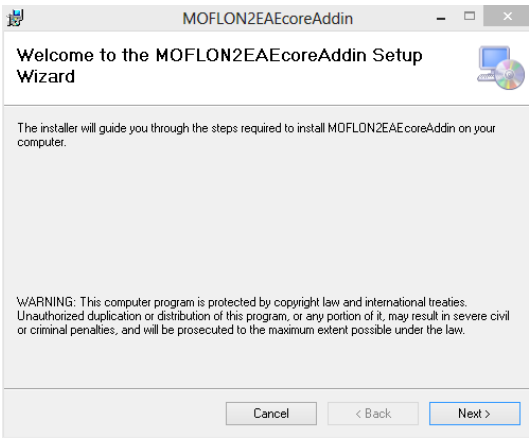


Figure 1.4: Install our extension for EA

[3]Unified 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` (Fig. 2.1).

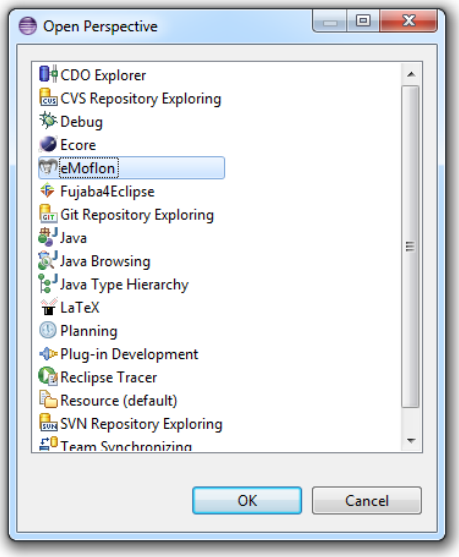


Figure 2.1: Choose the eMoflon perspective

- At either the far right or centre of the toolbar, a new action set should have appeared. Navigate to "New Metamodel" (Fig. 2.2).

Figure 2.2: Invoke the "New Metamodel" wizard

[4] A 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`.

- A new dialogue should appear. This is the place where you officially decide which eMoflon syntax you'd like to use (Fig. 2.3).

  With the visual syntax, you'll be using Eclipse along with a UML tool, Enterprise Architect (EA). You'll use EA to specify several types of diagrams, then export and refresh your workspace in Eclipse to generate the corresponding Java code.

  With the textual syntax (MOSL),[5] you'll be working entirely within the Eclipse IDE.

  No matter which syntax you choose however, give your project a name and make sure the `Add Demo Specification` button is selected. This will create all the files and tests required to ensure you've set up eMoflon correctly.

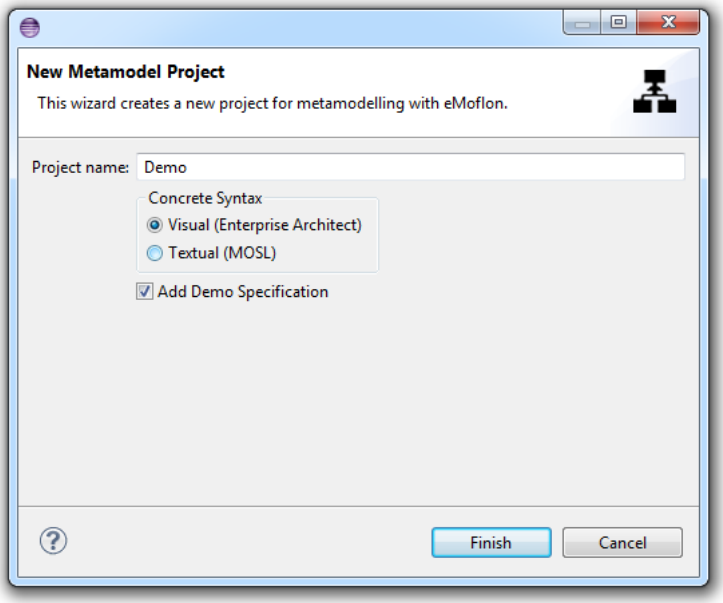


Figure 2.3: Choose your syntax

- Another button in the new action set is "View and configure logging" represented by an `L` (Fig. 2.4). 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`

[5] Moflon Specification Language

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

perspective and contains important information for us if something goes wrong!

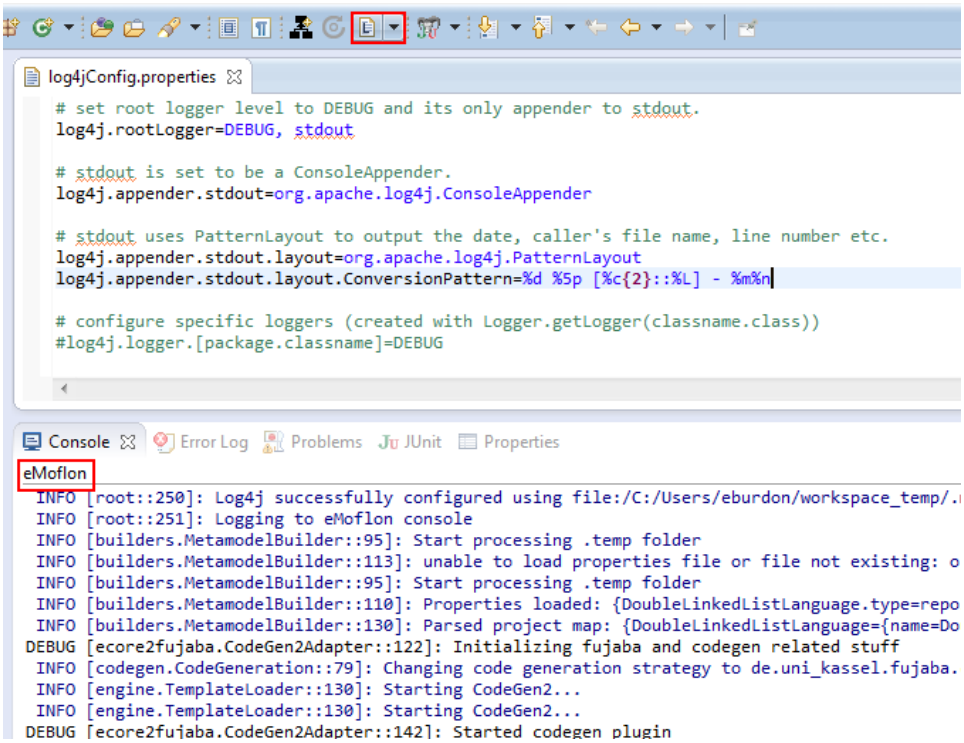


Figure 2.4: The eMoflon console with log messages

▷ Next [visual]
▷ Next [textual]

## 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, navigate to and select "Extensions/Add-In Windows" (Fig. 2.5). This will activate our tool's full control panel.

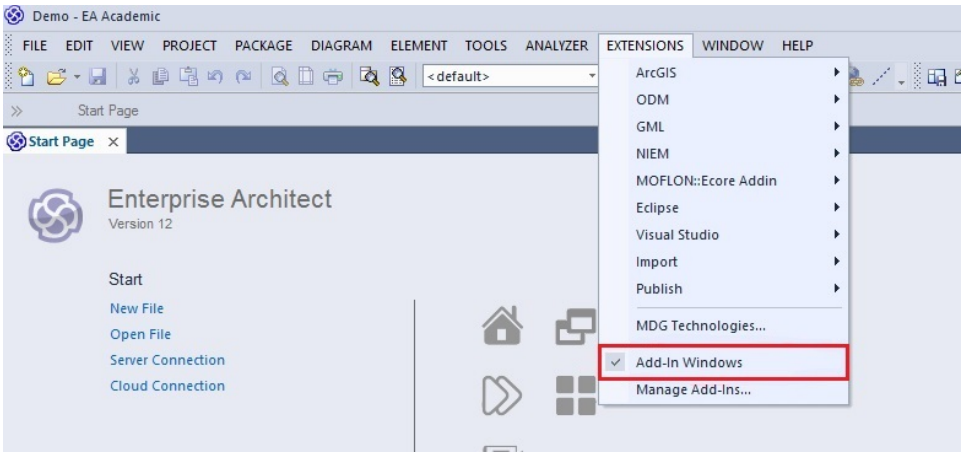


Figure 2.5: Export from EA

- 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 `All` (Fig. 2.6).

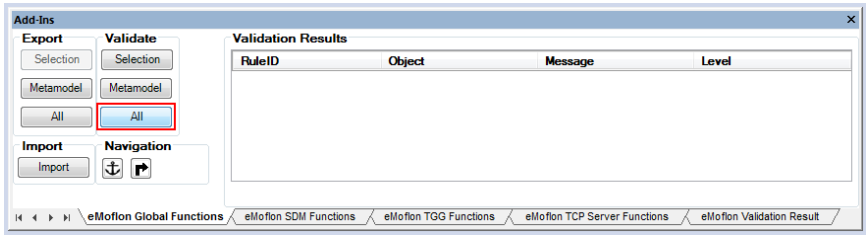


Figure 2.6: eMoflon's control panel in EA

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

- The three asterisks (Fig. 2.7) signal that the project still needs to be built.

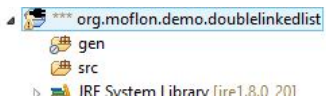


Figure 2.7: Dirty projects are marked with ***

- 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 (Fig. 2.8). 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`.

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

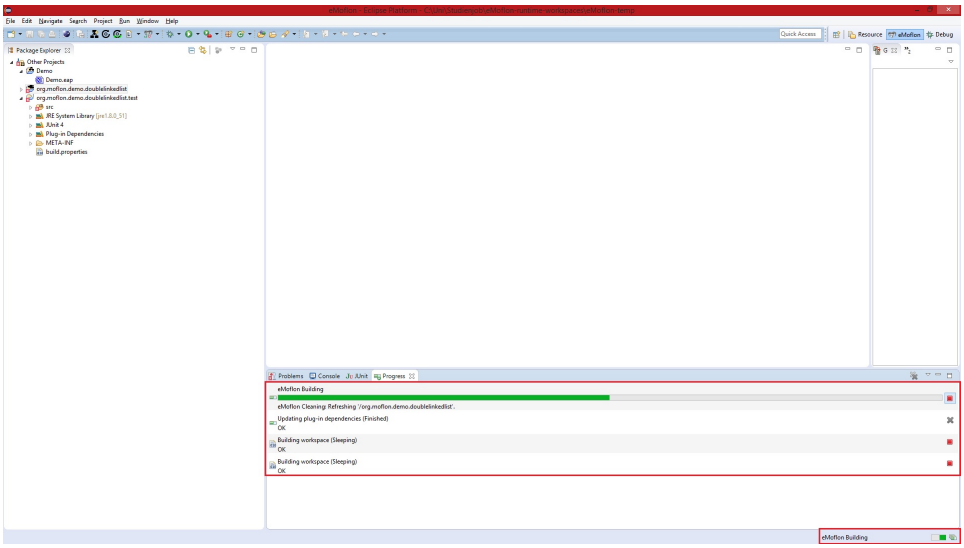

Figure 2.8: Eclipse workspace when using visual syntax

## 2.2 A first look at MOSL

Please note that the textual syntax is not as thoroughly tested as the visual syntax because most of our projects are built with the visual syntax. This means: Whenever something goes wrong even though you are sure to have followed the instructions precisely, do not hesitate to contact us via contact@emoflon.org.

- You should immediately have 3 folders available in your project explorer – Your metamodel project, `org.moflon.demo.doublelinkedlist`, and `orgmoflon.demo.doublelinkedlist.test` (Fig. 2.9). Initially, you'll see a red exclamation mark indicating there are errors, but once you give Eclipse a few seconds to refresh, all problems should be solved.

- That's it! You're all set up! But, while you're here, feel free to explore the "Demo" folder. It has the basic code that implements this demo, and we recommend you take a brief look to get a feel for the the general syntax.

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

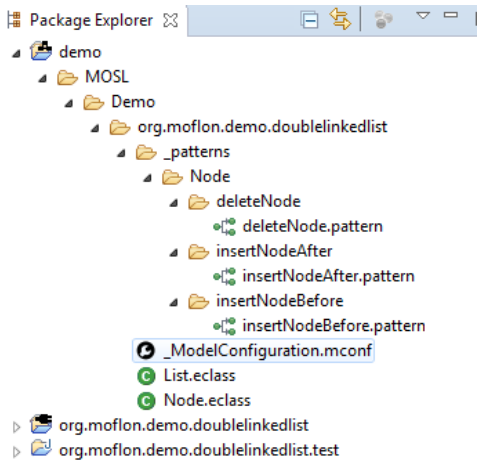


Figure 2.9: Metamodel file structure

# 3 Validate your installation with JUnit

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

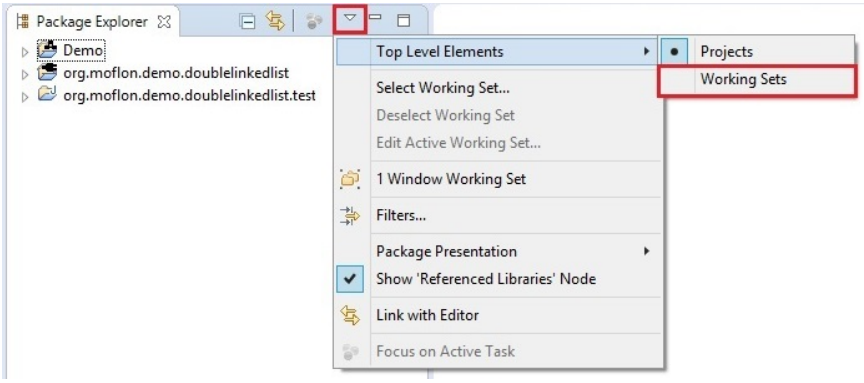


Figure 3.1: Top level elements in Eclipse

- Locate "Other Projects/orgmoflon.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`.

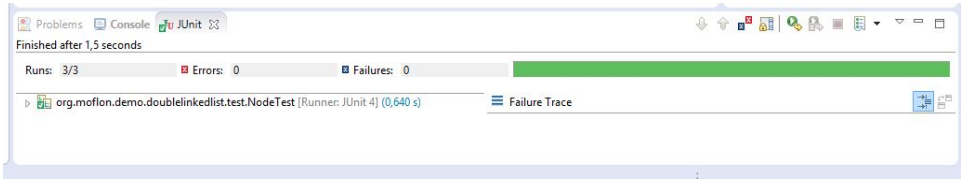


Figure 3.2: All's well that ends well...

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

▷ Next [visual]
▷ Next [textual]

# 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 (Fig. 4.1).

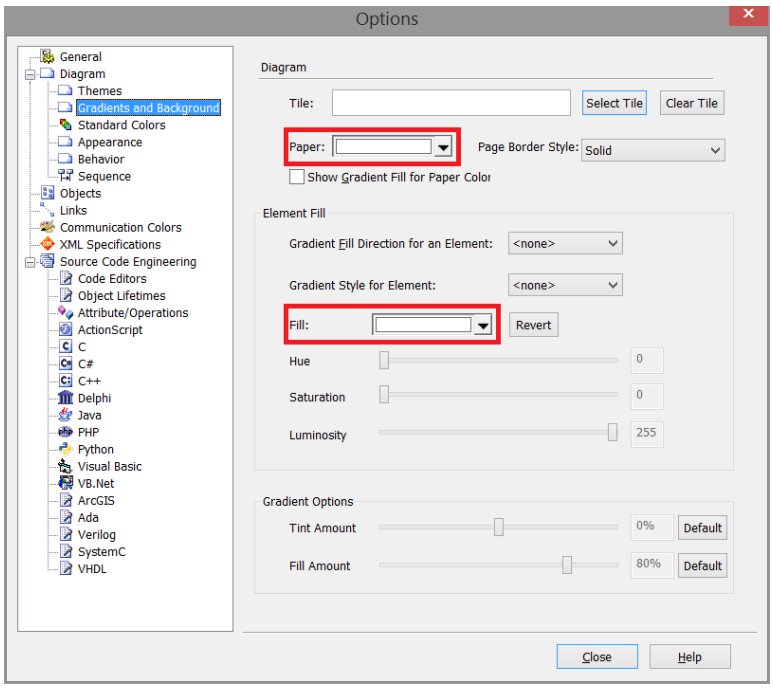


Figure 4.1: Suggested paper background and element fill

- In the "Standard Colors" tab, and set your colours to reflect Fig. 4.2.

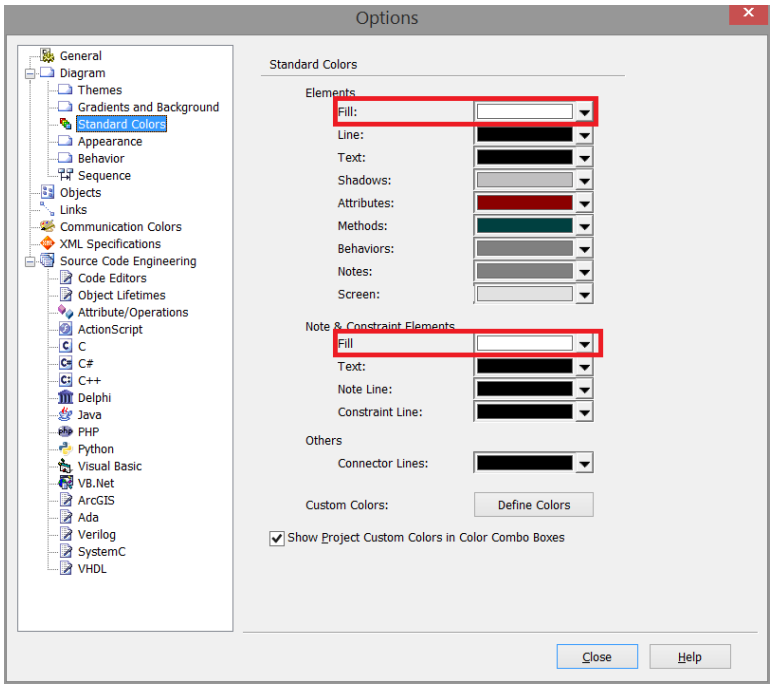


Figure 4.2: Our choice of standard colours for diagrams in EA

- In the same dialogue, go to "Diagram/Appearance" and reflect the settings in Fig. 4.3. Again, this is just a suggestion and not mandatory.

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

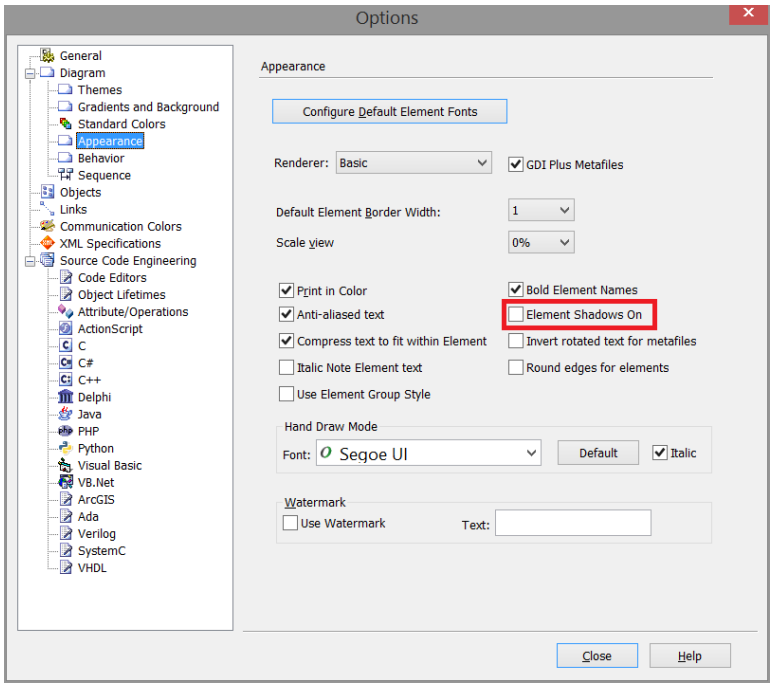


Figure 4.3: Our choice of the standard appearance for model elements

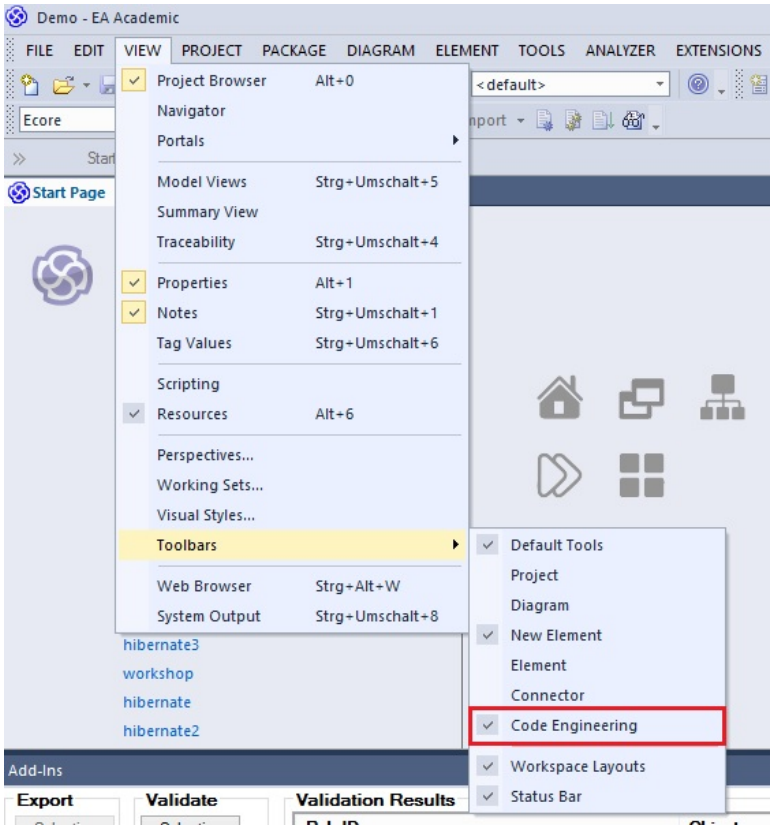


Figure 4.4: Open the "Code Engineering" toolbar

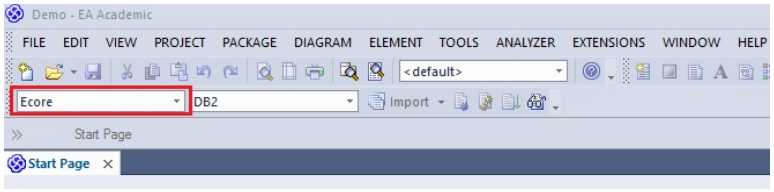


Figure 4.5: Make sure you set the standard language to `Ecore`

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` consists of a single package `DoubleLinkedListLanguage`. 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` (Fig. 4.6).

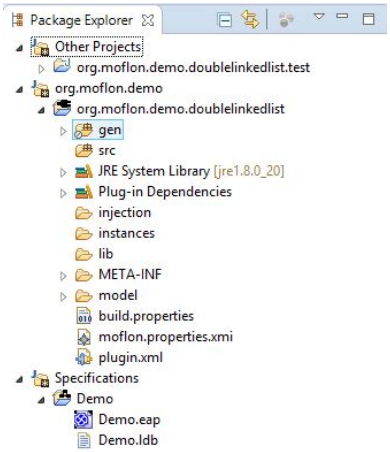


Figure 4.6: Project structure

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 `Demo` containing our EA project file `Demo.eap`.

Figure 4.7 depicts how the Eclipse working set `Demo` and its contents were generated from the EA model `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.

These projects, however, are of a different *nature* than, for example, metamodel 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.

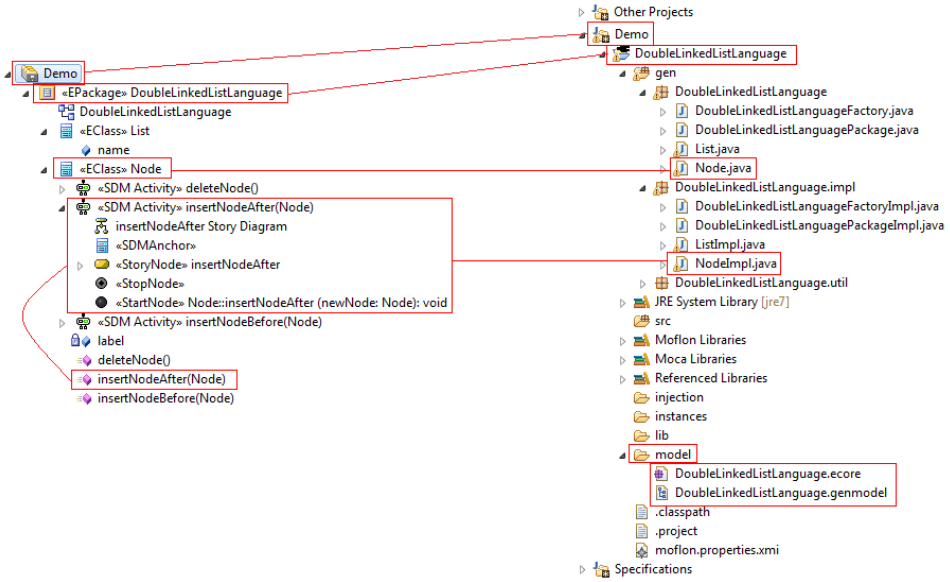


Figure 4.7: From EA to Eclipse

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 metamodelling 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 Fig. 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 modelled `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 *modelled* 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 modelled in an abstract, platform (programming language) independent fashion using a blend of activity diagrams and a "graph pattern" language called *Story Driven Mod-*

*elling* (SDM). In our EA project, these *Story Diagrams* or simply *SDMs*, are placed in SDM Containers named according to the method they implement. E.g. ≪`SDM Activity`≫ `insertNodeAfter SDM` for the method `insertNodeAfter(Node)` as depicted in Fig. 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 Fig. 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).

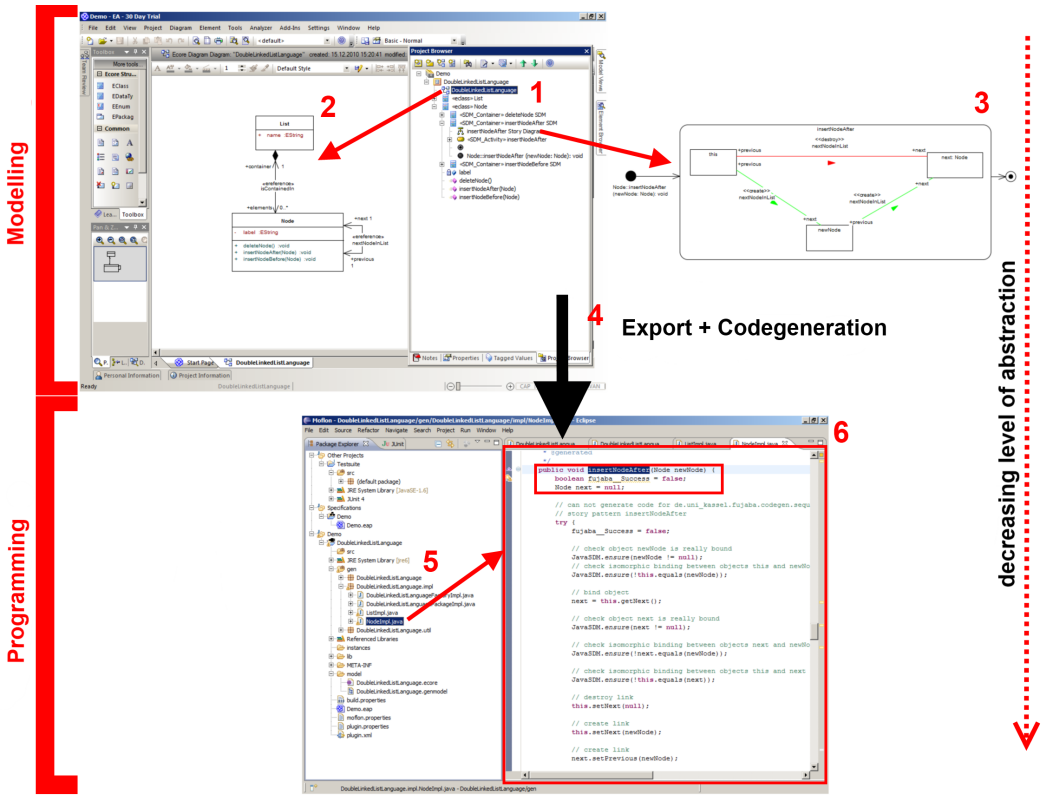


Figure 4.8: Overview

▷ Next

## 4.2 Your MOSL workspace

As you now know, eMoflon is a plug-in for Eclipse. More precisely, eMoflon requires the Eclipse Modeling Framework (EMF) in order to work. EMF uses two separate models, a Genmodel and an Ecore model, for code generation. The Genmodel contains boring information about code generation such as path, file prefixes, and other information. We are more interested in the Ecore model, which we specify with MOSL.

When you switched the "Top Level Elements" from `Projects` to `Working Sets`, you noticed that a few extra nodes were displayed in the project browser. Each node you see has different criteria for grouping related Eclipse Projects together, which makes them your project *working sets*.

The `Specifications` working set contains all *metamodel projects* in a workspace (Fig. 4.9). This means that for every new metamodel you create in your current workspace, all of the relevant files will be placed here.

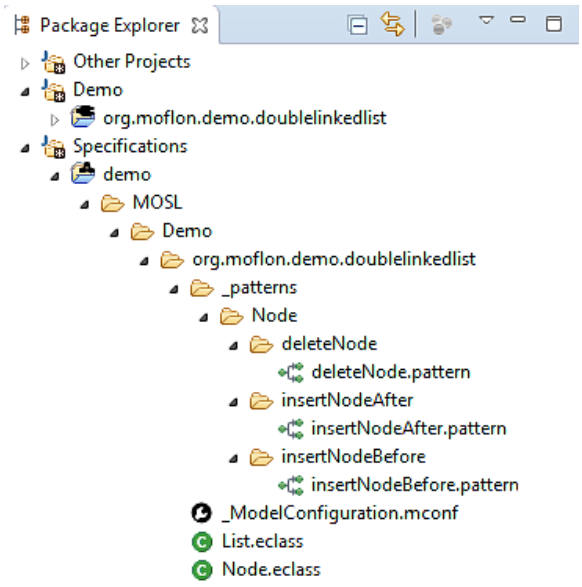


Figure 4.9: Specification working set

Let's have a look at the two *eclass* files. While you can combine several short class declarations in a single file in some languages (such as Java), each class is kept separated in its own file when using MOSL.

Inspect the file `List.eclass` (Listing 4.1), and you'll see it has just one *EAttribute*. EAttributes are defined by their name, followed by a colon and type (line 3).

This class also has one *EReference*, a *container reference*. This is represented by the diamond operator in front of a long arrow (two minuses and a greater than) which points at a type. In the middle of the arrow the EReference name is represented, followed by their multiplicity (line 6).

```
1 class List
2 {
3   name : EString
4
5   // Sets a container reference to hold an unknown number of 'node'
6   <> - elements(0..*) -> Node
7 }
```

Listing 4.1: The `List` eclass

Switch to the `Node` eclass (Listing 4.2) and you can observe the second reference type, a *simple reference*. This reference type will be created by writing a container reference without a diamond (line 6).

In the `Node` eclass , a few methods have been declared. You can see that each function is remarkably small. In fact, the only thing the functions are doing are invoking *patterns*. These patterns represent structural changes, and their container functions are used exclusively for control flow (i.e., sequences, branches, and loops).

```
1  class Node
2  {
3    label : EString
4
5    // Set up simple references
6    - container(0..1) -> List
7    - next(1..1) -> Node
8    - previous(1..1) -> Node
9
10   // Destroys all references to this node and repairs those that
         remain
11   deleteNode() : void
12   {
13     [deleteNode]
14     return
15   }
16
17   // Sets 'next' reference of current node to 'newNode,' updating
         other nodes where required
18   insertNodeAfter(newNode : Node) : void
19   {
20     [insertNodeAfter]
21     return
22   }
23
```

```
24    // Sets 'previous' reference of current node to 'newNode' and
          updates any others
25    insertNodeBefore(newNode: Node) : void
26    {
27      [insertNodeBefore]
28      return
29    }
30  }
```

Listing 4.2: The `Node` eclass

After many long discussions, it was decided that patterns should always be implemented in separate files. Inspect Fig. 4.9 again, and observe the locations where the patterns are placed. You'll notice that there is a folder for the `Node` EClass, and a subfolder for each method. There's no folder for `List` because it never calls a pattern.

Check out the `deleteNode` pattern (Listing 4.3). You can see that there is a destroy command on `@ this`, denoted by `--`. It gets rid of the node, but it doesn't do anything about the previous and next references defined on it (remember that we're dealing with a double linked list here!). That's because when the node is removed, everything attached to it will be automatically cleaned and removed. The final command reconnects the next and previous nodes of the deleted node to close the 'hole' in the list. This is accomplished by setting (`++`) the previous reference of the next node to the previous node.

```
1   pattern deleteNode
2   {
3
4     // repairs 'next' link
5     @ this : Node
6     {
7       ++ - next -> newNext
8     }
9
10    // delete next Node
11    -- next : Node
12    {
13      -- - next -> newNext
14    }
15
16    newNext : Node
17  }
```

Listing 4.3: The deleteNode pattern

So that's a quick overview of the MOSL language but how do we generate code from all of this?

First, eMoflon does not generate code with every change. To improve performance, only the parser is invoked when you save files. This means that to generate code for the project, you need to either invoke the `Build` command from the eMoflon context menu after right clicking your metamodel project, or by navigating to the "Build (dirty projects)" button next to the "New Metamodel" button. Cleaning first deletes all generated files while building without cleaning tries to merge the newly generated code into existing files. We'll discuss this in more detail later in the handbook.

# 5 Generated code vs. hand-written code

Now that you've worked through the specifics of your syntax, lets 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` (Fig. 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.

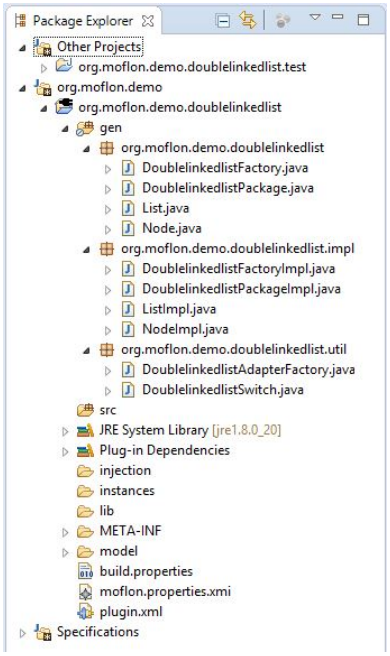


Figure 5.1: Package structure of generated code (`gen`)

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.

# 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 II[8]! 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 metamodelling 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 III[9], or Triple Graph Grammars (TGGs) in Part IV[10]. We'll provide instructions on how to easily download all the required resources so you can jump right in. For further details on each part, refer to Part 0[11].

Cheers!

[8] Download: `http://tiny.cc/emoflon-rel-handbook/part2.pdf`
[9] Download: `http://tiny.cc/emoflon-rel-handbook/part3.pdf`
[10] Download: `http://tiny.cc/emoflon-rel-handbook/part4.pdf`
[11] Download: `http://tiny.cc/emoflon-rel-handbook/part0.pdf`