An Introduction to Metamodelling and Graph Transformations

with eMoflon

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 Addin 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 *addins* to support new modelling languages.

- Download and install EA (Fig. 2.1)

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

Figure 2.1: Download Enterprise Architect

[1] Unified Modelling Language

- Install our EA-Addin (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

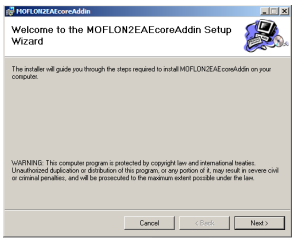

Figure 2.2: Install our plugin for EA.

2.2 Install Our Plugin for Eclipse

- Download and install Eclipse for Modeling "Eclipse Modeling Tools (includes Incubating components)" (Fig. 2.3)[2] from: `http://www.eclipse.org/downloads/packages/`

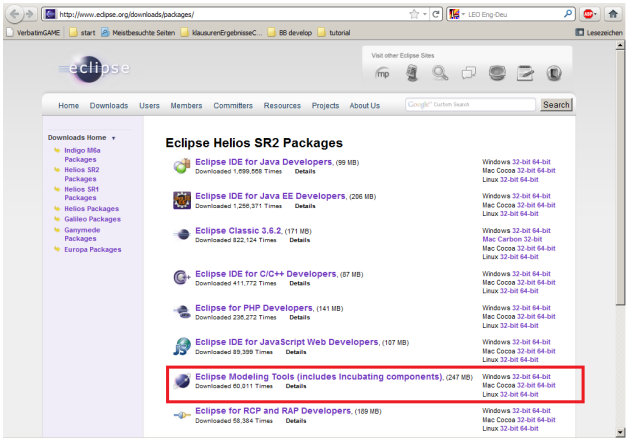

Figure 2.3: Download Eclipse Modeling Tools.

- Install our Eclipse Plugin from the following update site[3] [4]: `http://www.moflon.org/fileadmin/download/moflon-ide/eclipse-plugin/update-site2`

[2] Tested for Eclipse 3.6 (Helios).

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

[4] 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..."[5] and choose Moflon (Fig. 2.4).

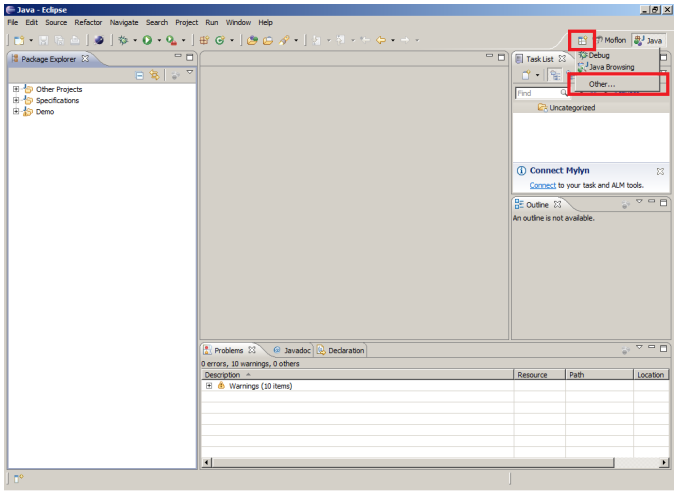

Figure 2.4: Choose the Moflon Perspective.

- 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!).

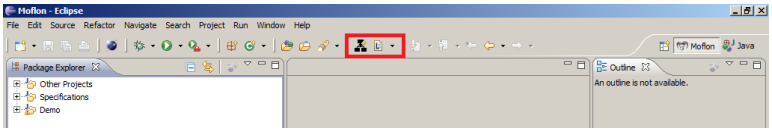

Figure 2.5: Eclipse New Metamodel

- 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. For the moment, please do not rename, move or delete any folders.

[5] A path given as "foo/bar" indicates how to navigate in a series of menus and submenus.

- 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.

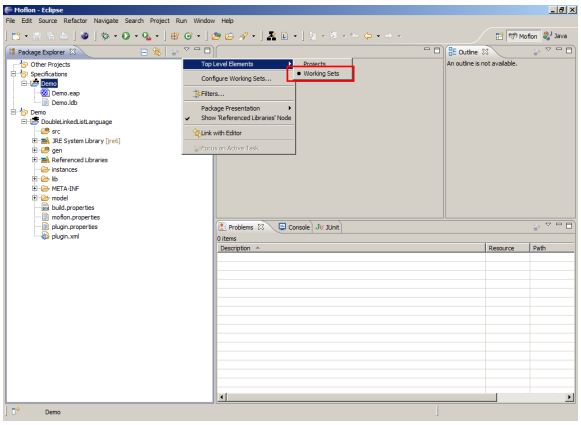

Figure 2.6: Choose Working Sets as Top Level Elements.

- 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). You can of course browse the project structure, but please do not rename, move or delete anything yet.

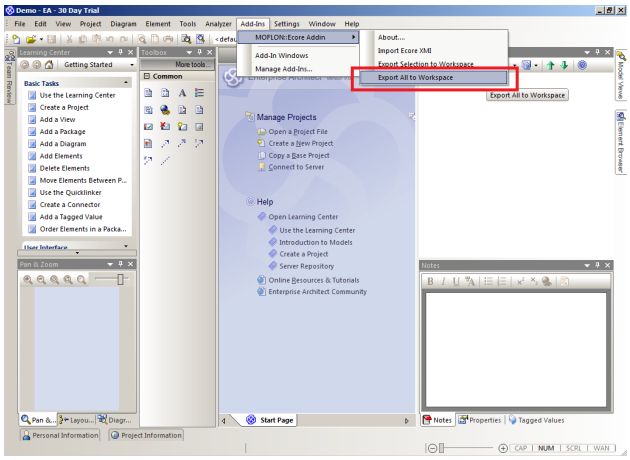

Figure 2.7: Export from EA with our plugin.

- 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. You don't need to worry about any of these details, just remember to refresh your Eclipse workspace after an export.

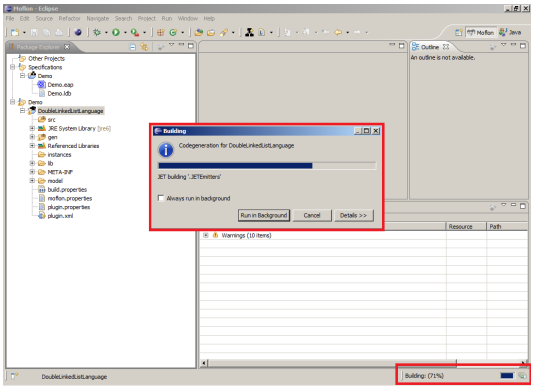

Figure 2.8: Automatically building the workspace after a refresh.

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.

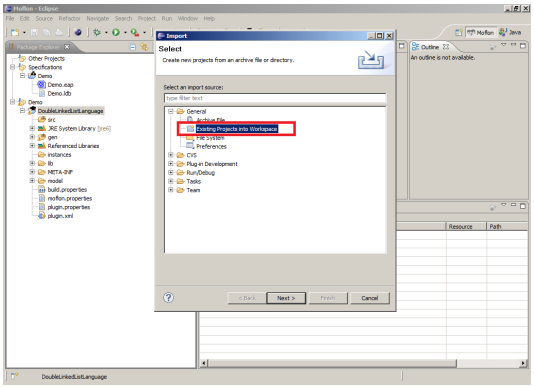

Figure 2.9: Import our Testsuite as an existing project.

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

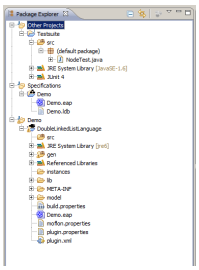

Figure 2.10: Workspace in Eclipse.

- 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!

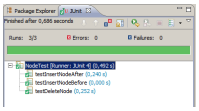

Figure 2.11: All's well that ends well...

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 our 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. This is advisable but you're of course free to choose your own colour schema.
- In the same dialogue, select "Diagram/Appearance" and reflect the settings in Fig. 2.13. Again this is just a suggestion and not mandatory.
- 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). This setting is very important.

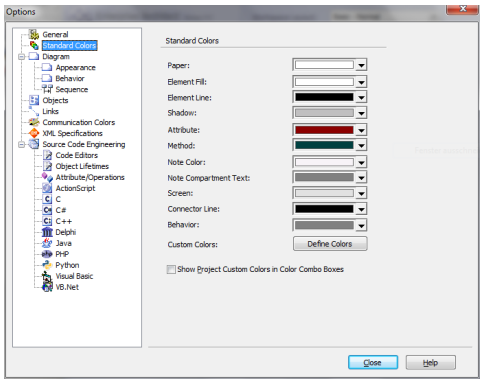

Figure 2.12: Our choice of standard colours for diagrams in EA.

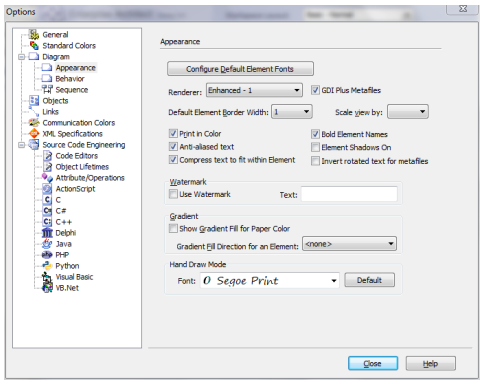

Figure 2.13: Our choice of the standard appearance for model elements in EA.

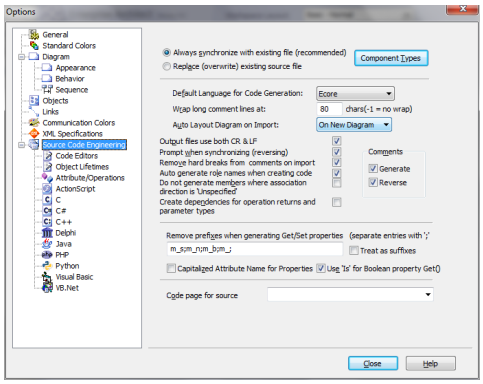

Figure 2.14: Make sure you set the standard language to Ecore.

In your EA "workspace", actually referred to as an *EA project*[6], take a careful look at the project structure: The root node `Demo`[7] 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`.

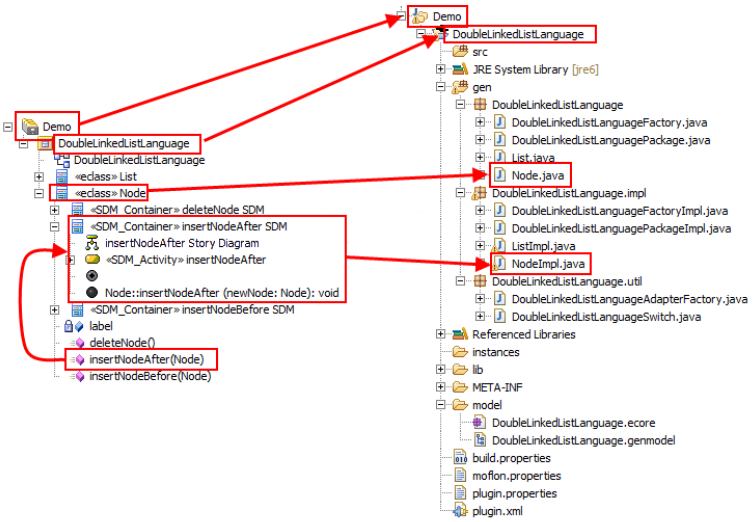

Figure 2.15: From EA to Eclipse

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-

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

[7] 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[8]. 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 `Foo-Impl.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

[8] Not to be mixed up with CVS or SVN repositories, although the idea of a source code "container" is the same here.

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. ≪ `SDM Container`≫ `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).

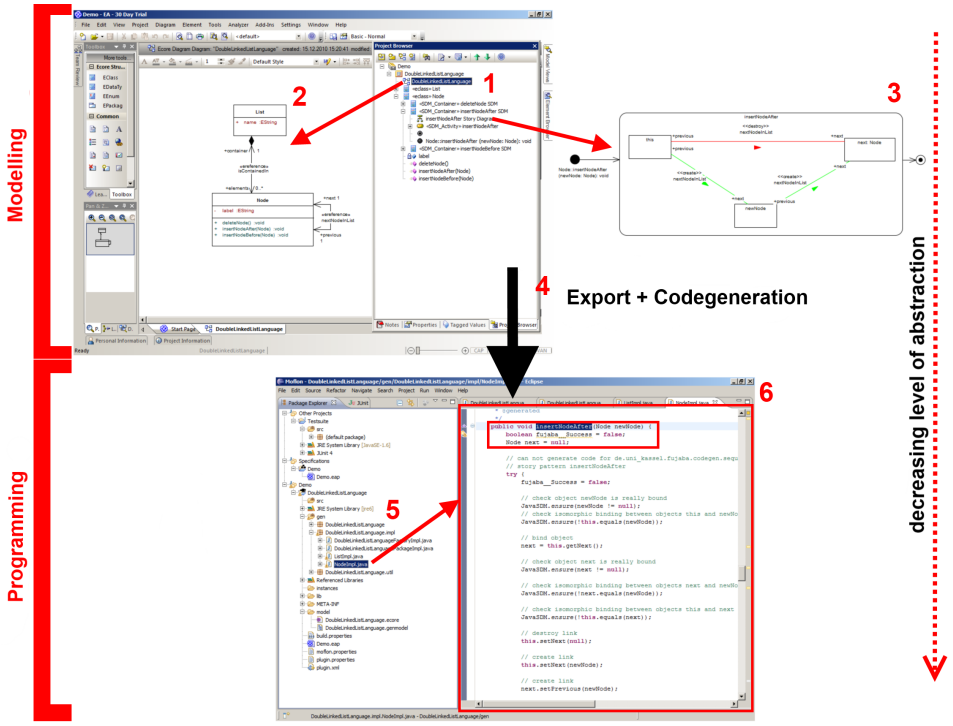

Figure 2.16: Overview

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

The toughest part of learning a new language is often building up a sufficient vocabulary. This is usually accomplished by repeating a long list of words again and again till they stick. A memory box is a simple but ingenious little contraption to support this tedious process of memorisation. As depicted in Fig. 3.1, it consists of a series of compartments or partitions usually of increasing size. The content to be memorised is written on a series of cards which are initially placed in the first partition. All cards in the first partition should be repeated everyday and cards that have been successfully memorised are placed in the next partition. Cards in all other partitions are only repeated when the corresponding partition is full and cards that are answered correctly are moved one partition forward in the box. Challenging cards that have been forgotten are treated as brand new cards and are always placed right back into the first partition regardless of how far in the box they had progressed. These "rules" are depicted by the green and red arrows in Fig. 3.1. The basic idea is to repeat difficult cards as often as necessary and not to waste time on easy cards which are only repeated now and then to keep them in memory. The increasing size of the partitions represents how words are easily placed in our limited short term memory and slowly move in our theoretically unlimited long term memory if practised often enough.

A memory box is an interesting system, because it consists clearly of a static structure (the box, partitions and their sizes, cards with their sides and corresponding content) and a set of rules that describe the dynamic aspects (behaviour) of the system. In the rest of the tutorial we shall build a complete memory box from scratch in a model-driven fashion and use it to introduce fundamental concepts in metamodelling and MDSD in general.

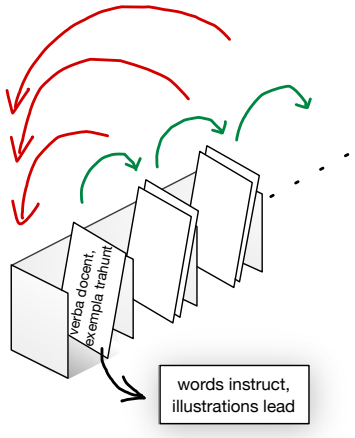

Figure 3.1: Possible *Concrete Syntax* of our Memory Box.

3.1 A Language Definition Problem?

Like in any area of study, metamodelling has its fair share of buzz words used by experts to communicate concisely. Although some concepts might seem quite abstract for a beginner, a well defined vocabulary is important so we know exactly what we are talking about.

The first step is understanding that metamodelling equates to language definition. This means that the task of building a system like our memory box can be viewed as defining a suitable language that can be used to describe the system. This language oriented approach has a lot of advantages including a natural support for product lines (individual products are valid members of the language) and a clear separation between platform independent and platform specific details.

Concrete Syntax

So what constitutes a language? The first question is obviously how the building blocks of your language actually "look" like. Is your language to be textual? Visual? This is referred to as the *Concrete Syntax* of a language and is basically an interface to end users who use the language. In the case of our memory box, Fig. 3.1 can be viewed as a possible concrete syntax. As we are however building a memory box as a software system, our actual concrete syntax will probably be composed of GUI elements like buttons, drop-down menus and text fields.

Grammar

Irrespective of how a language looks like, members of the language must adhere to the same set of "rules". For a natural language like English, this set of rules is usually called a *grammar*. In metamodelling, however, everything is represented as a graph of some kind and, although the concept

of a *graph grammar* is also quite well-spread and understood, metamodellers more often use a *type graph* that defines what types and relations constitute a language. A graph that is a member of your language must *conform to* the corresponding type graph for the language. To be more precise, it must be possible to type the graph according to the type graph, i.e., the types and relations used in the graph must exist in the type graph and not contradict the structure defined there. This way of defining membership to a language has many parallels to the class-object relationship in the Object Oriented paradigm and should seem very familiar for any programmer used to OO. This type graph is referred to as the *Abstract Syntax* of a language.

Graph Grammar
Type Graph
Abstract Syntax

Very often, one might want to further constrain a language, beyond simple typing rules. This can be accomplished with a further set of rules or constraints that members of the language must fulfil in addition to being conform to the type graph. These further constraints are referred to as the *Static Semantics* of a language.

Static Semantics

With these few basic concepts, we can now introduce a further and central concept in metamodelling, the *metamodel* (basically a simple class diagram). A metamodel defines not only the abstract syntax of a language but also some basic constraints (a part of the static semantics). Thinking back to our memory box, we could define the types and relations we want to allow, e.g., a box with partitions, cards, the box contains partitions that contain cards. Multiplicities are an example for constraints that are no longer part of the abstract syntax and belong to static semantics, but can nonetheless be expressed in a metamodel. For example, that a card can only be in one partition, or that a partition has only one next partition or none. More complex constraints that cannot be expressed in a metamodel are usually specified using an extra *constraint language* such as OCL (the Object Constraint Language). This goes beyond this tutorial however and we'll stick to metamodels without using an extra constraint language.

Metamodel
Constraint Language

A short recap: we have learnt that metamodelling starts with defining a suitable language. For the moment, we know that a language comprises a concrete syntax (how does the language look like), an abstract syntax (types and relations of the underlying graph structure), and static semantics (further constraints that members of the language must fulfil). Metamodels are used to define the abstract syntax and a part of the static semantics of a language, while *models* are graphs that conform to some metamodel (can be typed according to the abstract syntax and adhere to the static semantics).

Model

This tutorial is meant to be be hands-on so enough theory! Lets define, step-by-step, a metamodel for our memory box using our tool eMoflon.

3.2 Abstract Syntax and Static Semantics

Switch to EA, choose `Demo` and click on the button `Add a Package` as depicted in Fig. 3.2.

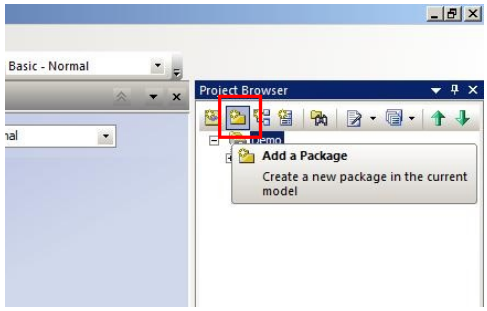

Figure 3.2: Add a new package to `Demo`.

In the dialogue that pops up (Fig. 3.3), choose `Class View`, enter `MemoryBoxLanguage` as the name of the new package and click `OK`.

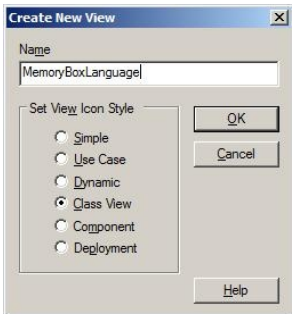

Figure 3.3: Enter the name of the new package.

In your EA workspace the `Project Browser` should now look like Fig. 3.4.

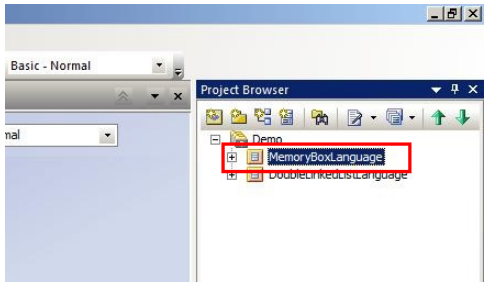

Figure 3.4: State after creating the new package.

Now click the button `New Diagram` (Fig. 3.5).

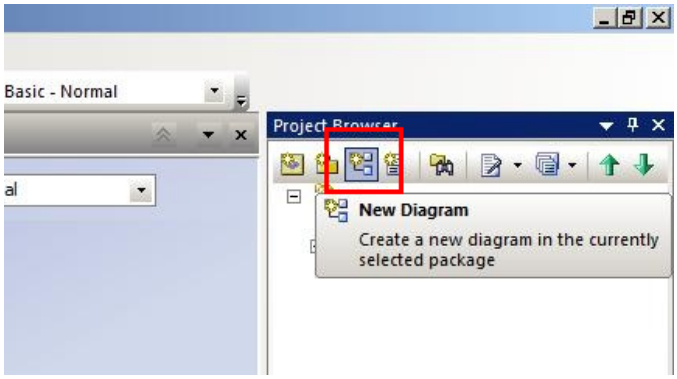

Figure 3.5: Add a diagram.

In the dialog that pops up (Fig. 3.6), choose `Ecore Diagram` and `OK`.

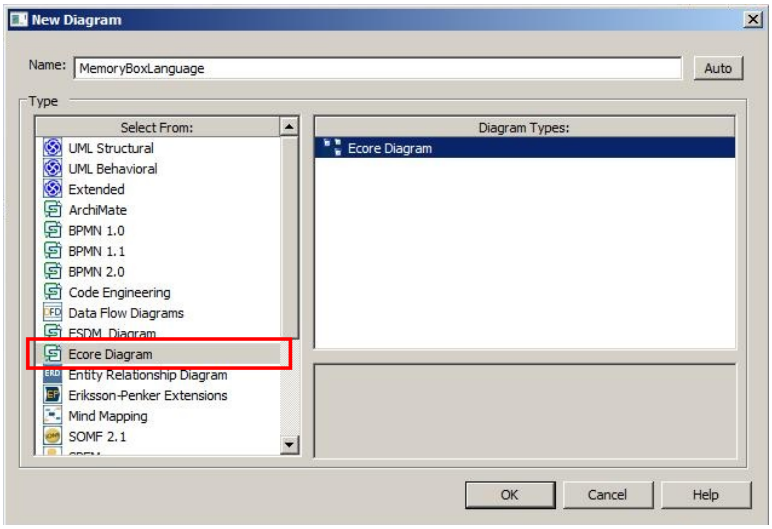

Figure 3.6: Choose type of diagram.

In analogy to the "everything is an object" principle in the OO paradigm, in metamodelling, everything is a model. This principle is called *Unification* and has a lot of advantages. If everything is a model, a metamodel that defines (at least a part of) a language must be a model itself. This means that it conforms to some *meta-metamodel* which defines a *(meta)modelling language* or *meta-language*. For metamodelling with eMoflon, we support *Ecore* as a modelling language and it defines types like `EClass` and `EReference`, which we will be using to specify our metamodels. Other modelling languages include MOF, UML and Kermeta.

Unification

Meta-metamodel
Meta-Language
Modelling Language

After creating the new diagram, your `Project Browser` should now resemble Fig. 3.7. Double-click the newly created diagram to ensure that it is open.

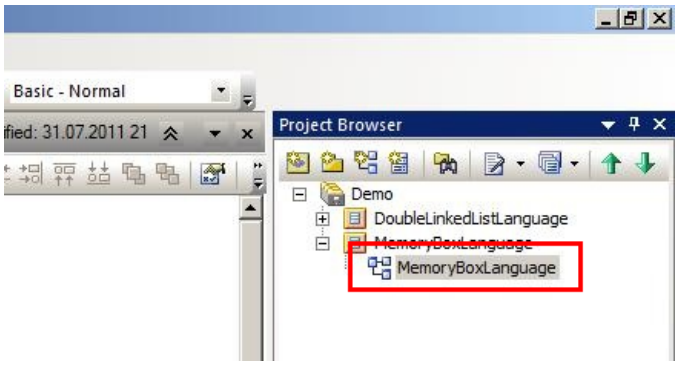

Figure 3.7: State after creating diagram.

To the left of the workbench in EA, a *Toolbox* should have appeared containing the types available in Ecore for metamodelling (Fig. 3.8). Click on `EClass` and click in the open diagram (the main window in EA).

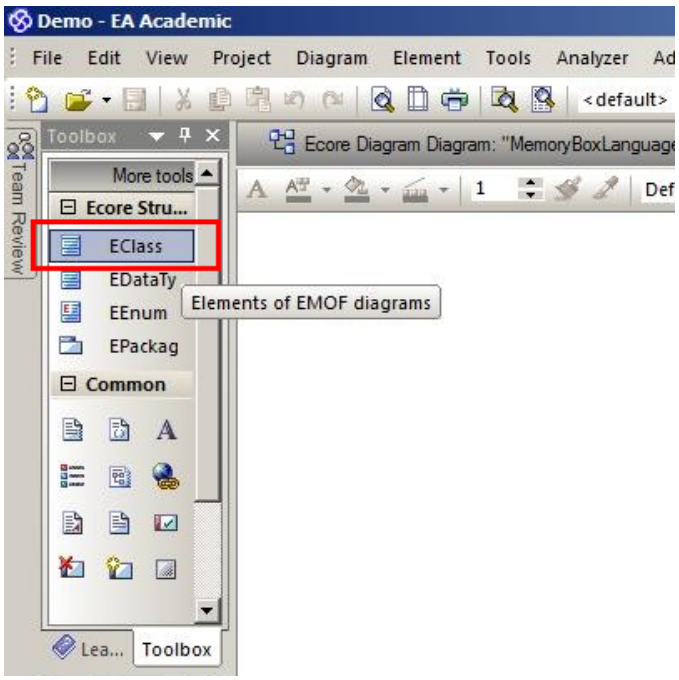

Figure 3.8: Create an EClass.

In the dialogue that pops-up, enter `Box` as the name of the class and click `OK` (Fig. 3.9). This dialogue can always be invoked by double-clicking the class and contains many other properties we'll be looking into later in the tutorial. In general, a similar "properties" dialogue can be opened in the same fashion for almost every element in EA.

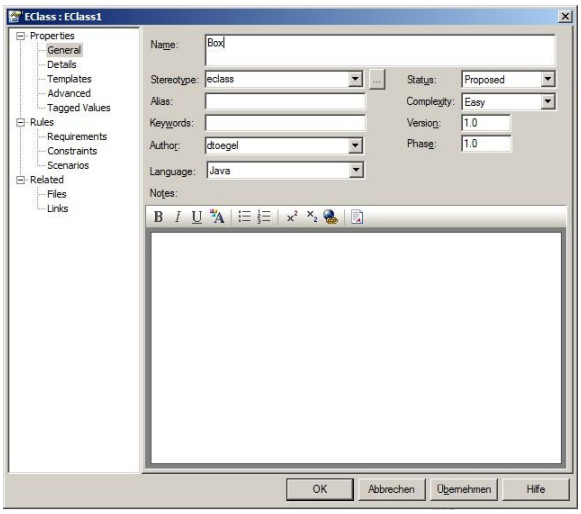

Figure 3.9: Enter properties of EClass.

After creating `Box`, your EA workspace should resemble Fig. 3.10.

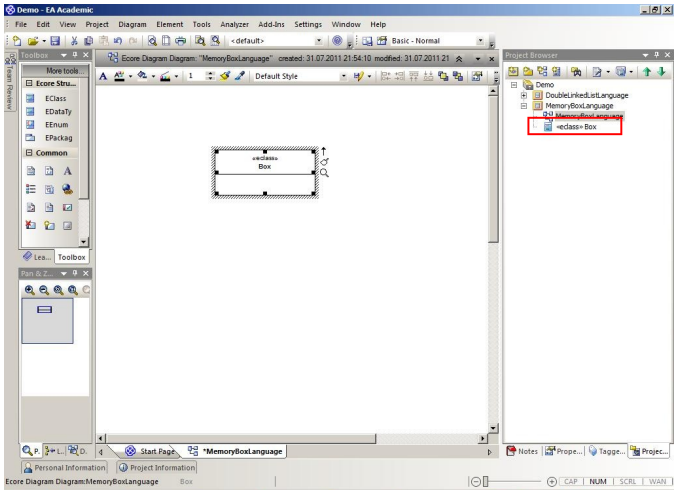

Figure 3.10: State after creating `Box`.

Now create `Partition` and `Card` in the same way, till your workspace resembles Fig. 3.11. These are the main classes for our memory box.

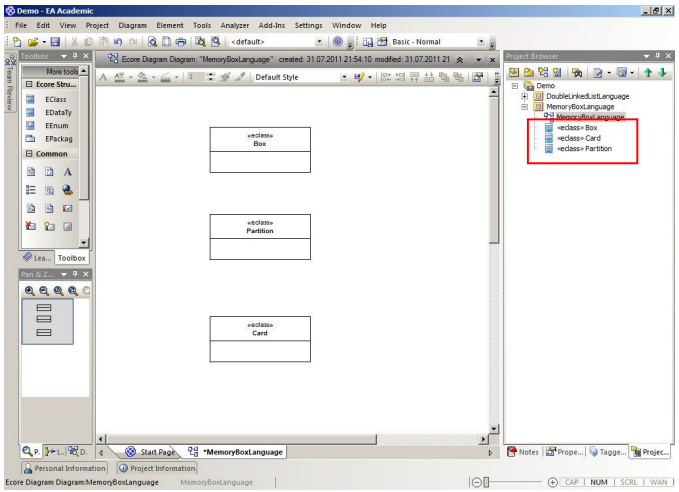

Figure 3.11: Main classes in our metamodel.

Now choose `Box`, right-click to call up the context menu and choose `Attributes...` (Fig. 3.12).

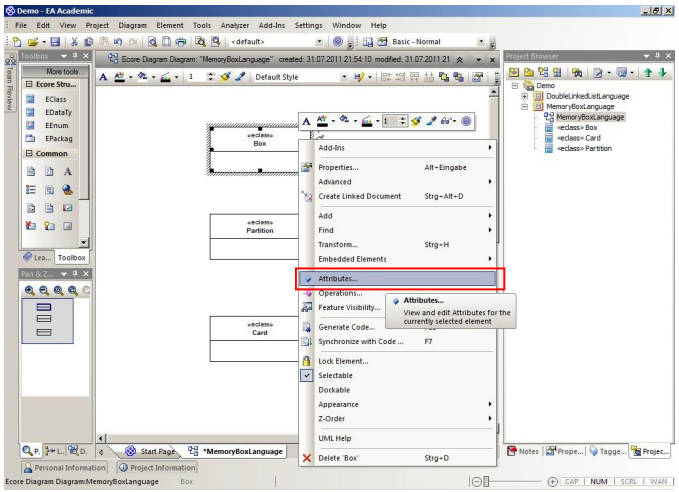

Figure 3.12: Context Menu for a class.

In the dialogue that pops-up, enter `name` as the name of the attribute, choose `EString` as its type and press `Save` (Fig. 3.13). A new attribute for the same class can be added by choosing `New`.

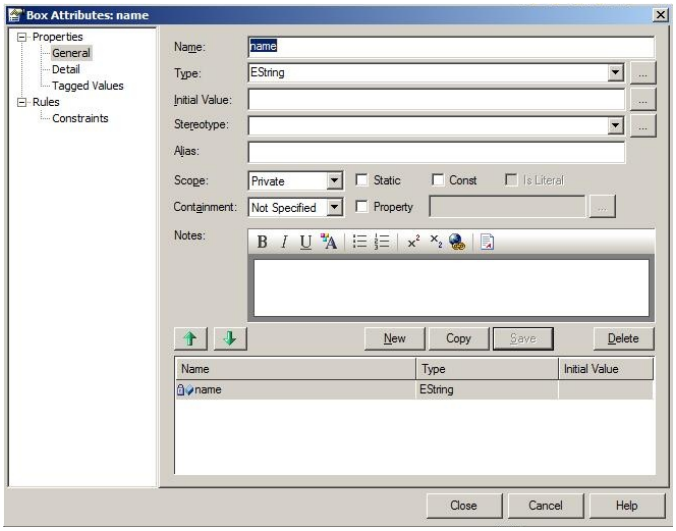

Figure 3.13: Adding attributes to a class.

Add attributes to the other classes till your workspace resembles Fig. 3.14.

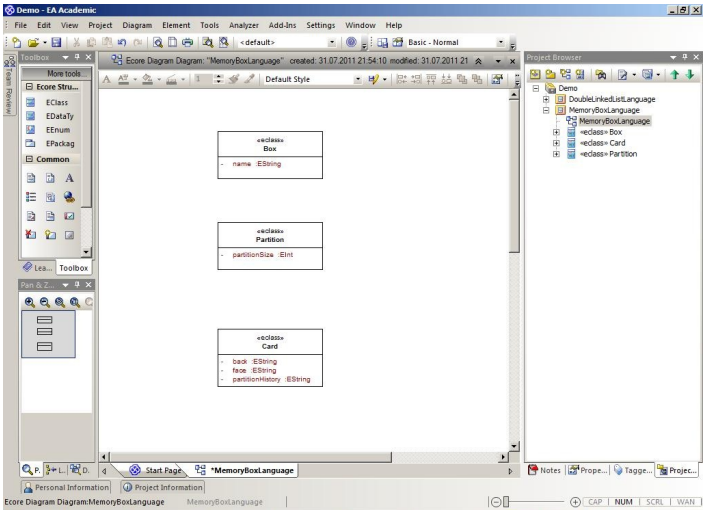

Figure 3.14: Main classes with attributes.

Now choose `EPackage` from the toolbox and add it to the current diagram just like how we added EClasses. Enter `facade` as the name of the package.

Ecore supports packages that can be used to structure and group classes in a metamodel. In our case, we need a util class that implements helper methods for our memory box. These methods will be implemented by hand in Java and the util class thus represents a kind of interface or "facade" between our model and hand-written code. We shall soon see how our Eclipse Plugin offers extra support if one follows this naming convention for packages containing hand-written code.

To add a class to our new package, first of all create a new diagram in the package by choosing the `facade` subpackage and selecting `New Diagram` in the Project Browser (Fig. 3.15).

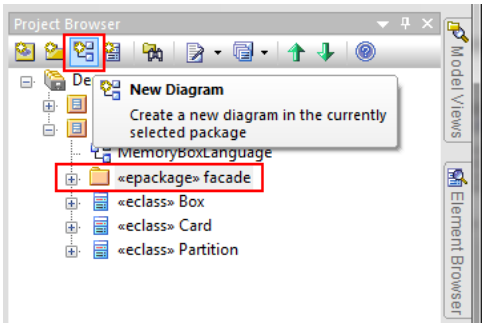

Figure 3.15: Add a class to the package.

In the dialogue that pops-up, choose `Ecore Diagram` and confirm with `OK`. In the new diagram, create a new class and enter `MemoryBoxUtil` as its `name`. You can switch between diagrams by choosing the tabs at the bottom of the screen or by pressing `alt` + $\Leftarrow$ or $\Rightarrow$ (Fig. 3.16).

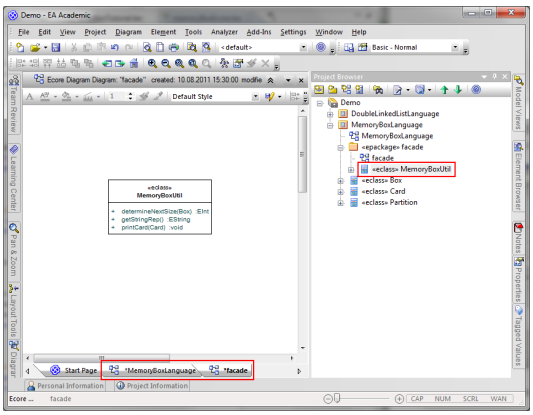

Figure 3.16: Main and subdiagrams in EA.

Your workspace should now resemble Fig. 3.17. Any subpackage like `facade` can contain diagrams that can be created and added using the Project Browser. In this way an arbitrary nesting of packages and diagrams is possible.

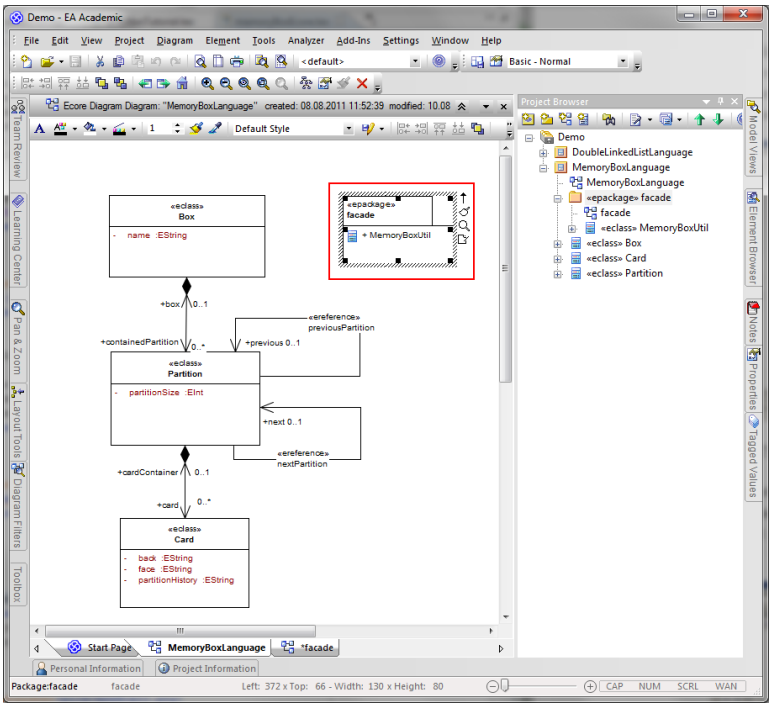

Figure 3.17: Workspace after adding package and util class.

A fundamental gesture in EA is *Quick Link*. Quick Link is used to create links between elements in a context sensitive manner. To use Quick Link, choose an element and note the little black arrow in its top-right corner (Fig. 3.18).

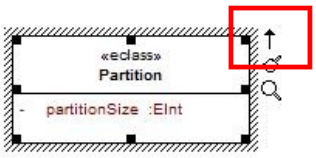

Figure 3.18: Quick Link is a central gesture in EA.

Now click on the black arrow and pull to another element you wish to "quick link" to. In this case quick link from `Partition` to `Box`. In the context-menu that pops-up, choose `EReference`. (Fig. 3.19).

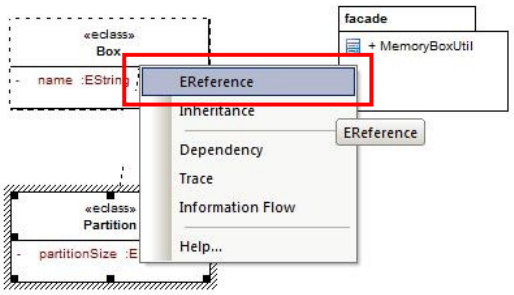

Figure 3.19: Create a reference via Quick Link.

In the dialogue that pops-up (Fig. 3.20), the direction of the reference can be set. The default is bidirectional and this is ok for our `Box`⇔`Partition` connection. A `Name` can also be entered, which is only used for documentation purposes and is not relevant for codegeneration.

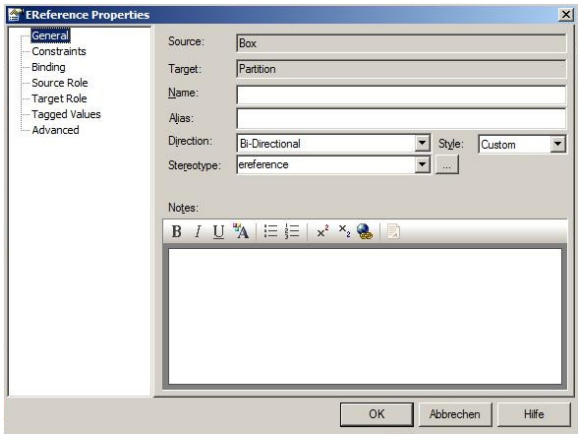

Figure 3.20: Enter properties of the reference.

In the same dialogue choose `Source Role` and enter the values in Fig. 3.20 to set the properties for the "source" end of the reference (the `Box` role). Important is a name for the role (`box`), the `Multiplicity`, `Aggregation` and `Navigability`. Repeat the process for the `Target Role`.

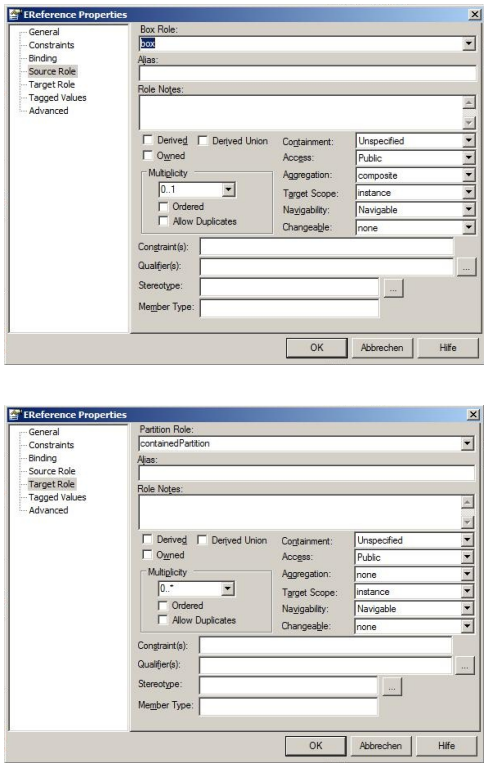

Figure 3.21: Enter properties for source and target of reference.

Navigable ends are mapped to class attributes with getters and setters in Java and therefore *must* have a specified name and multiplicity for successful codegeneration. Corresponding values for non-navigable ends can be regarded as additional documentation and do not have to be specified.

The multiplicity of a reference controls if the relation is mapped to a Java Collection (`*`, `1..*`, `0..*`), or a single valued class attribute (`1`, `0..1`).

In Ecore, the aggregation values of a reference can either be `none` or `composite`. Composite means that the current role is that of a *container* for the opposite role. In our case for example, `box` is a container for `partitions`. This has a series of consequences: (1) every element must have a container, (2) an element cannot be in more than one container at the same time, and (3) a container's contents are deleted together with the container. Non-composite (`none`) means that the current role is not that of a container and the rules for containment do not hold (reference is a simple "pointer").

If you've done everything right, your workspace should now resemble Fig. 3.22 with a relation between `Box` and `Partition`.

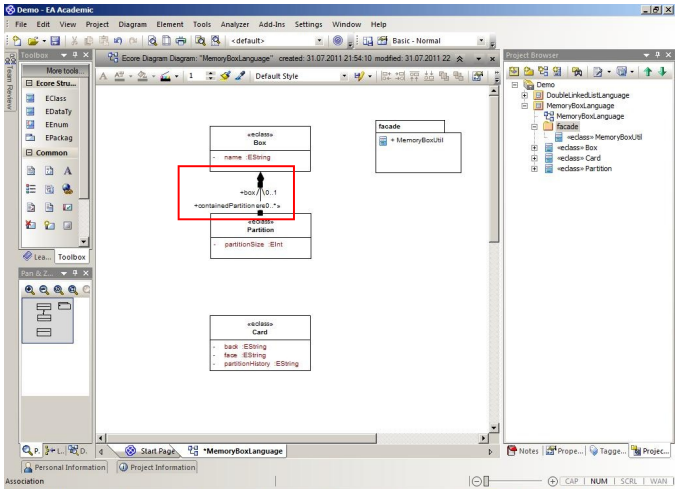

Figure 3.22: `Box` contains `Partitions`.

Create a bidirectional reference[1] between `Partition` and `Card` and two unidirectional self-references for `Partition` according to Fig. 3.23[2].

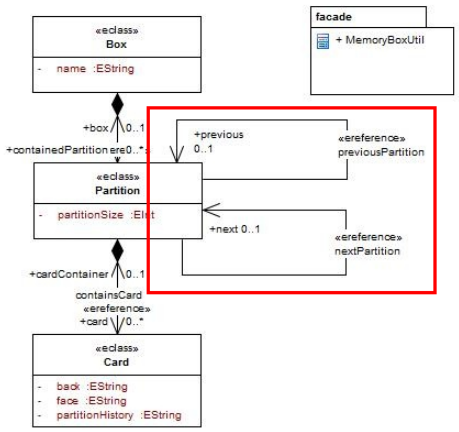

Figure 3.23: All relations in our metamodel.

[1] To be precise, *all* references in Ecore are actually unidirectional. A "bidirectional" reference in our metamodel is in reality mapped to two `EReferences` that are opposites of each other. We however believe it is simpler to handle these pairs as single references and prefer this concise concrete syntax.

[2] If you have difficulties deciphering the role names and other details in the screen shot please refer to Fig. 3.28 for a better diagram of the metamodel.

Every system has, in addition to its static structure, certain dynamic aspects that describe the system's behaviour and how it evolves over time or reacts to external stimulus. In a language, these rules that govern the dynamic behaviour of a system are referred to collectively as the *Dynamic Semantics* of the language. Although these rules can be defined as a set of separate *Model Transformations*, we take a holistic approach and advocate integrating the transformations directly in the metamodel as operations. This fits nicely to the object oriented paradigm and is quite natural in many cases. In the next few steps we shall define the *signatures* of some operations for our memory box. We will of course use SDMs to *implement* the methods later.

Dynamic Semantics

Right-click `Partition` to invoke the context-menu depicted in Fig. 3.24 and choose `Operations...`.

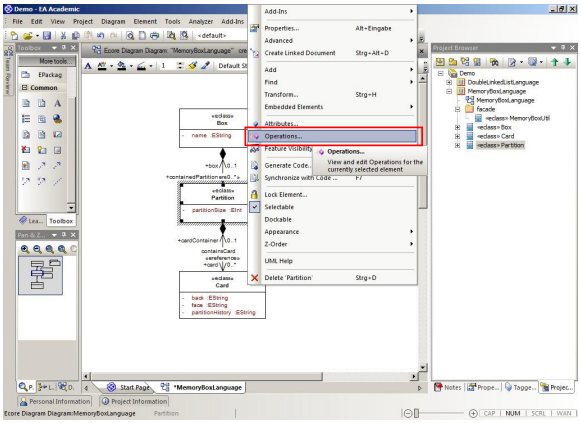

Figure 3.24: Add an operation.

In the dialogue that pops-up (Fig. 3.25), enter `empty` as the `Name` of the operation, leave the `Return Type` as `void`. Press `Save`.

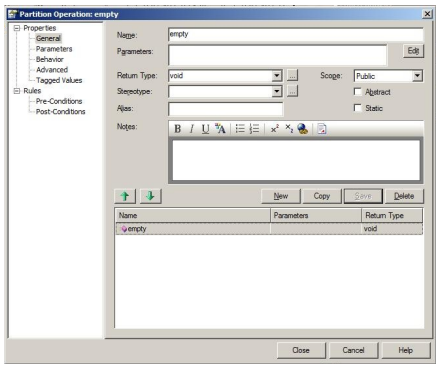

Figure 3.25: Properties for operation.

In the same dialogue, press `New` to add further operations and enter the values in Fig. 3.26. Parameters can be added by pressing `Edit` and entering the name and choosing the type of each Parameter in a separate dialogue.

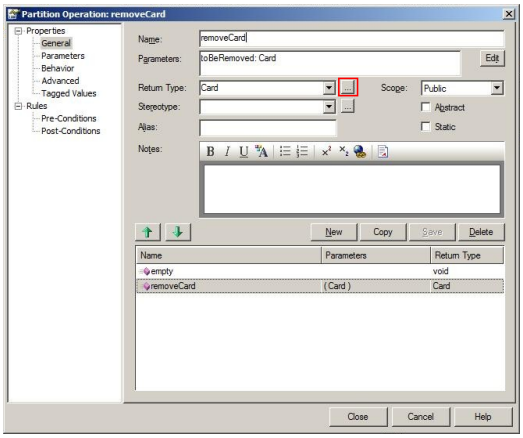

Figure 3.26: Parameters and Return Type.

Repeat the process for the values in Fig. 3.27. The `Return Type` can be chosen via the drop-down menu for primitives (`EBoolean`), or via the `...` button for types in the metamodel (`Card`).

> Please note: Non-primitive types *must* be chosen via the `...` button that allows you to browse for the corresponding elements in your project. Just typing them unfortunately won't work due to EA API restrictions!

If you've done everything right, your dialogue should now contain three methods `check`, `empty`, and `removeCard` with corresponding parameters and return types as in Fig. 3.27.

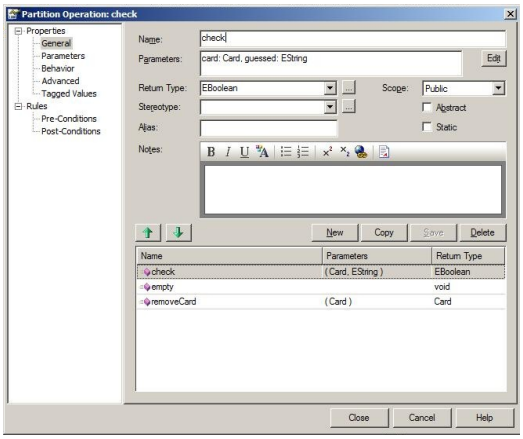

Figure 3.27: All operations in `Partition`.

Add all operations analogously for `Box`, `Card` and `MemoryBoxUtil`, so that your metamodel closely resembles Fig. 3.28.

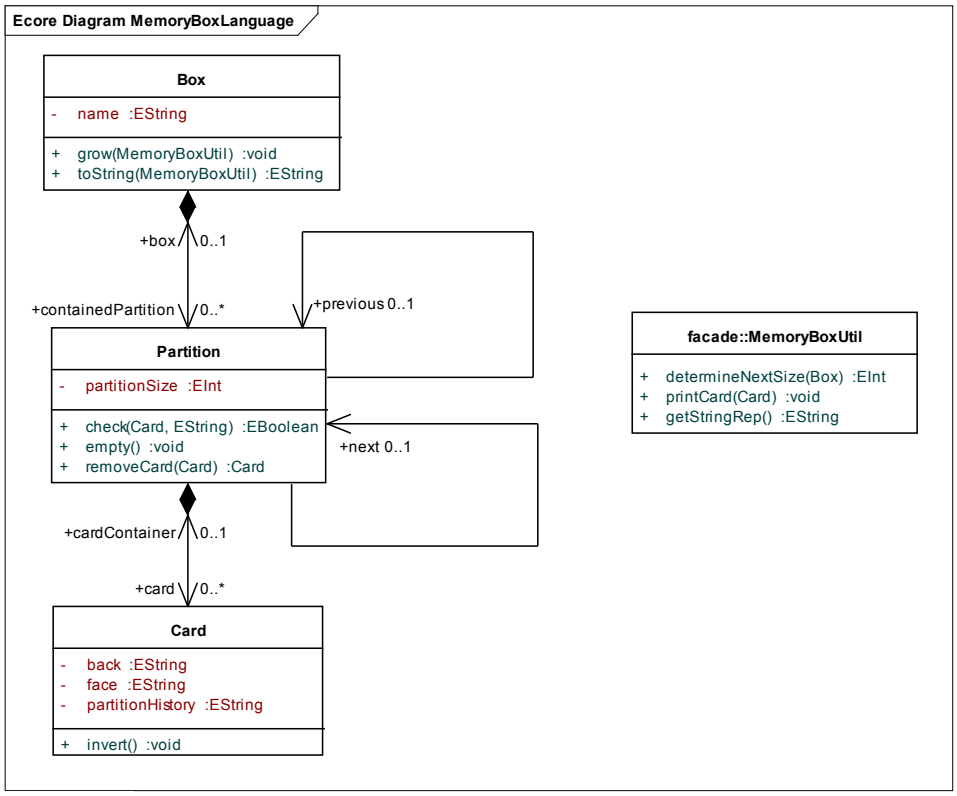

Figure 3.28: Complete metamodel for our memory box.

Lets take a step back and review our metamodel. We have modelled a `Box` that contains arbitrary many `Partition`s. A `Partition` in the `Box` has a `next` and `previous` `Partition` that can be set or not. Finally, `Partition`s contain `Card`s.

A `Box` has a `name`, and can be extended by calling `grow`. A `Box` can print out its contents via `toString`. We'll see later in the tutorial why these two methods need our `MemoryBoxUtil` as a parameter.

The main method of the memory box is `Partition::check` that takes a `Card` and the user's guess as an `EString` and returns `true` or `false` depending on if the guess was correct or not. A `Partition` can also `empty` itself of all `Card`s, or `remove` a particular `Card`. Last but not least, a `Partition` has a `partitionSize` that can be used to indicate that the `Partition` is full and is ready to be revised.

A `Card` contains the actual content to be learnt as a question on the card's `face` and the answer on the card's `back`. A `Card` also maintains a `partitionHistory` which can be used to keep track of how often a `Card` has been answered correctly/wrongly. This might indicate how difficult the `Card` is for a specific user. When learning a language, it makes sense to be able to swap the target and source langauge and this is supported by `Card` via `invert` (turns the card around).

Now try to export the metamodel for codegeneration in Eclipse. To do this right-click on `MemoryBoxLanguage` and choose "Add-In/MOFLON::Ecore Addin/Export Selection to Workspace". Then switch to your Eclipse workspace and refresh the metamodel workspace.

If you have done everything right, a new project `MemoryBoxLanguage` should be created in the `Demo` working set in your Eclipse workspace. If this is not the case please ensure that your metamodel is identical with Fig. 3.28. If you believe everything is correct and things still don't work then feel free to contact us at `contact@moflon.org`. If code is generated successfully, take a look at all the stuff that has been generated under `/gen`, especially the default implementation for all methods that just throws an `OperationNotSupported` exception. We shall see later in the tutorial that the EMF codegenerator actually supports merging hand-written implementations of methods with generated code. With eMoflon however, we can also model a large part of the dynamic semantics and only need to implement small helper methods for e.g. string manipulation by hand.

Let's move on and model the dynamic behaviour of our memory box!

3.3 Dynamic Semantics with SDM

The core idea when modelling behaviour is to regard dynamic aspects of a system (let's call this a model as from now on) as bringing about a change of state. This means a model in state S can evolve to state S^* via a transformation $\Delta : S \stackrel{\Delta}{\rightarrow} S^*$. In this light, dynamic or behavioural aspects of a model are synonymous with *model transformations*, and the dynamic semantics of a language equate simply to a suitable set of model transformations. This approach is once again quite similar to OO where objects have state and can *do* things via methods that manipulate their state.

So how do we model model transformations? There are quite a few possibilities. We could employ a suitably concise imperative programming language with which we simply say in a step-by-step manner how the system morphs. There actually exist quite a few very successful languages and tools in this direction.

But isn't this almost like just programming directly in Java? There must be a better way to do this... From the relatively mature area of graph grammars and graph transformations we take a *declarative* and *rule-based* approach. Declarative in this context means that we do not want to specify exactly how and in what order changes to the model must be carried out to achieve a transformation. We just want to say under what conditions the transformation can be executed (precondition), and the state of the model after executing the transformation (post condition). The actual task of going from precondition to postcondition should be taken over by a transformation engine and all related details are basically regarded as a black box.

Ok - so a model transformation is of the form $(pre, post)$. Inspired by string grammars, let's call this black box transformation a *rule*, and consequently the precondition the left-hand side of the rule L and the postcondition the right-hand side R.

A rule $r : (L, R)$ can be *applied* to a model (a typed graph) G by:

1. Finding an occurrence of the precondition L in G via a *match* m,

2. Cutting out $Destroy := (L \setminus R)$ i.e., the elements that are present in the precondition but not in the postcondition are to be deleted, from G to form $(G \setminus Destroy)$ and

3. Pasting $Create := (R \setminus L)$ i.e., new elements that are present in the postcondition but not in the precondition and are to be created, into the hole in $(G \setminus Destroy)$ to form a new graph $H = (G \setminus Destroy) \cup Create$.

Rule application is denoted as $G \overset{r}{\Rightarrow} H$ and is depicted in Fig. 3.29.

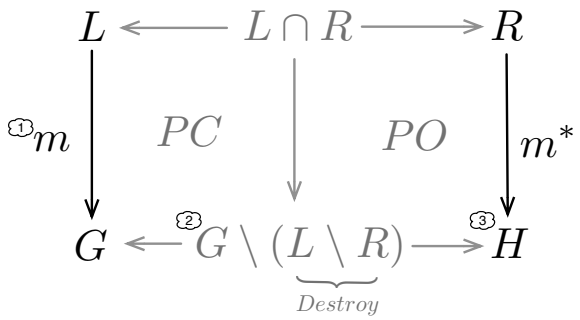

Figure 3.29: Applying a rule $r : (L, R)$ to G to yield H

(1) is called *graph pattern matching*, (2) is called building a *push-out complement* $PC = (G \setminus Destroy)$, so that $L \cup (G \setminus Destroy) = G$ and (3) is called building a *push-out* $PO = H$, so that $(G \setminus Destroy) \cup R = H$. A push-out is a generalised union defined on typed graphs. As we are dealing with graphs here, it is not such a trivial task to define (1) – (3) in precise terms with conditions when a rule can be applied and not, and there exists substantial theory with exactly that goal. As this formalisation of rule application involves two push-outs: one (deletion) when cutting out $Destroy := (L \setminus R)$ from G to yield $(G \setminus Destroy)$, and one (creation) when inserting $Create := (R \setminus L)$ in $(G \setminus Destroy)$ to yield H, this is referred to as a *double push-out*. We won't go into further details in this tutorial, but the interested reader can refer to [Ref] for the exciting details.

Now that we know what rules are, let's take a look at a simple example for our memory box. How would a rule look like for moving a card from one partition to the next? Fig. 3.30 depicts the rule *moveCard*.

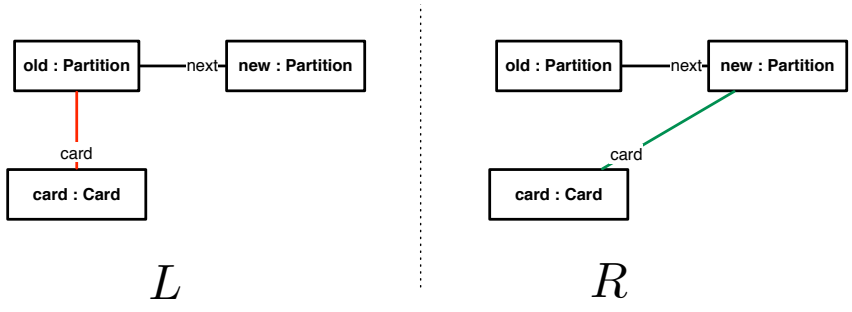

Figure 3.30: Rule *moveCard* as a graph transformation rule.

As already indicated by the colours used for *moveCard* we employ a compact representation of rules that is formed by merging (L, R) into a single *story pattern* composed of $Destroy := (L \setminus R)$ in red, $Retain := L \cap R$ in black, and $Create := (R \setminus L)$ in green (Fig. 3.31).

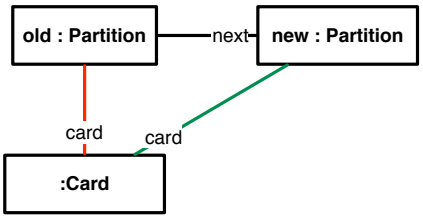

Figure 3.31: Compact representation of *moveCard* as a Story Pattern.

As we shall see in a moment, this representation is quite intuitive and one can just forget the details of rule application and think in terms of what is to be deleted, retained and created. Applying *moveCard* to a memory box according to steps (1) – (3) is depicted in Fig. 3.32.

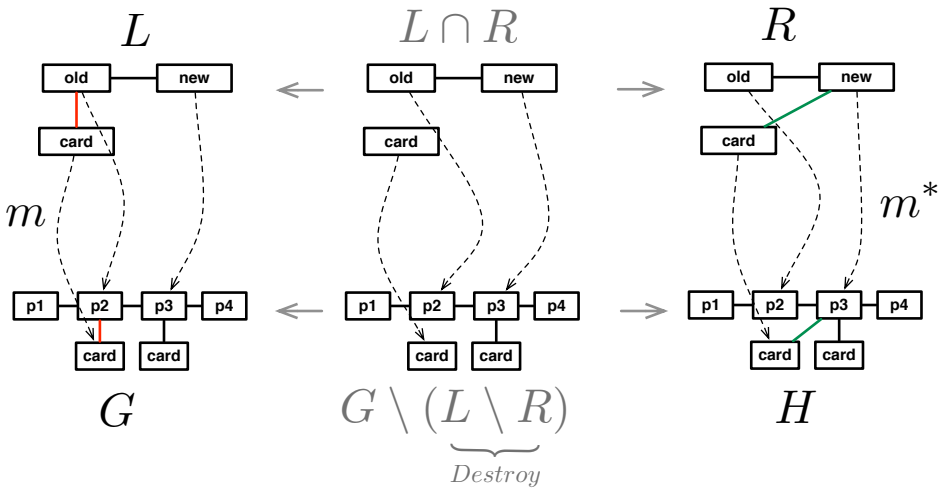

Figure 3.32: Applying *moveCard* to a memory box.

One last thing before we continue with our memory box; individual rules still have to be applied in a suitable sequence to realise complex model transformations that consist of many steps. This is realised with simplified activity diagrams, where a single activity node is a pattern as discussed above, and activity edges join nodes to form a control flow. This can be viewed as two layers: an imperative layer to define the top-level control flow via activity diagrams (if-else statements, loops etc), and a pattern layer consisting of a story pattern in each activity node that specifies, via a graph transformation rule, how the model is to be manipulated in that step.

Enough theory! Grab your mouse and let's get cracking with SDMs...