An Introduction to Metamodelling and Graph Transformations

with eMoflon

Copyright ©2011–2012 Anthony Anjorin and Contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but *without any warranty*; without even the implied warranty of *merchantability* or *fitness for a particular purpose*. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this document; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Contents

Chapter 1

Introduction

This tutorial has been engineered to be *fun*.

If you work through it and, for some reason, do *not* have a resounding "I-Rule" feeling afterwards, please send us an email and tell us how to improve it: `contact@moflon.org`

Figure 1.1: How you should feel when you're done.

To enjoy the experience, you should be fairly comfortable with Java or a comparable object-oriented langauge, and know how to perform basic tasks in Eclipse. Although we assume this, we give references to help bring you up to speed as necessary. Last but not least, very basic knowledge of common UML notation would be helpful.

Our goal is to give a *hands-on* introduction to metamodelling and graph transformations using our tool *eMoflon.* The idea is to *learn by doing* and all concepts are introduced while working on a concrete example. The language and style used throughout is intentionally relaxed and non-academic. For those of you interested in further details and the mature formalism of graph transformations, we give relevant references throughout the tutorial.

The tutorial is divided into two main parts: In the first part (Chapter 2), we provide a very simple example and a few JUnit tests to test the installation and configuration of eMoflon.

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

In the second part of the tutorial (Chapter 3), we go, step-by-step, through a more realistic example that showcases most of the features we currently support.

Working through this chapter should serve as a basic introduction to model-driven engineering, metamodelling and graph transformations.

One last thing – at the moment we unfortunately only support Windows. This should hopefully change in future releases.

That's it – sit back, relax, grab a coffee and enjoy the ride!

Chapter 2

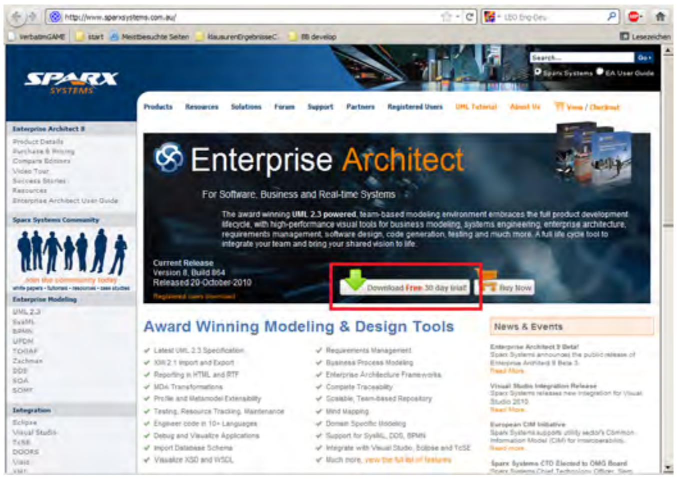
Installation

2.1 Install Our Plugin for Enterprise Architect (EA)

Enterprise Architect (EA) is a modelling tool that supports UML[1] and a host of other modelling languages. EA is not only affordable but is also quite flexible and can be extended via *plugins* to support new modelling languages.

- ▶ Download and install EA (Fig. 2.1)

  Go to `http://www.sparxsystems.com.au/` to get a free 30 day trial.



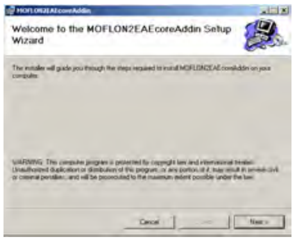
http://www.sparxsystems.com.au/
SPARX SYSTEMS
Products Resources Solutions Forum Support Partners Registered Users UML Tutorial About Us View / Checkout
Enterprise Architect
For Software, Business and Real-time Systems
Current Release
Version 8, Build 864
Released 20-October-2010
Download Free 30 day trial
Buy Now
Award Winning Modeling & Design Tools
News & Events

Figure 2.1: Download Enterprise Architect

[1]Unified Modelling Language

- ▶ Install our EA-Plugin (Fig. 2.2) to add support for our modelling languages.

  Download `http://www.moflon.org/fileadmin/download/moflon-ide/eclipse-plugin/ea-ecore-addin/ea-ecore-addin.zip`, unpack, and run setup.exe


MOFLON2EAEcoreAddin

Welcome to the MOFLON2EAEcoreAddin Setup Wizard

The installer will guide you through the steps required to install MOFLON2EAEcoreAddin on your computer.

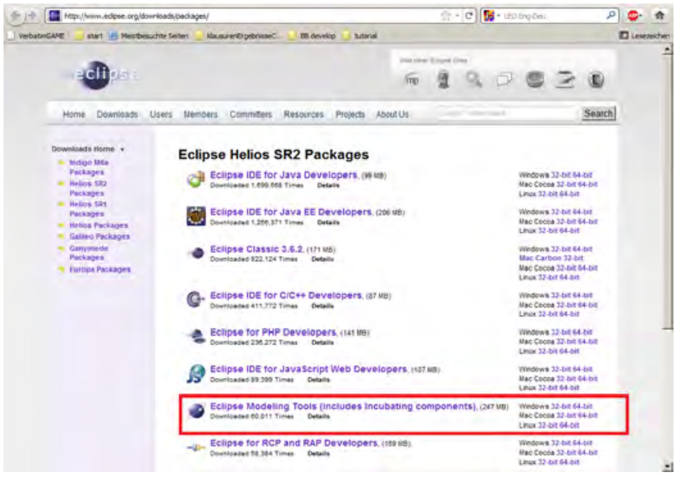
WARNING: This computer program is protected by copyright law and international treaties. Unauthorized duplication or distribution of this program, or any portion of it, may result in severe civil or criminal penalties, and will be prosecuted to the maximum extent possible under the law.

Cancel | < Back | Next >

Figure 2.2: Install our plugin for EA.

2.2 Install Our Plugin for Eclipse

- ▶ Download and install Eclipse for Modeling "Eclipse Modeling Tools (includes Incubating components)" (Fig. 2.3) from: `http://www.eclipse.org/downloads/packages/`



Eclipse Helios SR2 Packages

Eclipse IDE for Java Developers, (99 MB) Downloaded 1,699,668 Times Details	Windows 32-bit 64-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit
Eclipse IDE for Java EE Developers, (206 MB) Downloaded 1,266,371 Times Details	Windows 32-bit 64-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit
Eclipse Classic 3.6.2, (171 MB) Downloaded 822,124 Times Details	Windows 32-bit 64-bit, Mac Carbon 32-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit
Eclipse IDE for C/C++ Developers, (87 MB) Downloaded 411,772 Times Details	Windows 32-bit 64-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit
Eclipse for PHP Developers, (141 MB) Downloaded 236,272 Times Details	Windows 32-bit 64-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit
Eclipse IDE for JavaScript Web Developers, (107 MB) Downloaded 89,399 Times Details	Windows 32-bit 64-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit
Eclipse Modeling Tools (includes Incubating components), (247 MB) Downloaded 60,011 Times Details	Windows 32-bit 64-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit
Eclipse for RCP and RAP Developers, (189 MB) Downloaded 58,384 Times Details	Windows 32-bit 64-bit, Mac Cocoa 32-bit 64-bit, Linux 32-bit 64-bit

Figure 2.3: Download Eclipse Modeling Tools.

- ▶ Install our Eclipse Plugin from the following update site[2] [3]: `http://www.moflon.org/fileadmin/download/moflon-ide/eclipse-plugin/update-site2`

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

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

2.3 Get a Simple Demo Running

- ▶ Go to "Window/Open Perspective/Other. . . " and choose Moflon (Fig. 2.4).

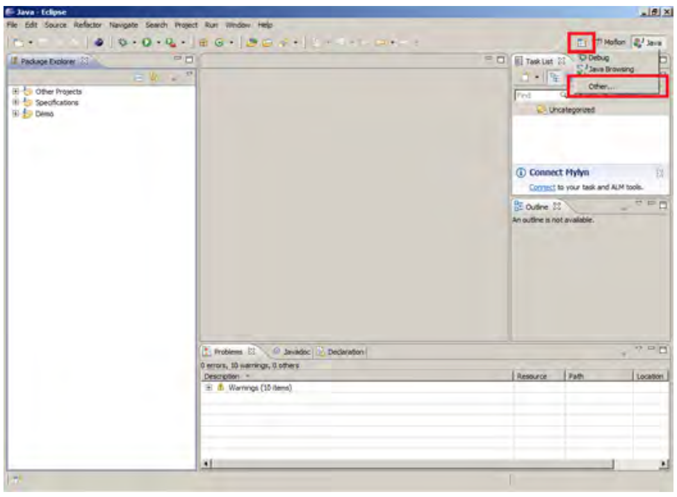


Figure 2.4: Choose the Moflon Perspective.

- ▶ In the toolbar a new action set should have appeared. Choose "New Metamodel" (Fig. 2.5). The button with an "L" shows you our logfile (important input for us if something goes wrong!).

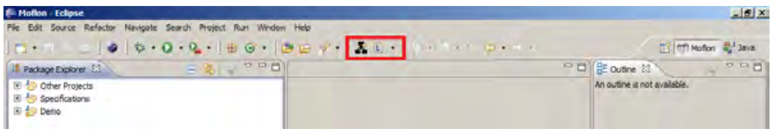

Moflon - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Other Projects
Specifications
Demo
Moflon
Java
Outline
An outline is not available.

Figure 2.5: Eclipse New Metamodel

- ▶ Enter "Demo" as the name of the new metamodel project and confirm. An empty EA project file "Demo.eap" will be created in a new project with a certain project structure according to our conventions.

- ▶ Choose working sets as your top level element in the package explorer (Fig. 2.6). We work a lot with working sets and use them to structure the workspace in Eclipse.

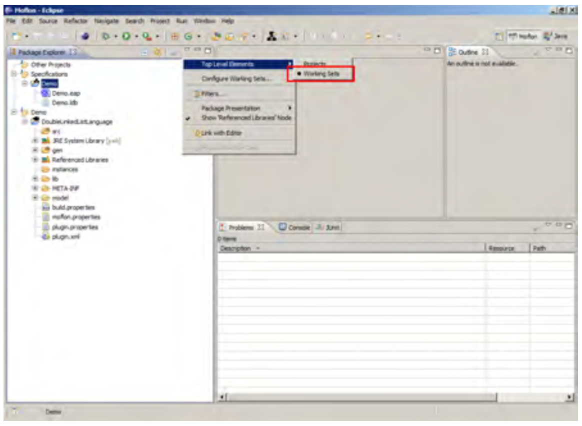


Figure 2.6: Choose Working Sets as Top Level Elements.

- ▶ Open the newly created project and replace the “Demo.eap” file with the Demo.eap that you will find in the same folder as this tutorial. This EA file already contains our simple demo project.
- ▶ Double click “Demo.eap” to start EA. Please choose “Ultimate” when starting EA for the first time.
- ▶ In EA, choose “Add-Ins/MOFLON::Ecore Addin/Export all to Workspace” (Fig. 2.7).

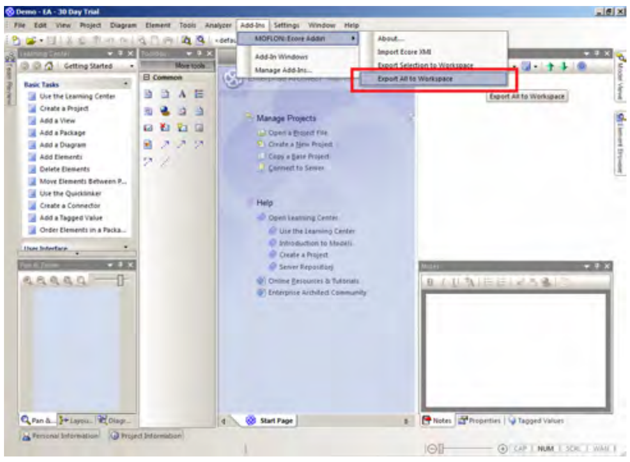


Figure 2.7: Export from EA with our plugin.

▶ 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 in the project, and refreshing triggers a build process that invokes our different code generators automatically. You should be able to monitor the progress in the lower right corner (Fig. 2.8). Pressing the symbol opens a monitor view that gives more details of the build process.

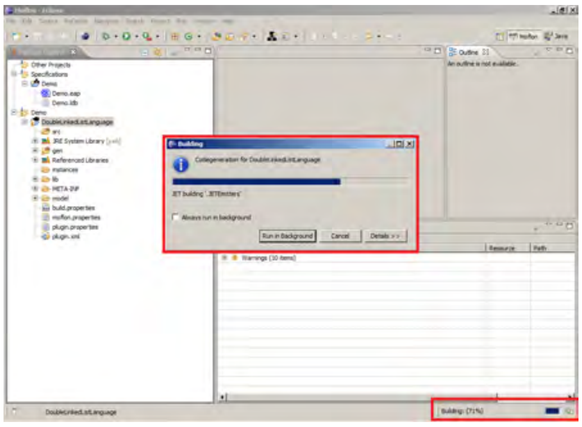


Figure 2.8: Automatically building the workspace after a refresh.

2.4 Validate Your Installation with JUnit

▶ Go to "File/Import/General/Existing Projects into Workspace" (Fig. 2.9) and choose the Testsuite project that is also in the same folder as this tutorial.

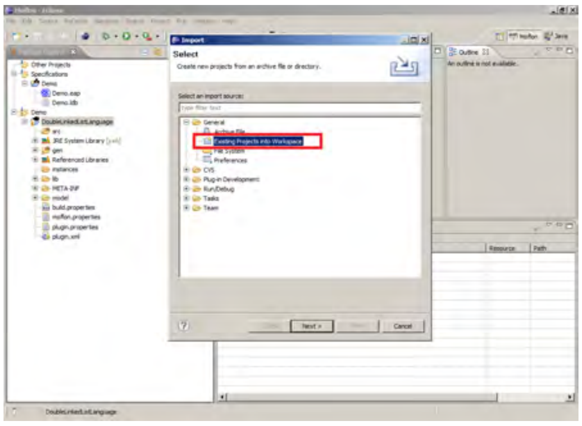


Figure 2.9: Import our Testsuite as an existing project.

At this point, your workspace should resemble Fig. 2.10.

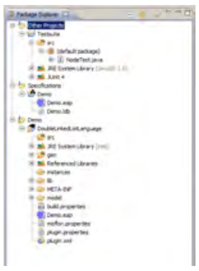


Figure 2.10: Workspace in Eclipse.

► Right-click on the Testsuite project and select "Run as/JUnit Test". Congratulations! If you see a green bar (Fig. 2.11), then everything has been set-up correctly and you are now ready to start metamodelling!

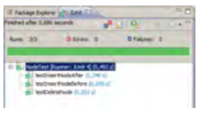


Figure 2.11: All's well that ends well...

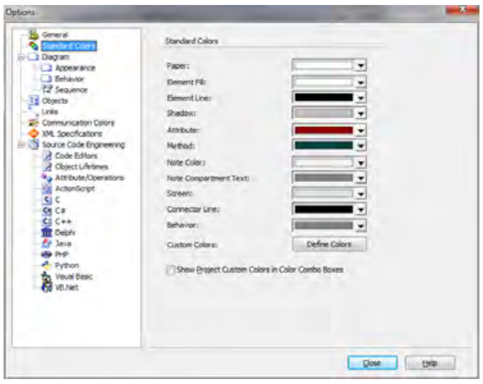
2.5 Project Structure and Setup

Now that everything is installed and setup properly, let's take a closer look at the different workspaces and workflow. Before we continue, please make a few slight adjustments to EA so you can easily compare your current workspace to our screenshots:

► Select "Tools/Options/Standard Colors" in EA, and set your colours to reflect Fig. 2.12.

► In the same dialogue, select "Diagram/Appearance" and reflect the settings in Fig. 2.13.

► Last but not least, and still in the same dialogue, select "Source Code Engineering" and be sure to choose "Ecore" as the default language for code generation (Fig. 2.14).

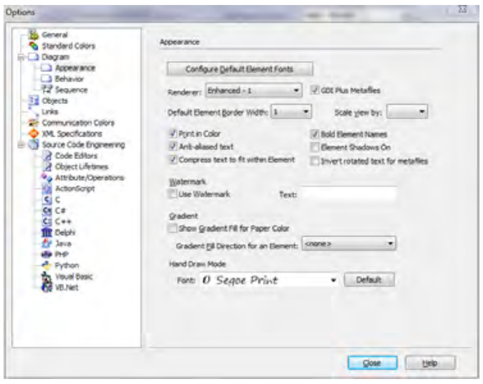


Standard Colors

Paper:	⬜
Element Fill:	⬜
Element Line:	⬛
Shadow:	▨
Attribute:	🟥
Method:	🟩
Note Color:	⬜
Note Compartment Text:	▨
Screen:	⬜
Connector Line:	⬛
Behavior:	▨
Custom Colors:	Define Colors

☐ Show Project Custom Colors in Color Combo Boxes

Figure 2.12: Our choice of standard colours for diagrams in EA.



Appearance

Renderer: Enhanced - 1 ☑ GDI Plus Metafiles

Default Element Border Width: 1 Scale view by:

☑ Print in Color ☑ Bold Element Names

☑ Anti-aliased text ☐ Element Shadows On

☑ Compress text to fit within Element ☐ Invert rotated text for metafiles

Watermark

☐ Use Watermark Text:

Gradient

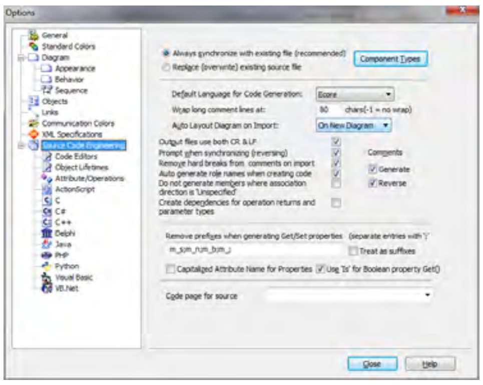
☐ Show Gradient Fill for Paper Color

Gradient Fill Direction for an Element: <none>

Hand Draw Mode

Font: Segoe Print

Figure 2.13: Our choice of the standard appearance for model elements in EA.



● Always synchronize with existing file (recommended)

○ Replace (overwrite) existing source file

Default Language for Code Generation: Ecore

Wrap long comment lines at: 80 chars(-1 = no wrap)

Auto Layout Diagram on Import: On New Diagram

Output files use both CR & LF	☑
Prompt when synchronizing (reversing)	☑
Remove hard breaks from comments on import	☑
Auto generate role names when creating code	☑
Do not generate members where association direction is 'Unspecified'	☐
Create dependencies for operation returns and parameter types	☐

Comments

☑ Generate

☑ Reverse

Remove prefixes when generating Get/Set properties (separate entries with ';')

m_;s_;m_;n;m_;b;m_;

☐ Treat as suffixes

☐ Capitalized Attribute Name for Properties ☑ Use 'Is' for Boolean property Get()

Code page for source

Figure 2.14: Make sure you set the standard language to Ecore.

In your EA "workspace", actually referred to as an *EA project*[4], take a careful look at the project structure: The root node `Demo`[5] 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 can however consist of numerous models that in turn group numerous packages.

Now switch to your *Eclipse workspace* and note the two nodes named `Specifications` and `Demo`. 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. A metamodel project contains a single EAP (EA project) file and is used to communicate with EA and initiate codegeneration 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`.

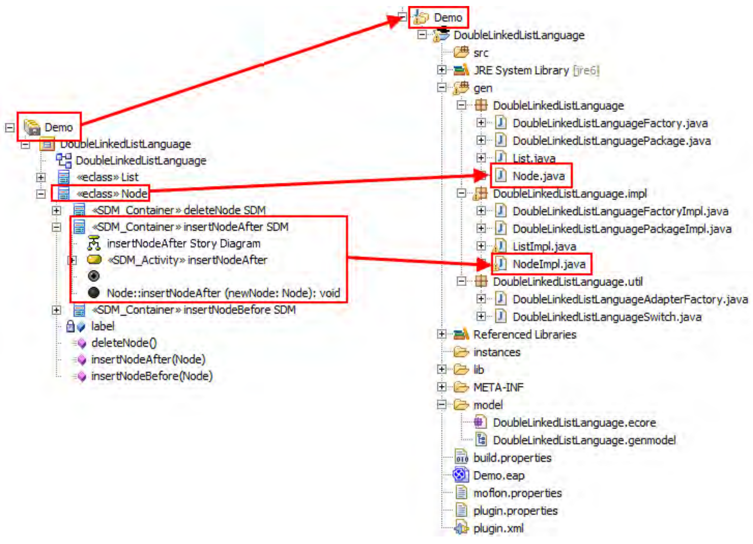

Demo
DoubleLinkedListLanguage
DoubleLinkedListLanguage
«eclass» List
«eclass» Node
«SDM_Container» deleteNode SDM
«SDM_Container» insertNodeAfter SDM
insertNodeAfter Story Diagram
«SDM_Activity» insertNodeAfter
Node::insertNodeAfter (newNode: Node): void
«SDM_Container» insertNodeBefore SDM
label
deleteNode()
insertNodeAfter(Node)
insertNodeBefore(Node)
Demo
DoubleLinkedListLanguage
src
JRE System Library [jre6]
gen
DoubleLinkedListLanguage
DoubleLinkedListLanguageFactory.java
DoubleLinkedListLanguagePackage.java
List.java
Node.java
DoubleLinkedListLanguage.impl
DoubleLinkedListLanguageFactoryImpl.java
DoubleLinkedListLanguagePackageImpl.java
ListImpl.java
NodeImpl.java
DoubleLinkedListLanguage.util
DoubleLinkedListLanguageAdapterFactory.java
DoubleLinkedListLanguageSwitch.java
Referenced Libraries
instances
lib
META-INF
model
DoubleLinkedListLanguage.ecore
DoubleLinkedListLanguage.genmodel
build.properties
Demo.eap
moflon.properties
plugin.properties
plugin.xml

Figure 2.15: From EA to Eclipse

Figure 2.15 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 packages 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, and are called *repository projects*. A nature is Eclipse lingo for "project type" and is visually indicated by a corre-

[4]Words are set in italics when they represent concepts that are introduced or defined in the corresponding paragraph for the first time.

[5]Words set in a `mono-space font` refer to things that you should find in a tool, dialogue, figure or code.

sponding nature icon on the project folder. Our metamodel projects sport a spanking little class diagram symbol. Repository projects are generated automatically with a certain project structure according to our conventions. The `model` subfolder is probably most important, and contains an *Ecore model.* Ecore is a metamodelling language that provides building blocks like *classes* and *references* for defining the static structure (concepts and relations between concepts) of a system. The export function of our EA plugin generates a valid Ecore model from the corresponding EA model and persists it as an XML file in the `model` subfolder. In our concrete example, this is the `DoubleLinkedListLanguage.ecore` file. Go ahead and double-click it to open the file in a simple tree-view editor in Eclipse. If you are really interested in the nitty-gritty details or have a masochistic hang, right-click the file and select "Open With/Text Editor".

This Ecore model is used to drive a codegenerator that maps the model to Java interfaces and classes. The generated Java code that represents the model is often referred to as a *repository* and 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 like Java. This is why we indicate repository projects using a cute adapter/plug symbol on the project folder.

Figure 2.15 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. Indicated by the source folders `src` and `gen`, we advocate a clean separation of hand-written (this should go in `src`) and generated code (lands automatically in `gen`). As we shall see later in the tutorial, hand-written code can also be integrated directly in generated classes and, if marked appropriately, merged nicely by the codegenerator. This is sometimes more elegant for small helper functions but can quickly get problematic especially in combination with source code management systems.

If you take a careful look at the code structure in `gen`, 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 like multiple inheritance (allowed and very common in Ecore models). A further package `util` contains some auxiliary classes like 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 EA 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 codegenerator 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 codegenerator 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 like Java. We use a mapping defined and implemented by the Eclipse Modelling Framework (EMF) which tends to favour efficiency and simplicity.

Although getting the *details* of mapping the static structure of our models to Java might be extremely difficult, it seems for the most part pretty straight forward. A fantastic productivity boost in any case but (yawn) not exactly exciting.

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`. 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 Modelling (SDM). In our EA project, these “Stories”, “Story Models” or simply “SDMs” are placed in SDM Containers named according to the method they implement. E.g. $\ll$ `SDM Container`$\gg$ `insertNodeAfter SDM` for the method `insertNodeAfter(Node)` as depicted in Fig. 2.15. We’ll spend the rest of the tutorial understanding why SDMs are so crazily cool!

To recap all we've discussed, let's consider the complete workflow as depicted in Figure 2.16. 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 codegeneration (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).

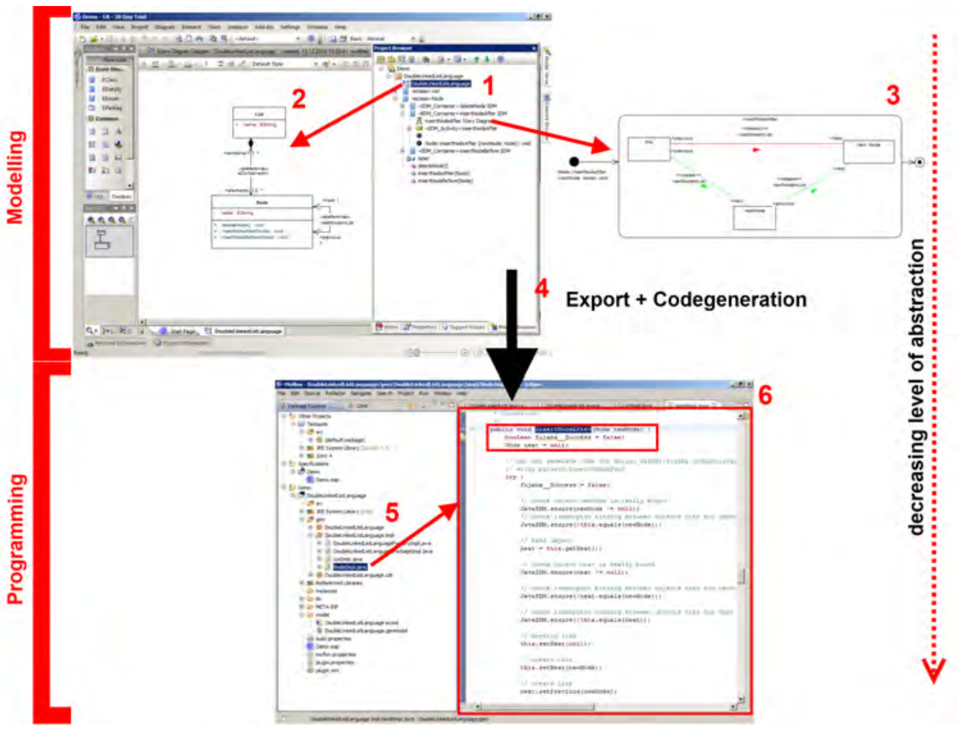


Figure 2.16: Overview

If you feel a bit lost at the moment please be patient, this first chapter has been a lot about installation and tool support and only aims at giving a very brief glimpse at the big picture of what is actually going on.

In the following chapter, we shall go step-by-step through a hands-on example and cover the core features of Ecore (static structure) and SDM (behaviour). We shall also give clear and simple definitions for the most important metamodelling and graph transformation concepts, always referring to the concrete example and providing lots of references for further reading.

Chapter 3

Modelling a Memory Box