# An Introduction to Metamodelling and Graph Transformations

## *with eMoflon*

Version 1.3

For your convenience, this document includes a copy of the *GNU General Public License* starting from page vii.

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

*The eMoflon team*
Darmstadt, Germany (September 2011)

# 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 Extension for Enterprise Architect

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-Extension (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

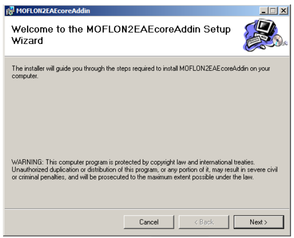



Figure 2.2: Install our extension for EA

## 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/`

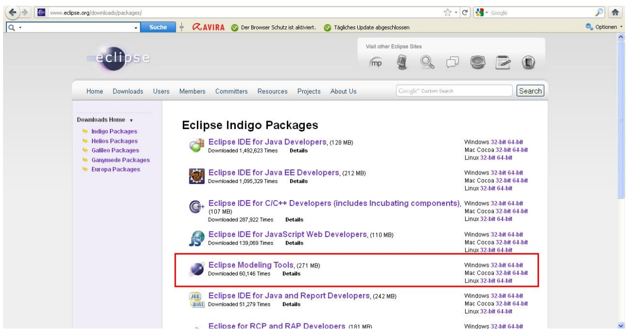



Figure 2.3: Download Eclipse Modeling Tools.

- 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 3.7 (Indigo).

[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).

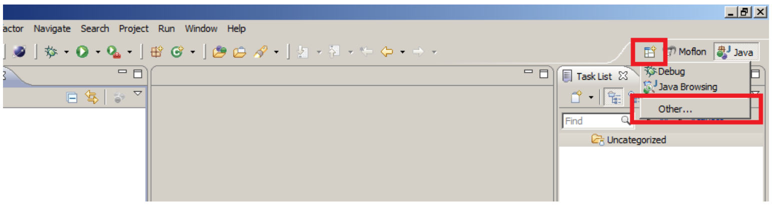



Figure 2.4: Choose the Moflon Perspective.

- ▶ 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!).

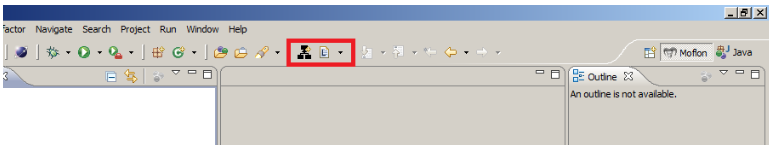



Figure 2.5: Eclipse New Metamodel

- ▶ 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[6]. Please do not rename, move or delete anything.

[5] A path given as “foo/bar” indicates how to navigate in a series of menus and submenus.
[6] At the moment, the project contains only the EA project file.

- 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.

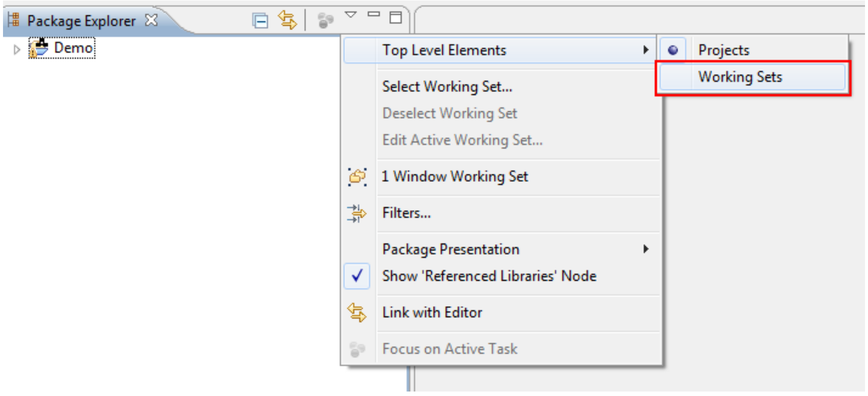



Figure 2.6: Choose Working Sets as Top Level Elements.

- Open the newly created project and replace the "Demo.eap" file with the Demo.eap that you will find in the `eMoflonTutorial.zip` file provided together with this tutorial.

- Double click "Demo.eap" to start EA. Please choose "Ultimate" when starting EA for the first time.

- In EA, choose "Extensions/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.

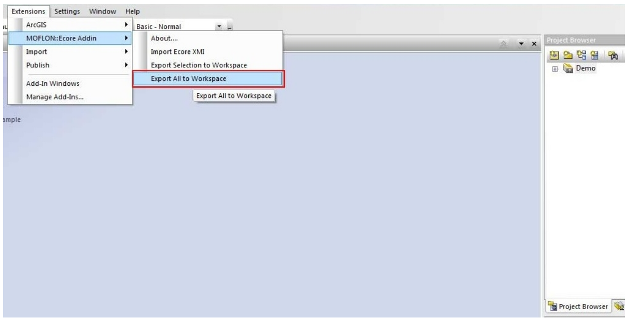


Figure 2.7: Export from EA using our extension

- 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 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.

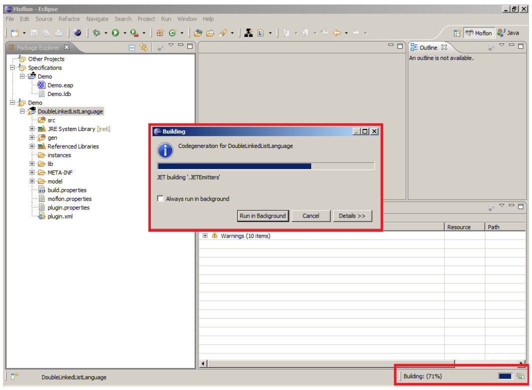


Figure 2.8: Automatically building the workspace after a refresh.

## 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 `eMoflonTutorial.zip` provided with this tutorial.

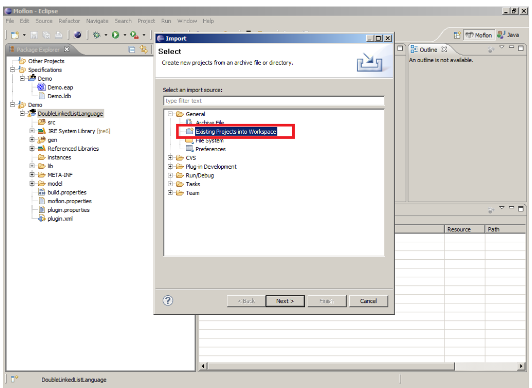


Figure 2.9: Import our Testsuite as an existing project.

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

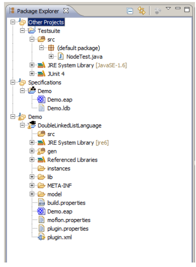


Figure 2.10: Workspace in Eclipse.

- 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!

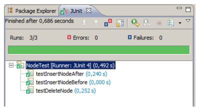


Figure 2.11: All's well that ends well. . .

## 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.

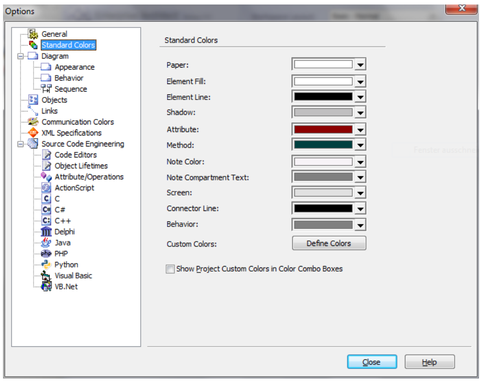


Figure 2.12: Our choice of standard colours for diagrams in EA.

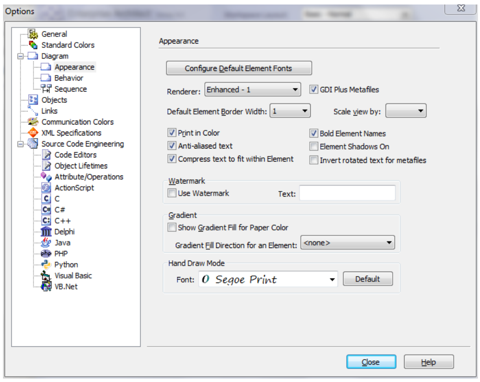


Figure 2.13: Our choice of the standard appearance for model elements in EA.

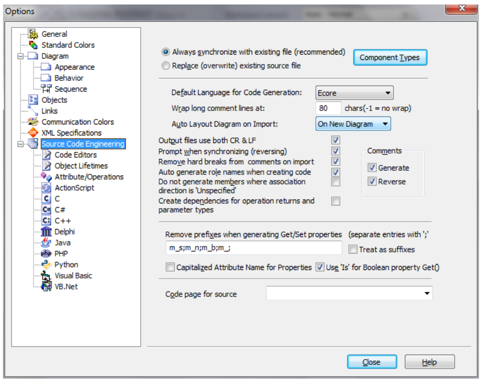


Figure 2.14: Make sure you set the standard language to Ecore.

In your EA "workspace", actually referred to as an *EA project*[7], take a careful look at the project structure: The root node `Demo`[8] 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 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`.

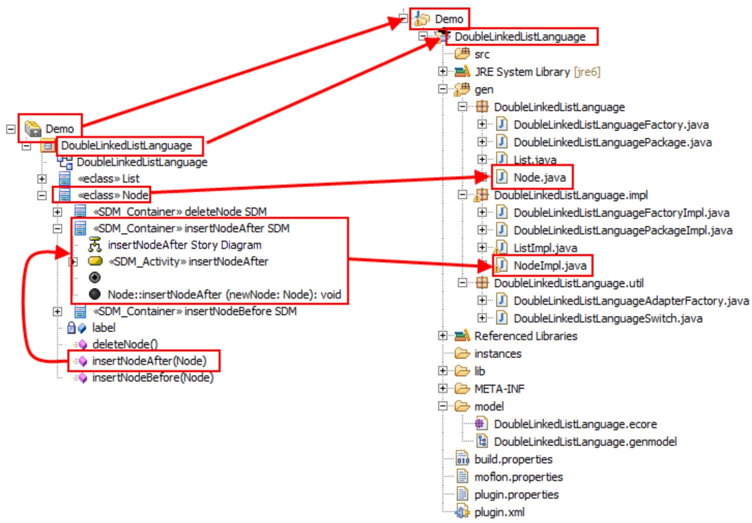



Figure 2.15: From EA to Eclipse

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-

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

[8]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 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* and this is the reason why we refer to such projects as repository projects[9]. 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 code generator. 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

[9]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 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 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 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).

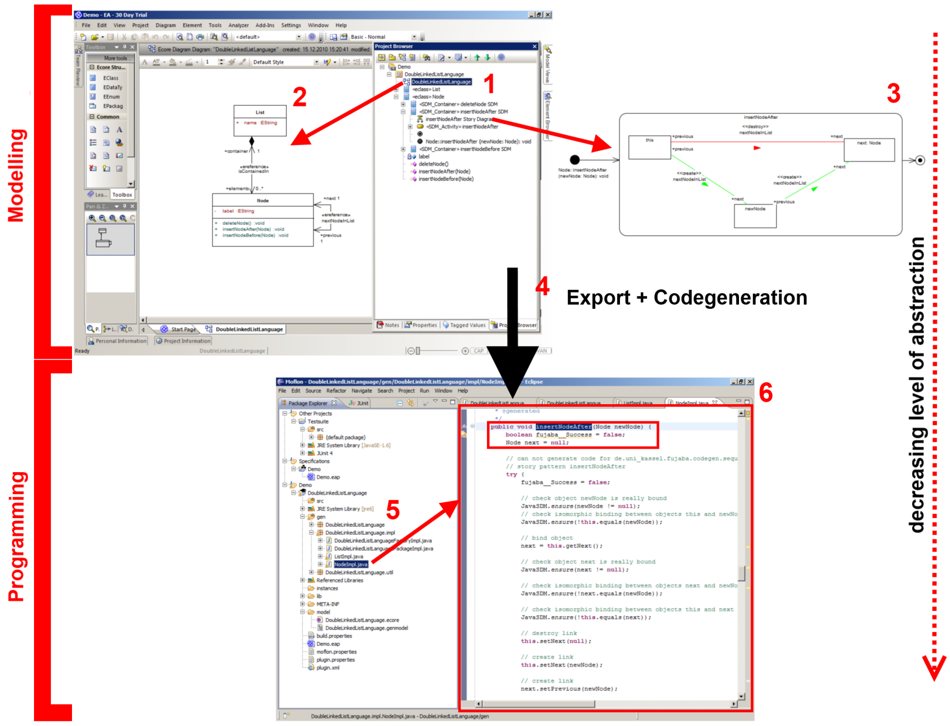



Figure 2.16: Overview

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.

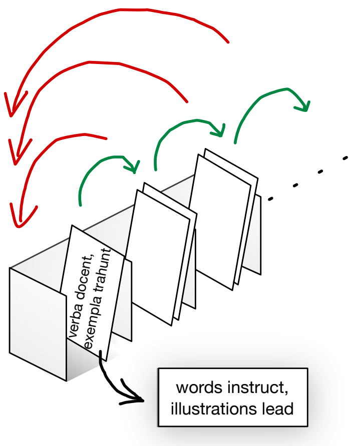



Figure 3.1: Possible *Concrete Syntax* of our Memory Box.

## 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.

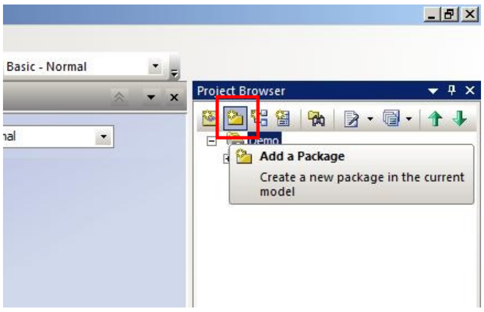



Figure 3.2: Add a new package to `Demo`.

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

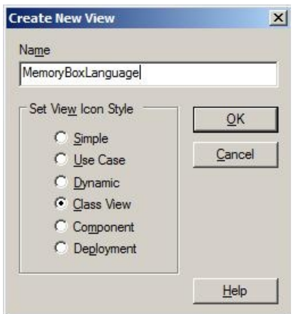



Figure 3.3: Enter the name of the new package.

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

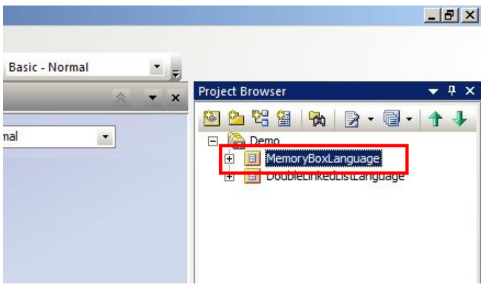



Figure 3.4: State after creating the new package.

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

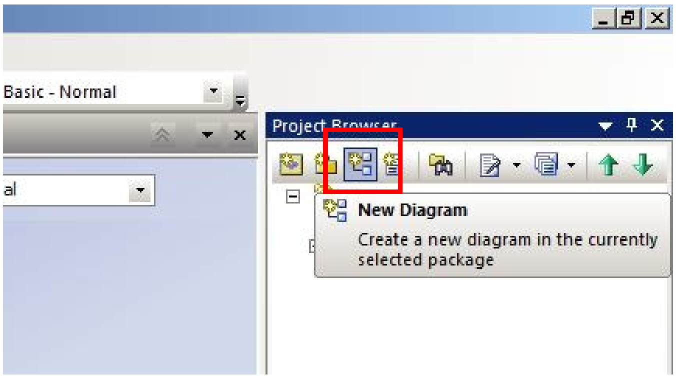



Figure 3.5: Add a diagram.

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

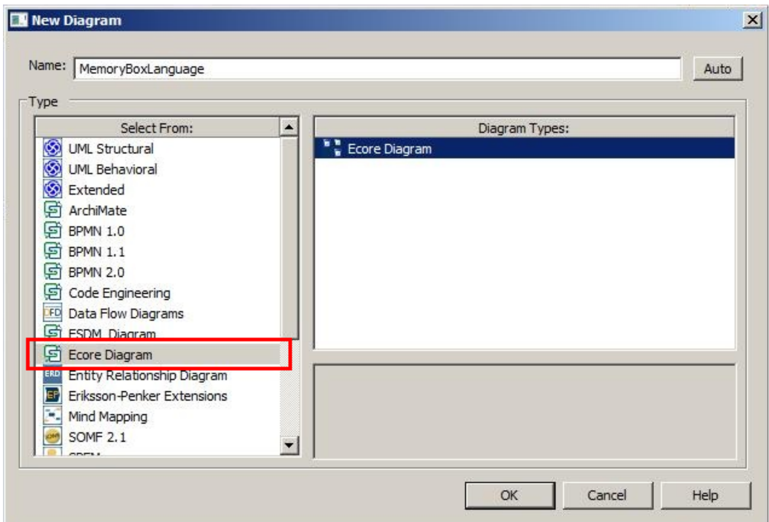



Figure 3.6: Choose type of diagram.

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.

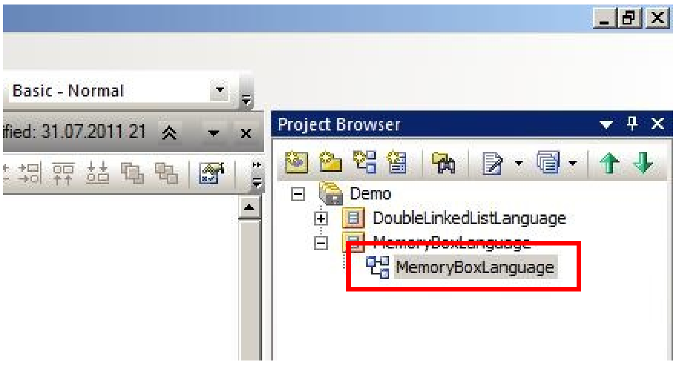



Figure 3.7: State after creating diagram.

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).

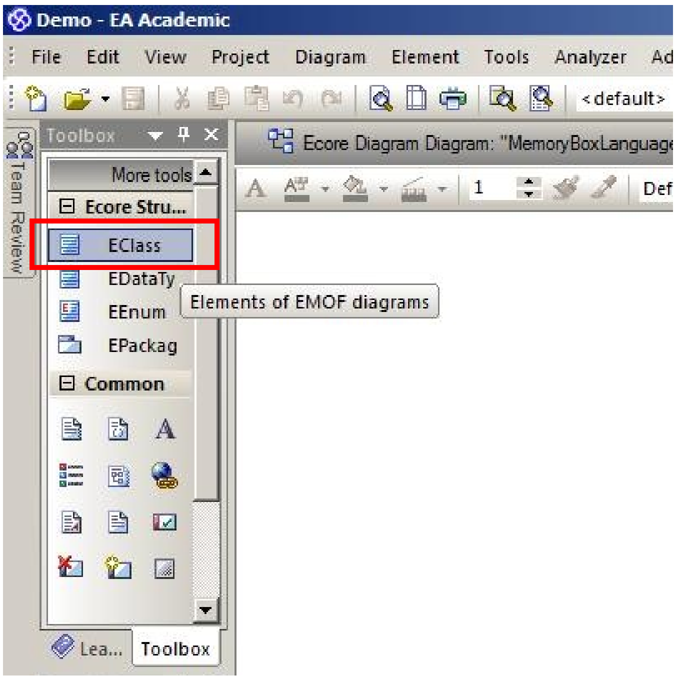



Figure 3.8: Create an EClass.

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.

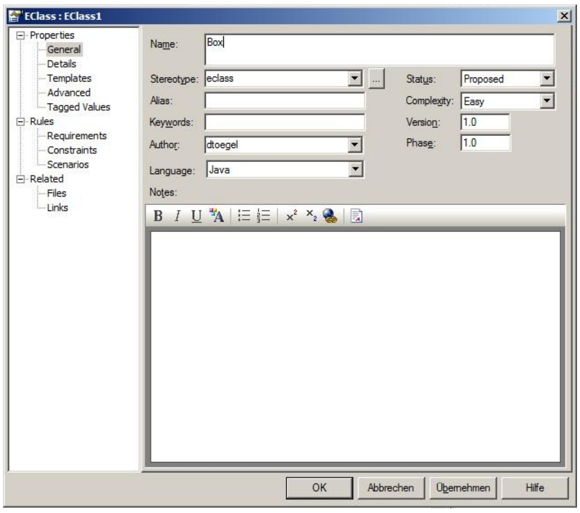



Figure 3.9: Enter properties of EClass.

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

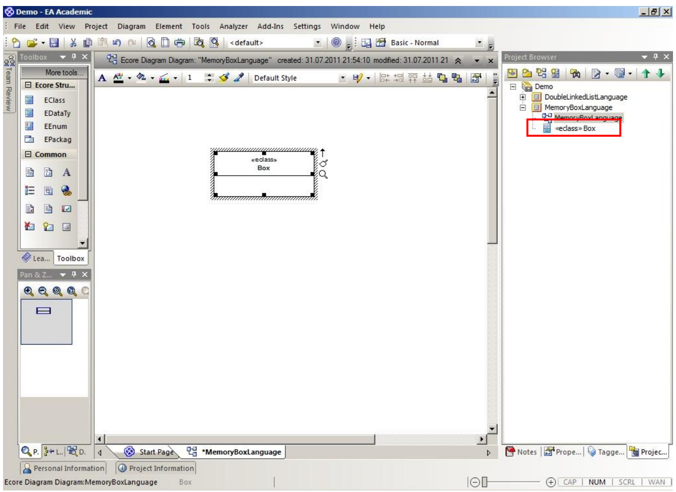


Figure 3.10: State after creating `Box`.

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.

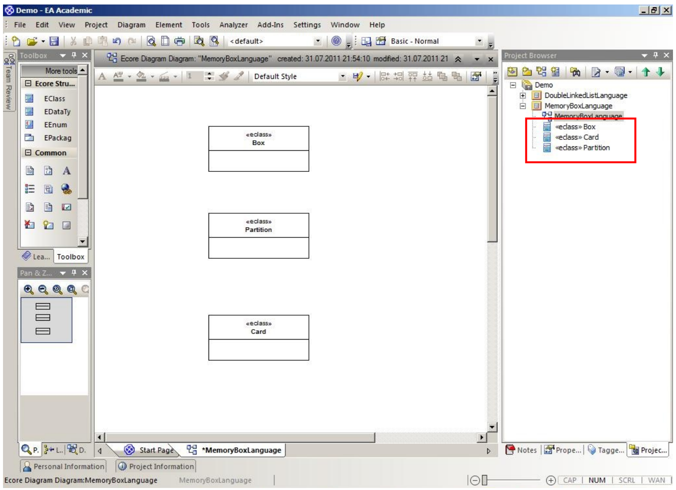



Figure 3.11: Main classes in our metamodel.

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

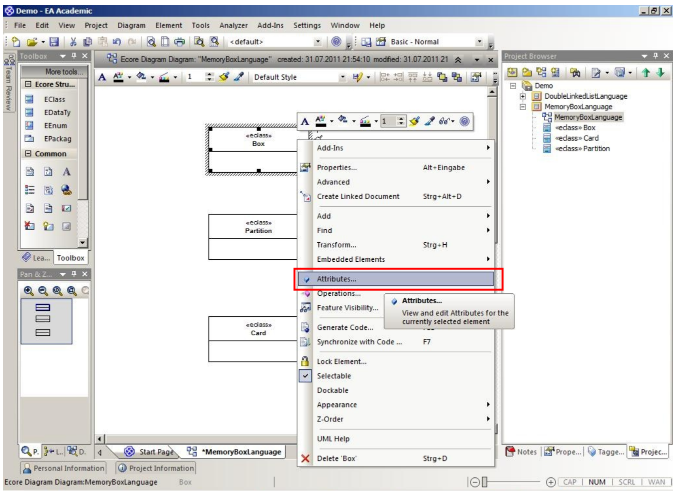



Figure 3.12: Context Menu for a class.

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`.

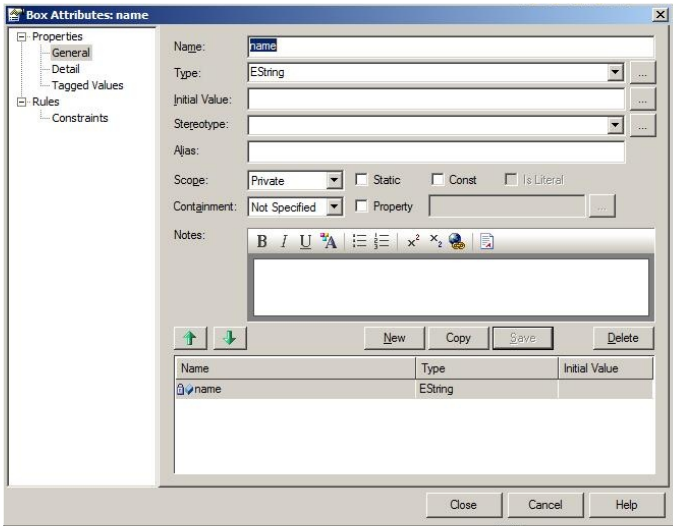



Figure 3.13: Adding attributes to a class.

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

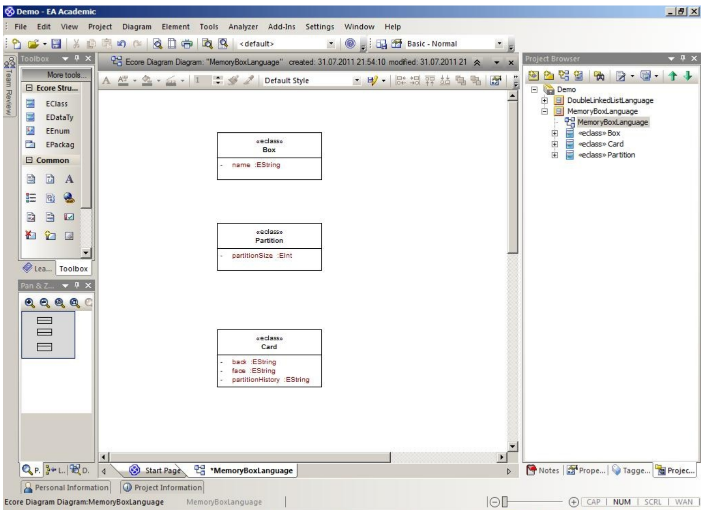


Figure 3.14: Main classes with attributes.

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).

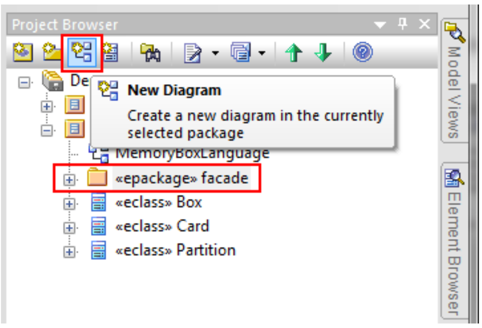



Figure 3.15: Add a class to the package.

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 +` ⇐ or ⇒ (Fig. 3.16).

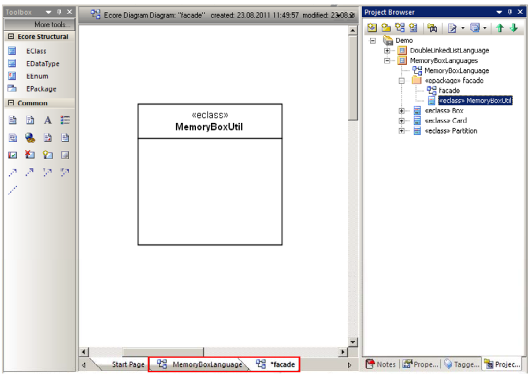



Figure 3.16: Main and subdiagrams in EA.

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.

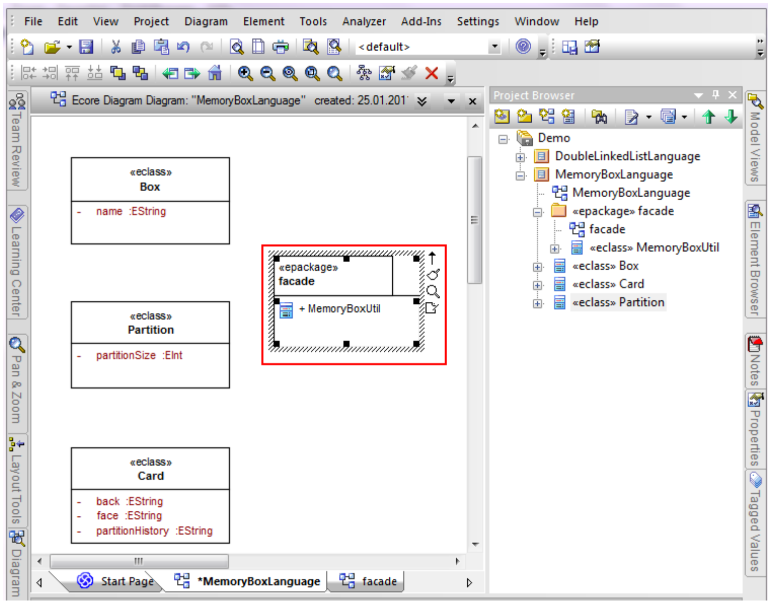



Figure 3.17: Workspace after adding package and util class.

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).

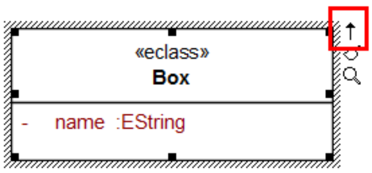



Figure 3.18: Quick Link is a central gesture in EA.

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

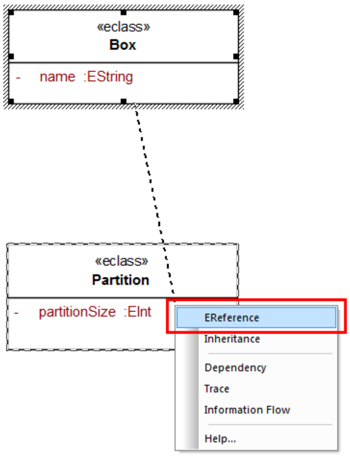



Figure 3.19: Create a reference via Quick Link.

Double click the reference to invoke a dialogue (Fig. 3.20), with which the direction of the reference can be set. The default is `Source` ⇒ `Target` and must be changed to `Bi-Directional` for our `Box`⇔`Partition` connection. A `Name` can also be entered, which is only used for documentation purposes and is not relevant for code generation.

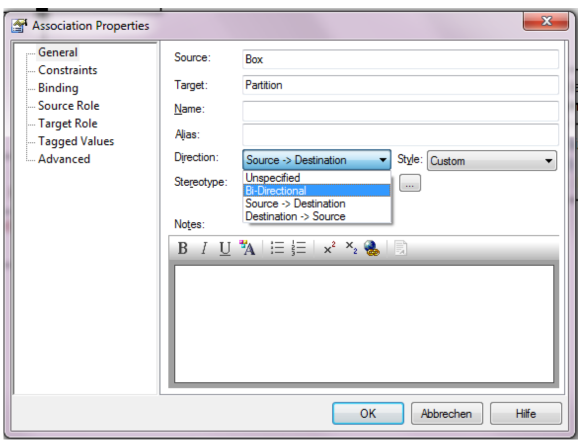
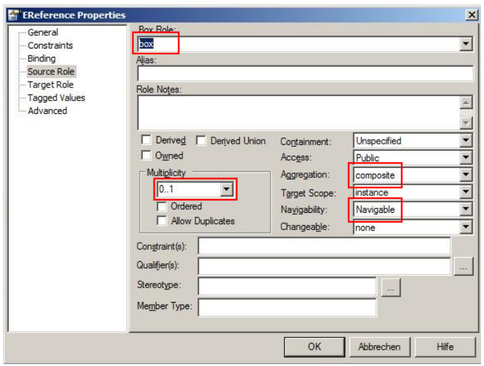



Figure 3.20: Enter properties of the reference.

In the same dialogue choose `Source Role` and enter the values in Fig. 3.21 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`.



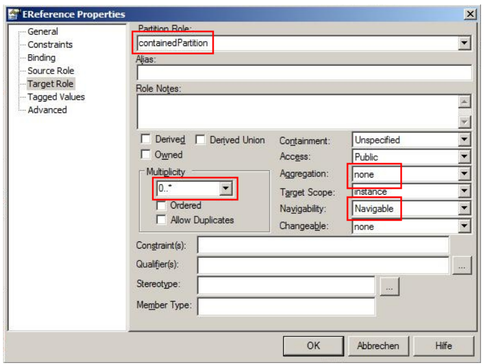




Figure 3.21: Enter properties for source and target of reference.

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`.

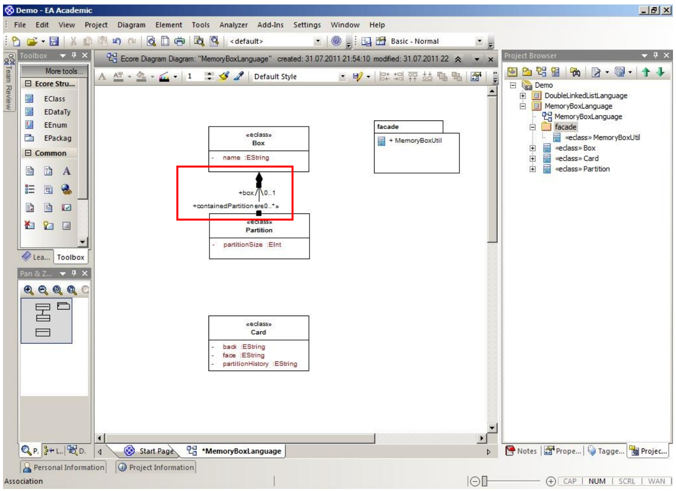


Figure 3.22: `Box` contains `Partitions`.

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

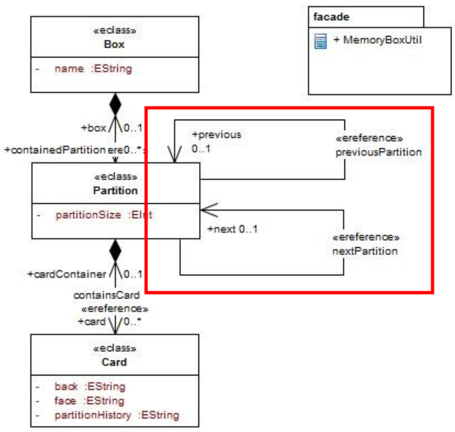



Figure 3.23: All relations in our metamodel.

[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...`.

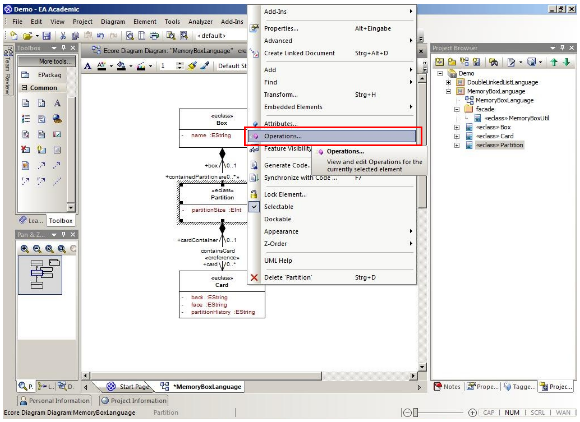


Figure 3.24: Add an operation.

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`.

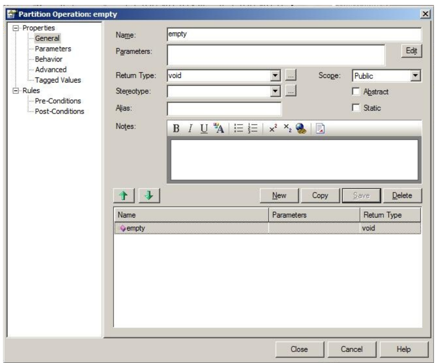


Figure 3.25: Properties for operation.

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.

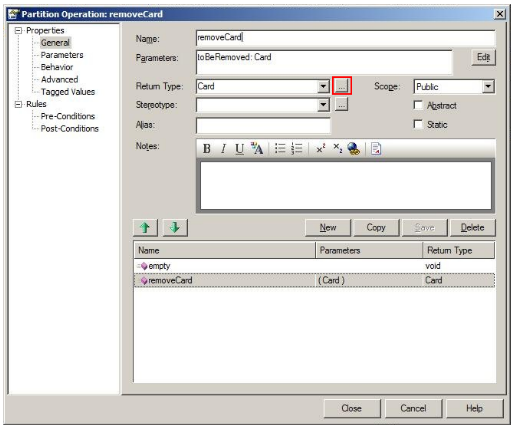



Figure 3.26: Parameters and Return Type.

Repeat the process for the values in Fig. 3.27. The `Return Type` can be chosen via the drop-down menu for primitives (e.g. `EBoolean`), or via the ... button (indicated in Fig. 3.26) for types in the metamodel (e.g. `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.

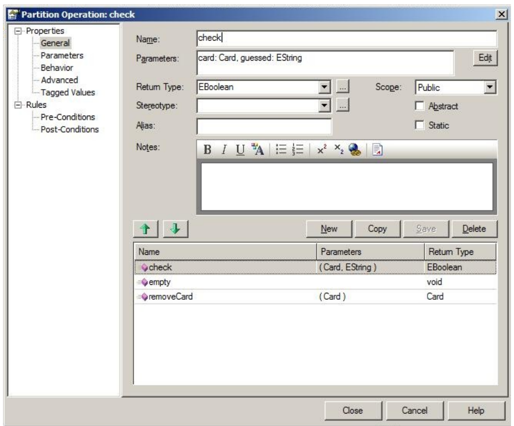



Figure 3.27: All operations in `Partition`.

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

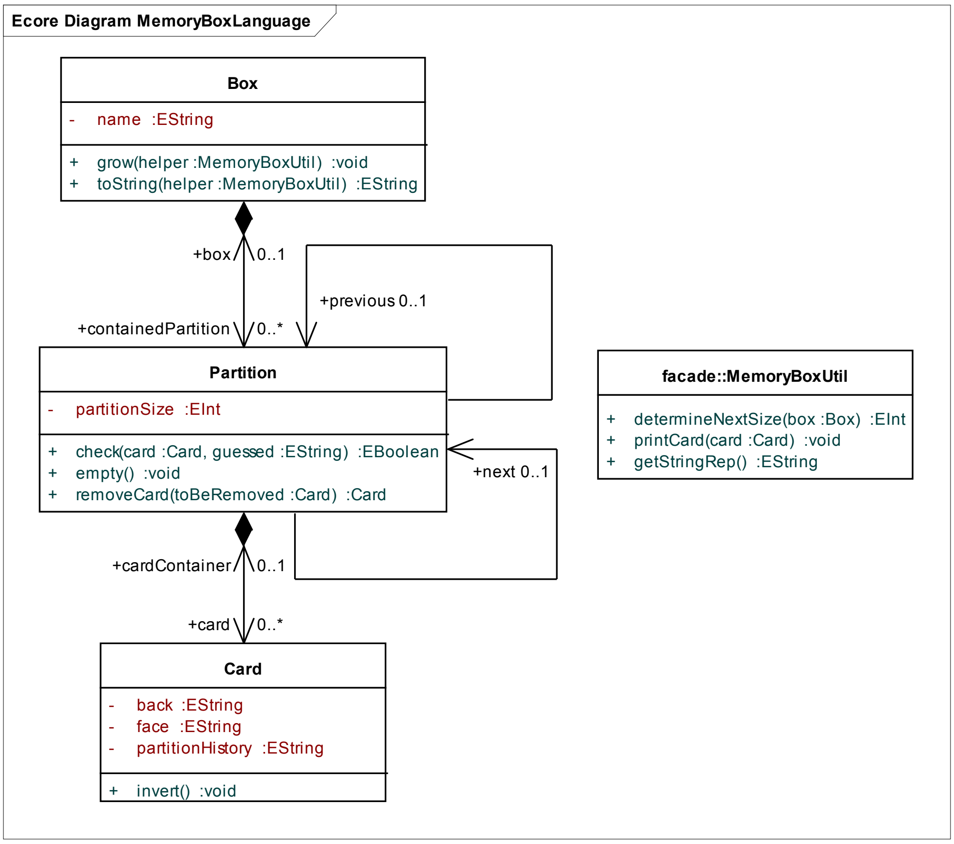



Figure 3.28: Complete metamodel for our memory box.

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

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 `Cards`, 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 partition-History 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 language 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 workingset.

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.

## 3.3 Creating an instance (model)

Before diving into modelling dynamic behaviour, let's have a brief look at how to create a concrete *instance model* of your metamodel in Eclipse. In the following, we use *metamodel* and *instance model* to differentiate between models that represent the abstract syntax and static semantics of a domain specific language (metamodel), and models that are expressed *in* such a language (instance models of the metamodel). To create an instance model, switch to your Eclipse workspace containing the generated working sets and projects from Sec. 3.2. EMF provides a generic model editor for free that allows us to create and edit an arbitrary instance of any metamodel specified with eMoflon.

Back in Eclipse, navigate to the model folder in your MemoryBoxLanguage project. Double-click the MemoryBoxLanguage.ecore model to invoke the *Ecore model editor*. Expand this tree to view the different classes and packages you modelled with EA in Sec. 3.2. To create a concrete instance of the metamodel, you must select a class which will become the root element of the new instance. For our example, right-click the class `Box` and choose `Create Dynamic Instance...` from the context-menu as depicted in Fig. 3.29.

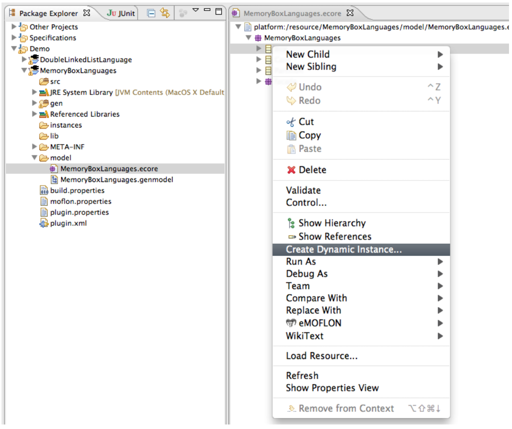



Figure 3.29: Context menu of Ecore model in Eclipse

A dialogue should pop up asking where instance model file should be persisted. We suggest saving all your instances in a folder named `instances` that is created in every new repository project. This is however just a convention, you are of course free to store your instances anywhere. Last but not least, enter a name for the instance model (Fig. 3.30).

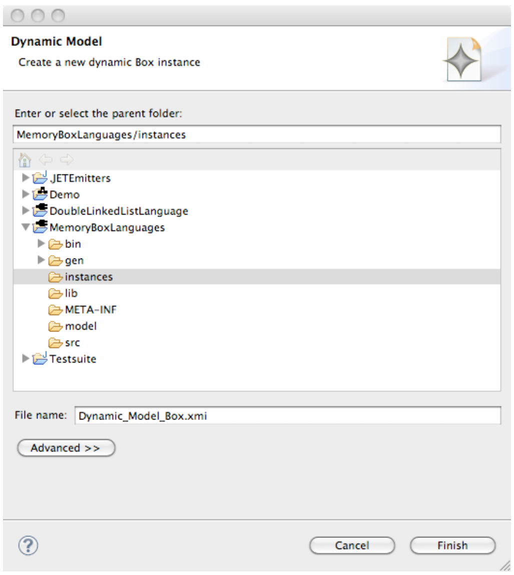



Figure 3.30: Dialogue for creating a dynamic model instance

Now click `Finish` and the *generic model editor* should be opened for your instance model. This editor works just like the Ecore model editor but is "generic" as it allows you to create and edit an instance of *any* metamodel not just of Ecore. You can populate your instance model by adding new children or siblings via a right-click on an element of the instance model to invoke the context-menu depicted in Fig. 3.31. Note that EMF supports you by respecting your metamodel and reducing the choice of creatable elements to valid types only, depending on the current context.

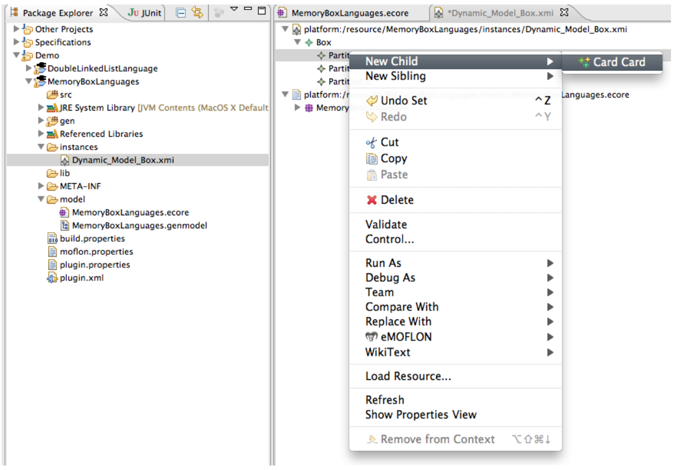



Figure 3.31: Context menu for creating model elements

You can save your model as an XMI file by pressing `Ctrl+S`. The model can be reloaded via a simple double-click to invoke the generic model editor.

*That's all for the "static part" of our metamodel. Let's move on and model the dynamic behaviour of our memory box!*

## 3.4 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 $\Delta : S \overset{\Delta}{\rightarrow} S^*$. 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. 3.32.

$$\begin{tikzcd}
L \arrow[d, "① m"'] & L \cap R \arrow[l] \arrow[r] \arrow[d] \arrow[dl, phantom, "PC"] \arrow[dr, phantom, "PO"] & R \arrow[d, "m^*"] \\
G & ② G \setminus \underbrace{(L \setminus R)}_{Destroy} \arrow[l] \arrow[r] & ③ H
\end{tikzcd}$$

Figure 3.32: 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 [3] 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. 3.33 depicts the rule *moveCard*.

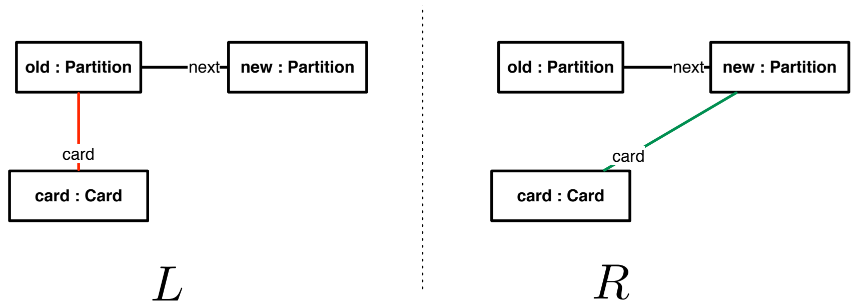



Figure 3.33: Rule *moveCard* as a graph transformation rule.

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. 3.34).

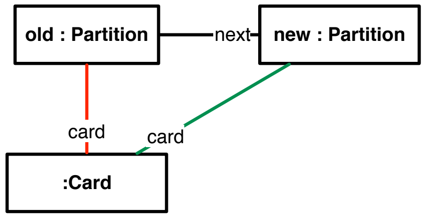



Figure 3.34: Compact representation of *moveCard* as a Story Pattern.

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. 3.35.

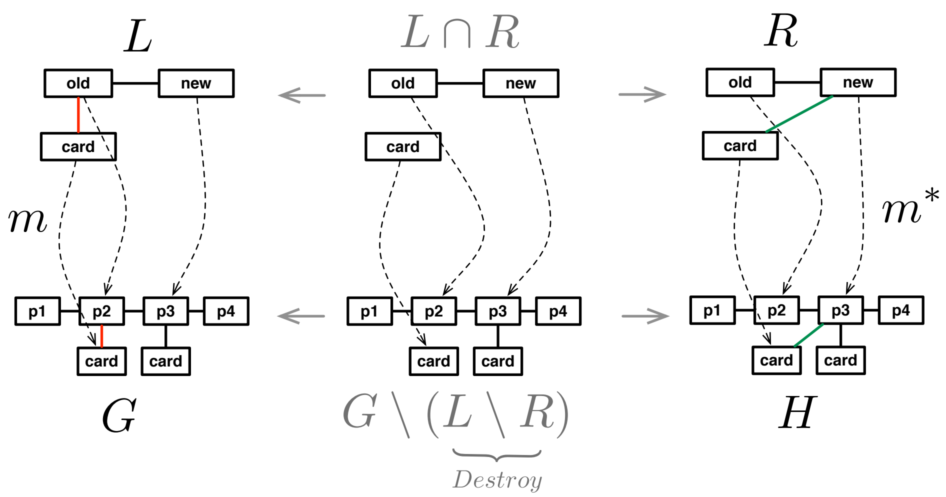



Figure 3.35: Applying *moveCard* to a memory box.

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...

### 3.4.1 Removing cards from a partition

Back in EA, open the main diagram (double-click in the project browser) and carefully do the following: (1) *Click once* on `Partition` to select it, then (2) *click once* on the method `removeCard` to choose it (Fig. 3.36). Now (3), *double-click* on the chosen method to indicate that you want to implement it.

*Create an SDM*

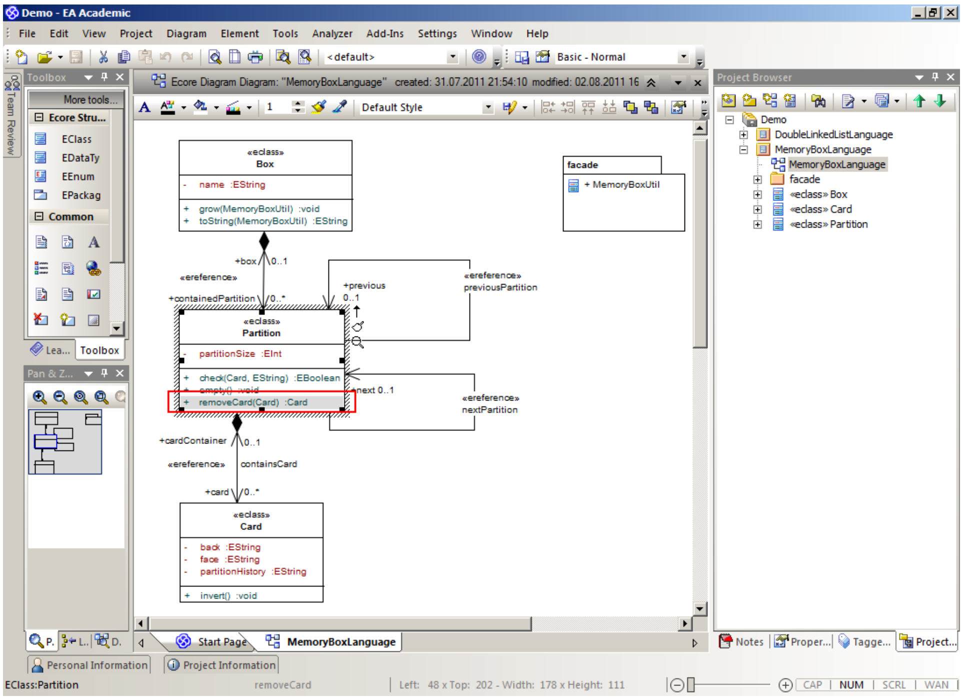



Figure 3.36: Double-click a method to implement it.

If you did everything right, a new *activity diagram* should be created with a cute little *start node* labelled with the signature of the method as depicted in Fig. 3.37. Inspect your project browser and note that an `SDM_Container` has been created for the method `removeCard` to contain the diagram. If you're at any time unhappy with an SDM[3], you can always delete the appropriate container in the project browser and start from scratch, following the steps described previously to create a skeleton for a new SDM. Also note the new toolbox `SDM` that has been automatically opened up for the diagram and placed to the left above the common toolbox.

*Activity Diagram*

*Start Node*

*SDM Toolbox*

[3] As you might have already noticed, we use "SDM" interchangeably to mean our graph transformation language or a concrete transformation (a story model) used to implement a method and consisting of an activity diagram and a pattern in each story node. This will all be explained in detail.

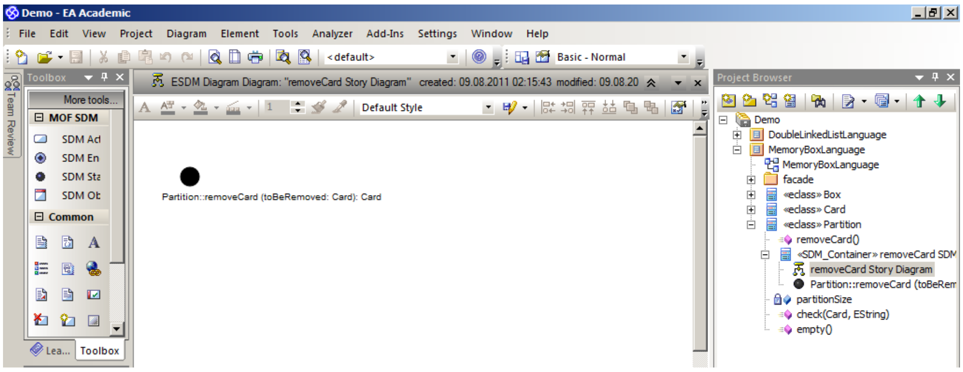



Figure 3.37: Generated SDM diagram and start node.

*Quick Create*

Now choose the start node, and note the small black arrow that appears (Fig. 3.38). Similar to quick linking which we learnt when creating our metamodel, a further fundamental gesture in EA is *Quick Create*. To quick create an element, pull the arrow and click on an empty spot in the diagram where the new element is to be created. This is basically quick linking to a non-existent element if you wish.

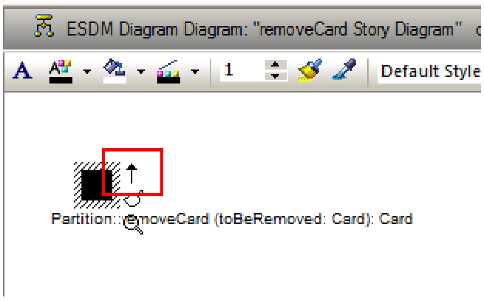



Figure 3.38: Quick link in SDM diagram to create new activity node.

EA notices that there is nothing to quick link to and pops up a small context-sensitive dialogue, not for creating a link as in the case of quick linking, but for creating an element that can be connected to the indicated source element.

*Activity*
*Activity Node*
*Stop Node*
*Activity Edge*

As indicated in Fig. 3.39 choose `Append Activity Node` to create an *activity node*. We shall refer to the whole activity diagram simply as an *activity* that always starts with a start node, terminates with a *stop node* and consists of activity nodes connected via *activity edges*. If you quick created correctly, you should now have a start node, an activity node called `ActivityNode 1` and an edge connecting the start node and the activity node.

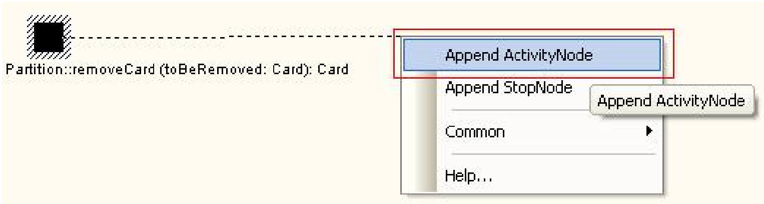



Figure 3.39: Create new activity node.

Complete the activity by quick creating a stop node as depicted in Fig. 3.40.

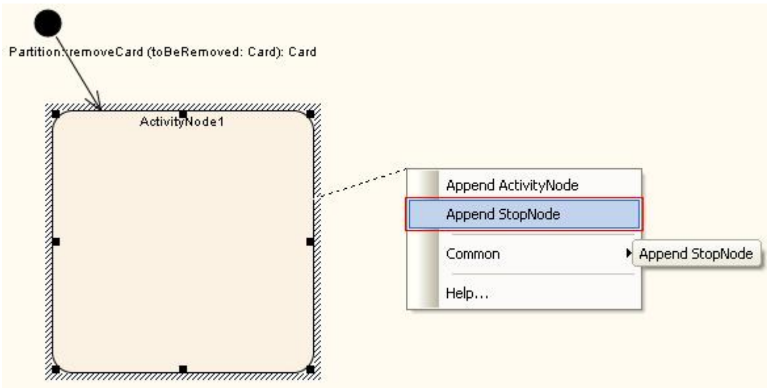



Figure 3.40: Complete activity with a stop node.

If you did that right as well you should now have a complete activity that models the procedural *control flow* of our method. The semantics of our activities is pretty straightforward – the control flow starts in the start node and flows along edges and connected activity nodes till it terminates in a stop node. The complete activity is depicted in Fig. 3.41 now with the activity node connected via an activity edge to the newly created stop node.

Integrated as an atomic step in this overall control flow, a single graph transformation step can be embedded in some activity nodes as a *story pattern*. These story patterns are declarative transformation rules as introduced in Sec. 3.4. As not all activity nodes can contain story patterns (e.g. start and stop nodes), those that can are called *story nodes*.

*Story Pattern*

*Story Node*

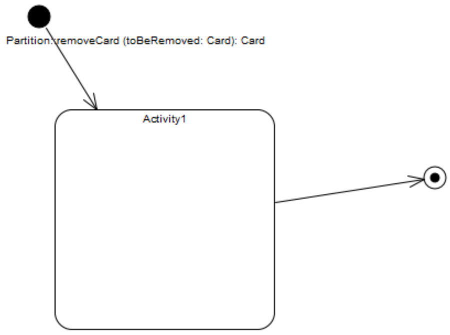



Figure 3.41: Control flow modelled as a simple activity diagram.

To create a story pattern, double click the story node `ActivityNode 1` in Fig. 3.41 to prompt the dialogue depicted in Fig. 3.42. Enter `removeCard-FromPartition` as the name of the story node, check `Create this Object` and click `OK`.

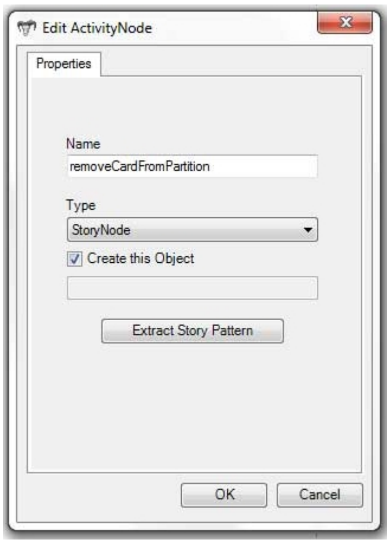



Figure 3.42: Start modelling story pattern in activity node.

*Object Variable*

*Pattern Matching*

The activity node should now have a single *object variable* `this` (Fig. 3.43). Object variables are, as the word "variable" indicates, place holders for actual objects in a model. During *pattern matching*, actual objects in the current model are assigned to the object variables in the pattern according to the indicated type of the object variable and other conditions[4]. In our

[4]We shall learn what conditions can be specified in a few pages.

case, the current story pattern consists of only one object variable, which is assigned (per convention) to `this` in Java (the object whose method is invoked).

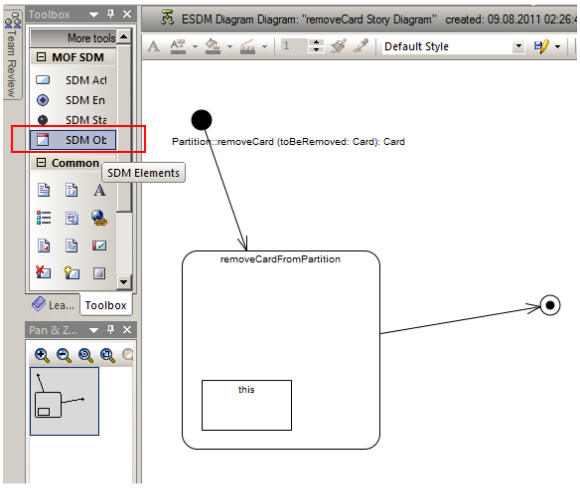



Figure 3.43: Add a new object variable from the tool-box.

To create an object variable that can be assigned to other objects, choose `SDM ObjectVariable` from the toolbox as indicated in Fig. 3.43 and click *in the* activity node `removeCardFromPartition` (Fig. 3.44).

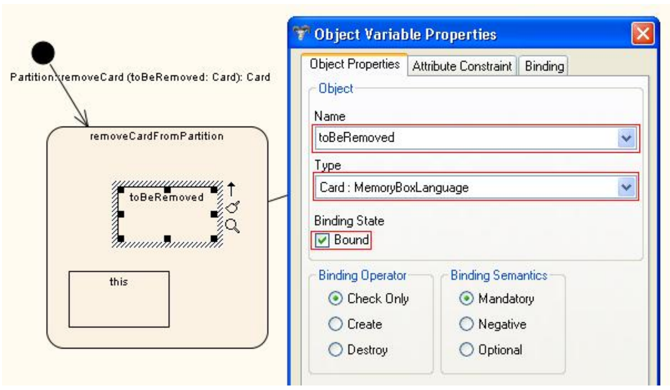



Figure 3.44: Specify properties of the added object variable.

In the dialogue that pops up, choose `toBeRemoved` as the name of the object variable and `Card` as its type using the corresponding drop-down menus.

Because `toBeRemoved` is a parameter of the method, it is offered as a possible name in the drop-down menu and can be directly chosen to prevent annoying mistakes due to typing the name of the parameter wrongly.

*Binding State*

In the dialogue, note the option `Bound` that must be set. For the pattern matcher, bound object variables do not need to be assigned as they already have a fixed value from the context of the method. We have already seen two cases for bound object variables: the assignment to `this` (the current partition who owns the method), and assignments to parameters of the method that are specified when invoking the method. Please note that the assignment or *binding* is in both cases implicit and via the *name* of the bound object variable.

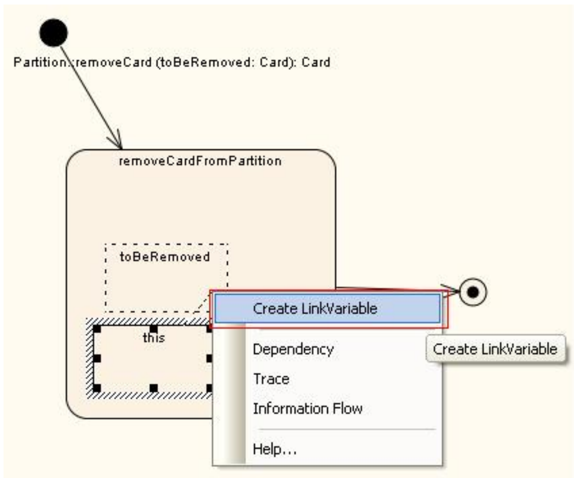



Figure 3.45: Create a link variable.

*Link Variable*

Models consist not only of objects but also of *links*. To match links one can thus create *link variables* in story patterns that act as place holders for links in a model. To create a link variable between the current partition, whose `removeCard` method is invoked, and the card to be removed, which is passed in as a parameter of the method, choose the object variable `this` and quick link it to the object variable `toBeRemoved`. In the quick link dialogue choose `Create LinkVariable` (Fig. 3.45).

*Binding Operator*

In the property dialogue that pops up, choose the offered link type (according to the metamodel, there is only one possible link type between a partition and a card), and set the *Binding Operator* to `Destroy` (Fig. 3.46). Every object or link variable's binding operator can be set to one of `Check Only, Create, Destroy`.

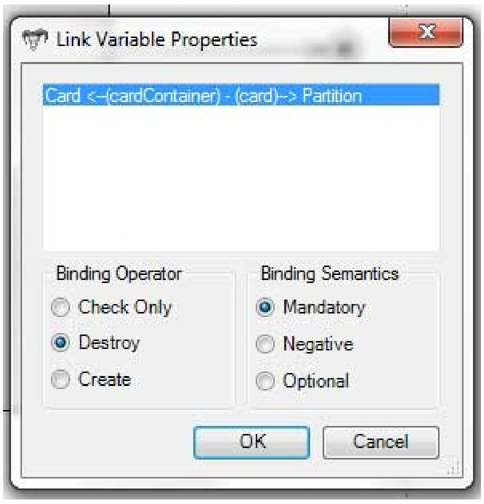



Figure 3.46: Specify properties for created link variable.

For a rule $r : (L, R)$, as discussed in Sec. 3.4, this marks the variable as belonging to the set of elements to be retained $(L \cap R)$, the set of elements to be newly created $(R \setminus L)$, or the set of elements to be deleted $(L \setminus R)$.

According to the signature of the method `removeCard`, we should return the card that has been deleted. Although this might strike you as slightly odd, considering that we already passed in this exact card as an argument, it still makes sense as it allows for chaining method calls:

```
aPartition.removeCard(aCard).invert()
```

*Return Values*
*Expressions*

In any case, a return value for an SDM can be specified in the stop node. As depicted in Fig. 3.47, double-click the stop node to prompt the `Edit StopNode` dialogue. In the `Expression` field, choose `ParameterExpression`, and `toBeRemoved` as the parameter. In many different dialogues, we employ a simple context-sensitive expression language for specifying required values. We have intentionally avoided creating a full-blown sub-language and limit expressions to a few simple types[5]. The philosophy here is to keep things simple and concentrate on what SDMs are good for – expressing structural change. Our approach is to provide a clear and type-safe interface to a general purpose language (in our case Java) and support a simple *fallback* as soon as things get low-level and difficult to express as a pattern.

The alternative approach would be to support arbitrary expressions, for

[5]We also do not support nesting expressions

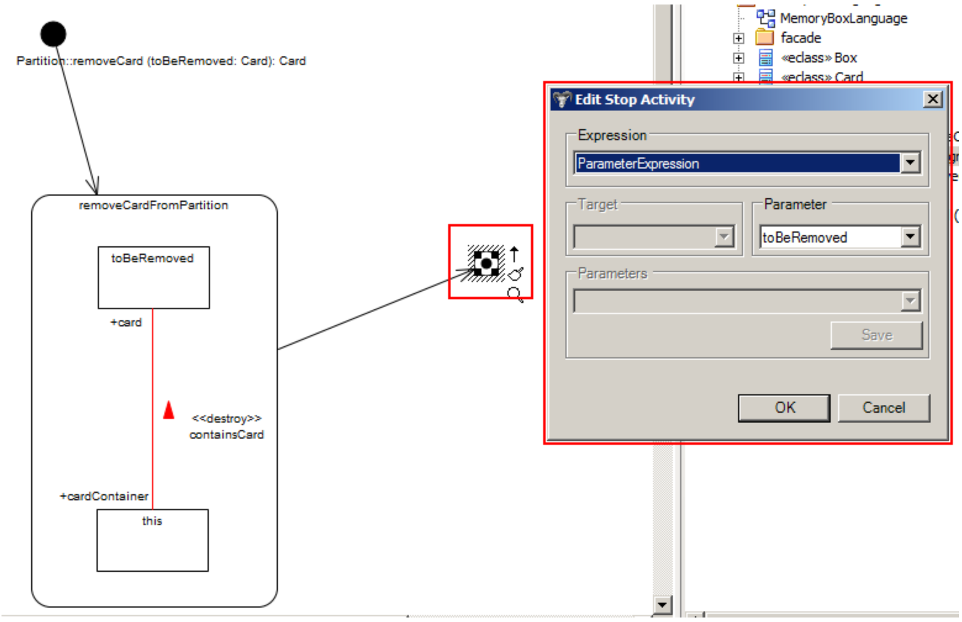



Figure 3.47: Adding a return value to the stop node.

example, in a script language like JavaScript or in an appropriate DSL[6] designed for this purpose. In a few pages we'll learn the other expression types we support and how to use them, for the moment, a *Parameter Expression* is used to refer to one of the parameters of the current method, which is exactly what we needed and have used for our `removeCard` SDM. If you've done everything right, your complete SDM should now look like Fig. 3.48 with the return value indicated below the stop node.

*Parameter Expression*

Let's take a step back and review briefly what we have specified: if `p.removeCard(c)` is invoked for a partition `p` with a card `c` as argument, the specified pattern will *match* if the card is contained in the partition. After determining a match for all variables, the link between the partition and the card is deleted, effectively "removing" the card from the partition. If the card is *not* contained in the partition, the pattern won't match and nothing happens. In both cases the card that was passed in is simply returned.

Congratulations! You have specified your very first SDM. Don't forget to export and generate code in your Eclipse workspace. Inspect the generated implementation for the method and see if you can get a feel for what

[6]A DSL is a Domain Specific Language: a language designed for a specific task which is usually simpler than a general purpose language like Java and more suitable for the exact task.

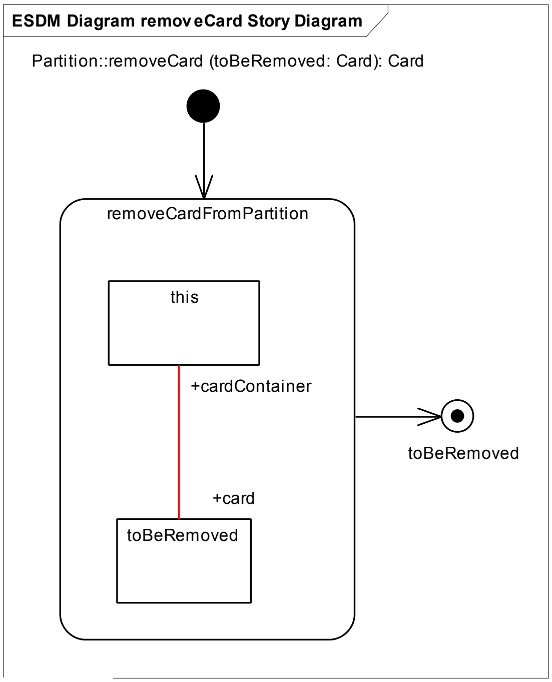



Figure 3.48: Complete SDM for `Partition::removeCard`.

the generated code does. Notice all the null checks that are generated automatically – only a very conscientious (and probably slightly paranoid) programmer would program so defensively!

If you're unable to export or generate code successfully, compare your SDM carefully with Fig. 3.48 and make sure you haven't forgotten anything.

In the following sections, we shall explore further features of SDM that allow for really expressive and powerful patterns.

### 3.4.2 Checking a card

The next method we shall model with SDMs is probably the most important: a user decides to try a card in the memory box and looks at the question on the card (`Card.face`), makes a guess and *checks* to see if the guess was correct by comparing with the answer on the back of the card (`Card.back`). If the guess was correct the card can be *promoted* by moving it to the *next* partition, if it was wrong the card is *penalised* by moving it to the *previous* partition.

As you're almost an SDM wizard already, try, using concepts we have already learnt, to create the control flow for `Partition::check` as depicted in Fig. 3.49.

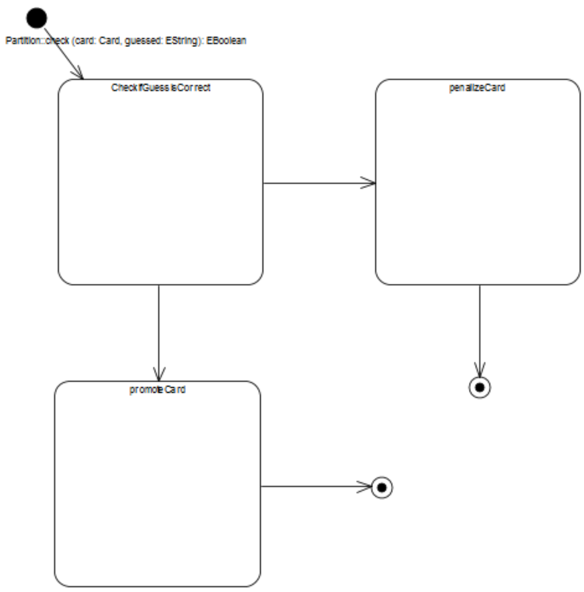



Figure 3.49: Activity diagram for `Partition::check`.

To check if the guess was correct, create an object variable that is bound to the argument `card`, representing the card the user has picked from the memory box. Remember that this binding is implicitly specified by choosing the name of the argument as the name of the object variable (Fig. 3.50).

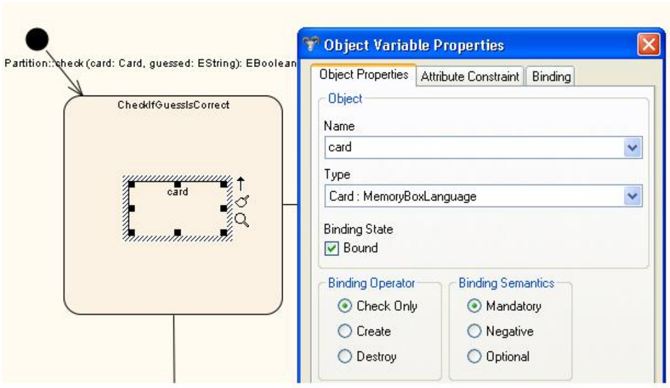



Figure 3.50: Add the card to be checked.

*Attribute Constraint*

Now that we have the card to be checked, we need to compare the user's guess to the actual answer on the back of the card. To do this we need to specify an *Attribute Constraint*. An attribute constraint is a non-structural condition that must be satisfied for a story pattern to match, and can be specified by choosing the `Attribute Constraint` tab as depicted in Fig. 3.51. In this dialogue, choose the attribute to be used in formulating the constraint (`back`) and the type of `Expression` used to express the constraint. As we shall compare the back of the card with the user's guess, passed in as a parameter, we need a `ParameterExpression` to refer to this value. In the previous section, we already used parameter expressions to specify the return value in a stop node. Now choose the parameter (`guessed`) and the type of constraint or *operation* to be executed – in this case an equality check (`==`). Press the button labelled `Add` and admire your first attribute constraint (Fig. 3.51)!

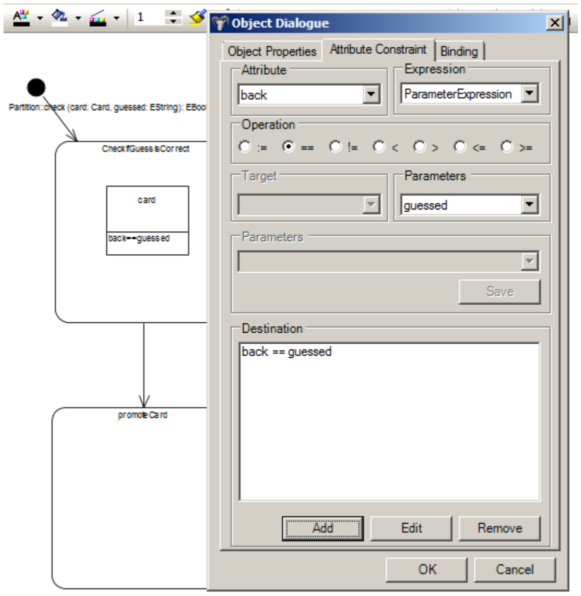



Figure 3.51: Add an attribute constraint with a parameter expression.

*Edge Guards*

*Guard Type*

Let's get back to the control flow for a bit. We need to specify that the card is to be penalised if the guess was wrong (the story node `CheckIfGuessIsCorrect` did not match) and to be promoted if it was correct (a match could be found, i.e., all constraints/conditions both structural and non-structural could be fulfilled). Such an if/else construct is specified in SDMs via *Edge Guards*. To add a guard to the edge leading from `CheckIfGuessIsCorrect` to `penalizeCard`, double click the edge and choose the *Guard Type* in the dialogue (Fig. 3.52). Choose `Failure`, repeat the process for the edge leading to `promoteCard` and choose `Success`.

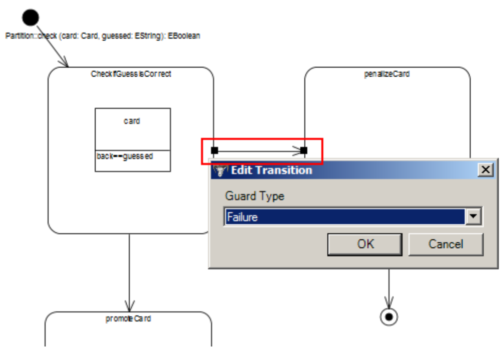



Figure 3.52: Add a transition with a guard.

The next feature of our tool we shall learn is a means of coping with large patterns. It might be nice to visualise *small* story patterns directly in their story nodes, but for large patterns or complex surrounding control flow, such diagrams would get very cumbersome and unwieldy pretty quickly. This is indeed a popular argument against visual languages and it might already have crossed your mind ("this is cute, but it'll *never* scale!"). With the right tools and concepts however, even huge diagrams can be mastered. We support *extracting* story patterns to their own extra diagrams and recommend this for most cases (unless the pattern is really concise and only contains about 2-3 object variables). *Extracting Patterns*

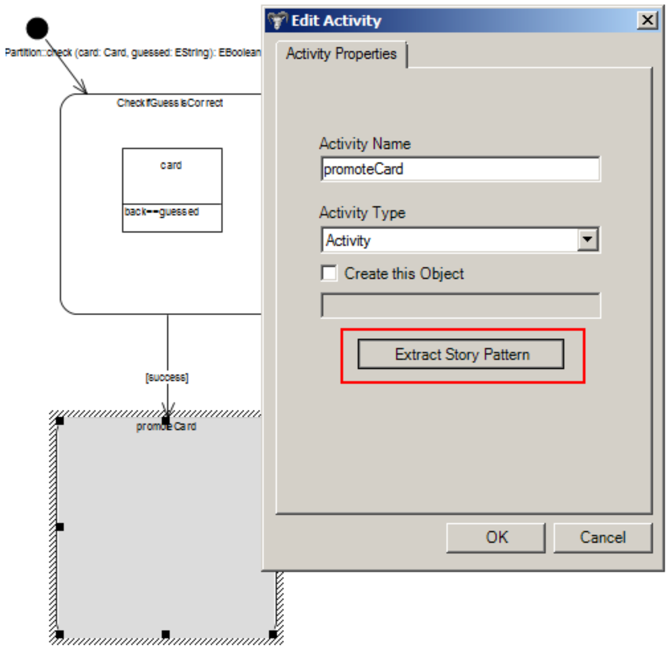



Figure 3.53: Extract a story pattern for more space and a better overview.

To extract an empty or already partially modelled story pattern, just double-click the corresponding story node (`promoteCard`) and choose `Extract Story Pattern` (Fig. 3.53). Note the new diagram that is immediately opened and created in the project browser (Fig. 3.54).

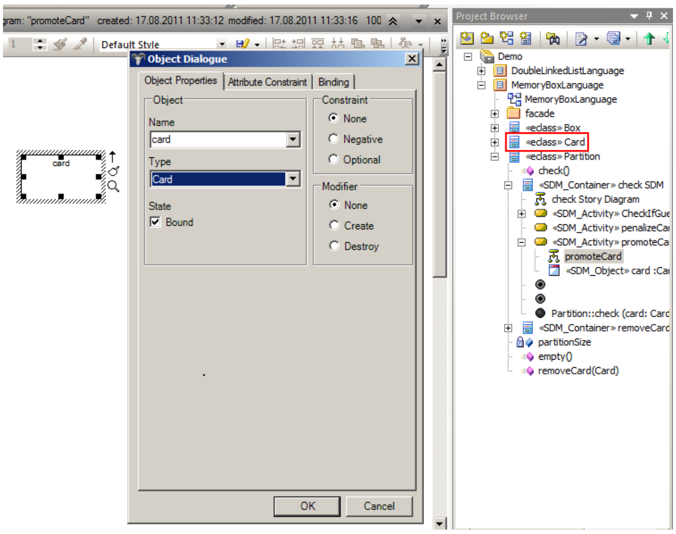


Figure 3.54: Add an object variable per drag and drop.

*Drag & Drop*

Yet another EA gesture is good old *Drag and Drop* from the project browser[7], which we use as an alternative to the SDM toolbox. To create an object variable, simply drag and drop the class `Card` from the project browser and into the extracted story pattern diagram (Fig. 3.54). A dialogue should pop-up asking if you want to (1) create a simple link (referring to `Card` as a class) or (2) create an Object (as an instance of `Card`), or (3), if you want to create a subclass. In our case we want to create an object(variable) and so (2) is nearest in meaning. As this `Paste Element` dialogue is a bit annoying, EA allows you to choose a default for *all* drag and drop gestures. Go ahead and check `All Drag and Drop` so that option (2) is used next time as the default. Furthermore, you should also check `Only show this dialog when Ctrl+Mouse drag is used`, so that the default is used *without* popping up this dialogue for confirmation. Don't worry, if you ever need options (1) and (3), for example when metamodelling, you just need to hold `Ctrl` when dragging to invoke the dialogue and change the settings to suit the current modelling activity.

The main advantage of drag and drop is that the `Object Variable Properties` dialogue that now pops-up should have the type of the object variable pre-configured. Choosing the type in the project browser and dragging it in is for most people a more natural gesture than choosing the type from a long drop-down menu and can really be a great time saver for large metamodels[8].

[7] Remember the other two gestures we have learnt: Quick Link and Quick Create.

[8] Drag and drop is also possible in embedded story patterns (still visualised in their story nodes). You must ensure however, that the object variable is *completely* contained inside the story node and does not stick out over any edge

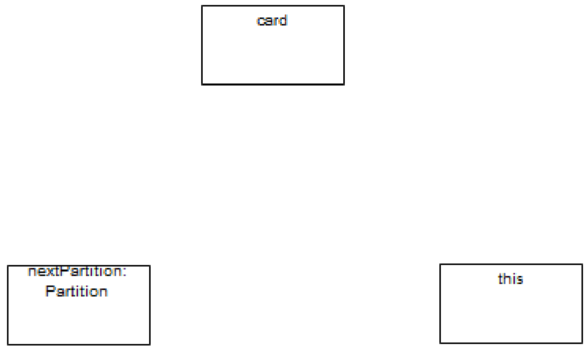



Figure 3.55: All object variables for story pattern `promoteCard`.

Let's move on with the current pattern. Remember that we want to promote the card. As a first step drag and drop two further object variables for `this` (the current partition) and the next partition according to (Fig. 3.55). An important point to note here is that `nextPartition` is visually differentiated from `this` and `card` by indicating its type, i.e. (`nextPartition:Partition`). This is how we differentiate *bound* variables (`this, card`) from *unbound* or *free* variables like `nextPartition`. We already know that matches for bound variables are completely determined by the current context (argument of the method, current "this" object). Matches for unbound variables on the other hand, have to be determined by the pattern matcher. Such matches are "found" by navigating and searching in the current model for possible matches that satisfy all specified constraints (e.g. type of the variable, links connecting it to other variables and attribute constraints).

*Bound vs. Unbound*

In our case, the next partition has to be determined, by navigating from `this` via the `next` link in the metamodel. Make sure the bound checkbox for `nextPartition` is left empty and quick link from `this` to `nextPartition`, or vice-versa, to create a `next` link variable as indicated in Fig. 3.56.

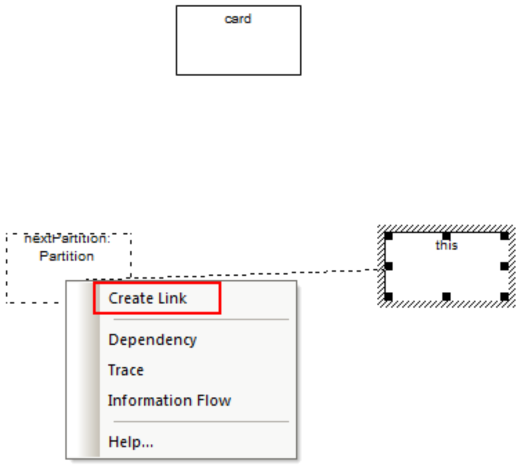



Figure 3.56: Create a link variable.

If you've done everything right, your story pattern should now closely resemble Fig. 3.57. Take a step back and reflect on what the pattern expresses.

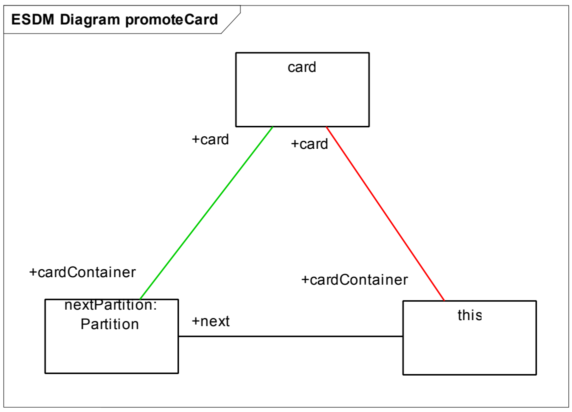



Figure 3.57: Complete story pattern for activity node `promoteCard`.

Now repeat the process for the story node `penalizeCard`: extract the story pattern, and create all variables as depicted in Fig. 3.58. This pattern is quite similar to `promoteCard` but moves the card from `this` to `previous` instead of `next`. Just like before, `previousPartition` is unbound and must be determined by navigating from `this` along the link `previous`.

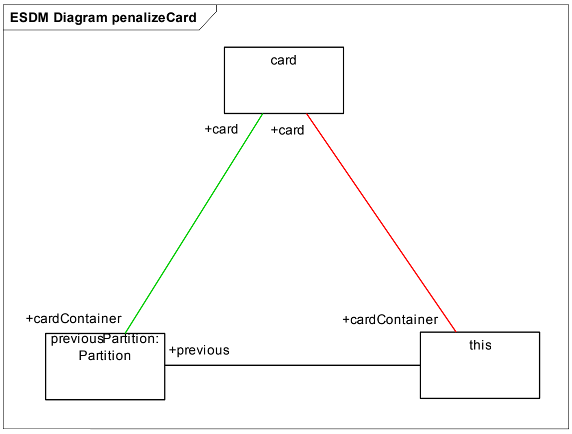



Figure 3.58: Story pattern for activity node `penalizeCard`.

To complete our SDM, we need to signalise, as a return value, if the guess was correct or not (and consequently if the card was promoted or penalised). To do this, double-click the stop node after `promoteCard` and choose `LiteralExpression` as the type of expression (Fig. 3.59).

*Literal Expression*

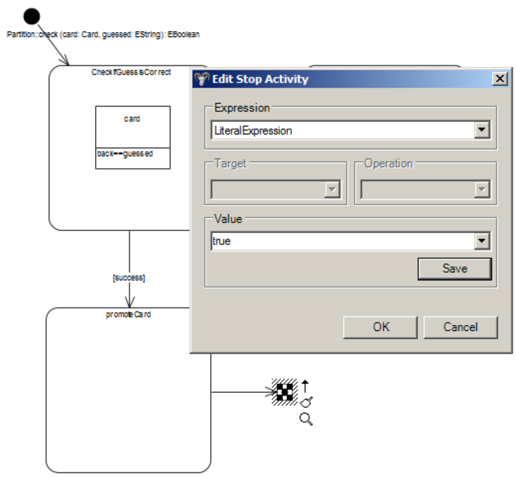



Figure 3.59: Add a return value with a literal expression.

*Literal expressions* can be used to specify arbitrary text. This should actually be used only for *literals* like 42, “foo” or `true` but can of course be (mis)used for formulating any (Java) expression that will simply be transferred “literally” into the generated code. This is obviously sort of dirty[9] and should be avoided if possible.

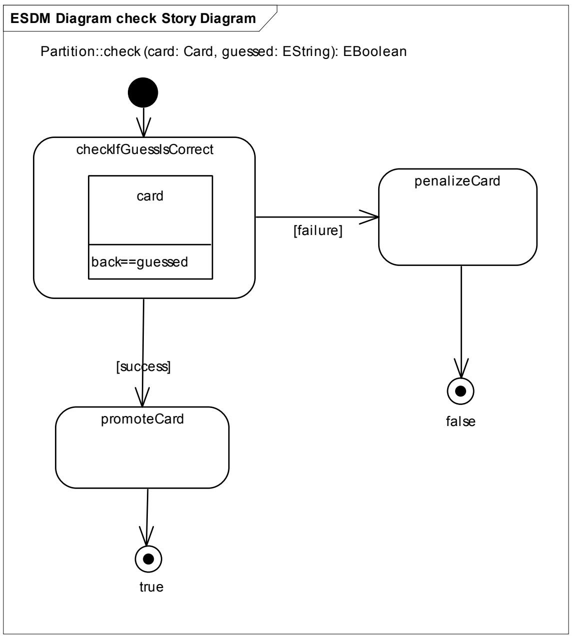



Figure 3.60: Complete SDM for `Partition::check`.

Type in `true` as the value of the expression (Fig. 3.59) and complete the SDM by returning `false` after penalising a card. Please ensure that your SDM (the control flow) closely resembles Fig. 3.60. As always, export the project, generate code and inspect the implementation for `check`. We strongly recommend that you even write a simple JUnit test (take a look at our simple test case in Sec. 2.4 for inspiration) to take your brand new SDM for a test-spin.

[9]It defeats, for example, any attempt to guarantee type safety.

### 3.4.3 Emptying a partition of all its cards

The next SDM we shall specify should *empty* a partition of all its cards, deleting the cards in the process. To do this we obviously need a construct for repeating the action for all cards in the partition. In SDM, this is accomplished via a *For Each* story node. A for each story node performs the specified actions for *every* match of its story pattern. To create a for each story node, create the initial diagram and start node for the method `Partition::empty` and quick create an activity node choosing `ForEach` as its type (Fig. 3.61).

*For Each*

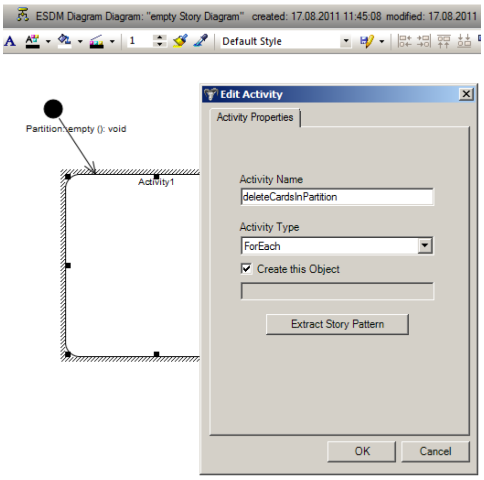



Figure 3.61: A for each loop in SDM.

A for each story node is visualised as a double node to indicate the potential repetition (Fig. 3.62). Complete the story pattern as indicated in Fig. 3.62. Please note that the `card` that is deleted in each match is unbound and both the `card` and link to `this` are set to `destroy`. Even more important, note that the guard that terminates the for each story node has an `[end]` guard. Indeed, a for each story node *must* have an end activity edge which is taken when all matches for the story pattern have been handled.

There are two interesting points to note: First of all, how would the pattern be interpreted if the story node where a normal story node and not a for each? Well, the pattern would specify that *a* card should be matched and deleted from the current partition. Note that the *exact* card is not specified and indeed the actual choice of the card is *non-deterministic* or random. This is a common property of graph transformations and pattern matching and is something that takes some getting used to. In general, there are no guarantees concerning the choice and order of valid matches.

*Non-Determinism*

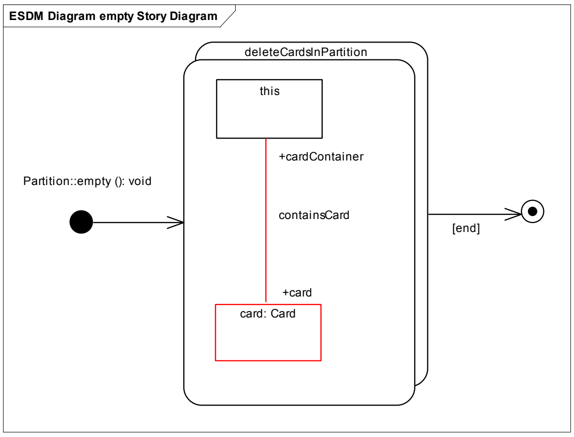



Figure 3.62: Complete story pattern with `[end]` guard.

*Dangling Edges*

The second point is if we need to destroy the link between `this` and `card`. Would the pattern be interpreted differently if we just destroyed `card` and left the link? The answer is no, the pattern would yield the same result, regardless of if the link is explicitly destroyed or not. This is because the transformation engine we use[10] ensures that there are never any *dangling edges* in a model. As deleting only `card` would result in a "dangling edge" attached to `this`, the link is deleted as well. Explicitly destroying the link or not is therefore a matter of taste, but ... why not be as explicit as possible?

[10] CodeGen2 which is part of Fujaba `http://www.fujaba.de/`

### 3.4.4 Turning a card around

The next SDM *inverts* a card by swapping its back and face values. This therefore "turns the card around" in the memory box, which makes sense when learning, for example, a new language. You're no longer an SDM beginner so try to model the SDM depicted in Fig. 3.63.

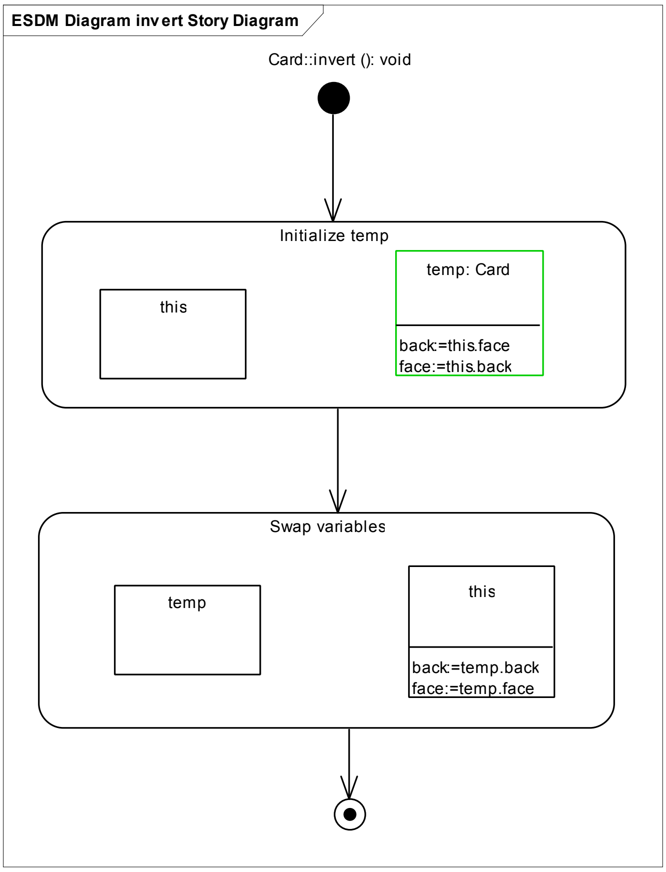



Figure 3.63: Swap back and face of the card.

Something new here is that we use *assignments* to set the attributes of `temp` in `Initialize temp` and to swap the attributes of `this` in `Swap variables`. An assignment is an attribute constraint with `:=` as operation. Although it might be slightly confusing to refer to an assignment as a constraint, if you think a while about it *everything* can be viewed as a constraint that can be fulfilled via different strategies. In this case, an assignment is fulfilled not by searching for a match, as in the case of an assertion (`==`, `>`, `<`, ...), but by *performing* the assignment. Similarly, non-context elements (set to create or destroy) can be viewed as structural constraints that are

*Assignments*

*Assertion*

fulfilled by creating or destroying the corresponding element. A constraint is therefore a unifying concept similar to "everything is an object" from OO and "everything is a model" from metamodelling and has the usual advantages. If you're interested in why unification is considered cool check out [1].

*Unification*

A last point before we move on to the next SDM. Did you notice that `temp` is bound in the story pattern of `Swap variables`? This is a new case for bound variables that we haven't treated yet. Till now we have seen object variables that can be (1) bound to an argument of the method that is set when the method is invoked, or (2) bound to the current object `this` whose method is invoked. In both cases, the object to be matched is completely determined by the context of the method and does not need to be determined by the pattern matcher.

Setting `temp` as bound in `Swap variables` is a third case in which an object variable is bound to the value already determined in a *previous activity node*. This means that in our case, the object variable `temp` in `Swap variables` is to be bound to the value determined for the unbound object variable `temp` in `Initialize temp`. This way, you can always refer to previous matches for object variables in the preceding control flow. Please note that the reference or mapping is again implicit via the same *name* of the object variable. As in the case of arguments of the method, the editor provides rudimentary support via a drop-down menu which can be used to choose the name of an object variable and avoid possible mistakes when typing by hand.

### 3.4.5 Growing the box by adding a new partition

In this SDM, we shall specify how our memory box is built up and how the contained partitions are connected. This controls how cards move back and forward in the box. Although very different behaviour can be implemented, we shall implement the classical rules as depicted in Fig. 3.1.

Start off by creating the simple control flow and story pattern depicted in Fig. 3.64. This matches the box itself (`this`), and *any* two partitions in the box.

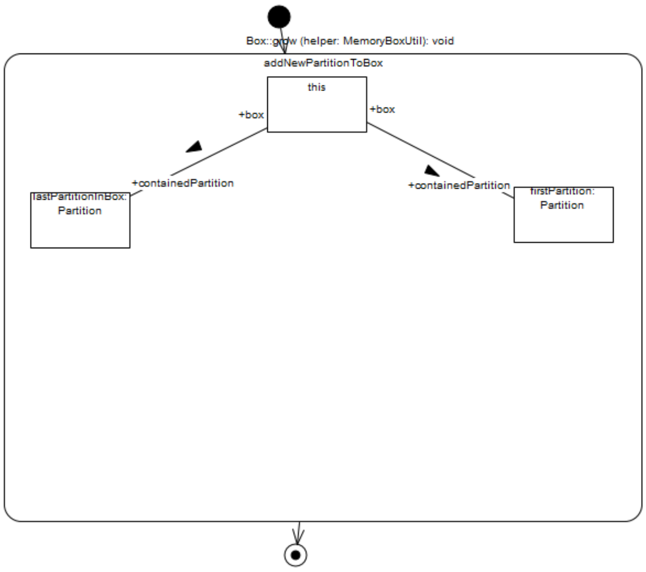



Figure 3.64: Context elements for SDM.

As already indicated by the chosen names of the object variables `firstPartition` and `lastPartitionInBox`, we actually want the pattern matcher to determine the first and the last partition in the box. But how do we specify this? As explained in section 3.4.3, the current story pattern will simply determine two partitions non-deterministically.

SDMs provide a declarative means of identifying the first and last partition via *Negative Application Conditions*, also simply referred to as NACs[11]. A NAC is a negative element that should *not* be present in a valid match. *NACs*
In the theory of algebraic graph transformations [3], NACs can be complex graphs that are much more general and powerful. In our implementation[12], however, we only support single negative elements (object or link variables).

[11] Pronounced \'*nak*\
[12] To be more precise CodeGen2 from Fujaba.

To create an appropriate NAC that constrains the possible matches for `lastPartitionInBox` to exactly the last partition in the box, create a new object variable `nextPartition` of type `Partition` and set its *Binding Semantics* to `negative` (Fig. 3.65). The object variable should be visualised as being cancelled or struck out.

*Binding Semantics*

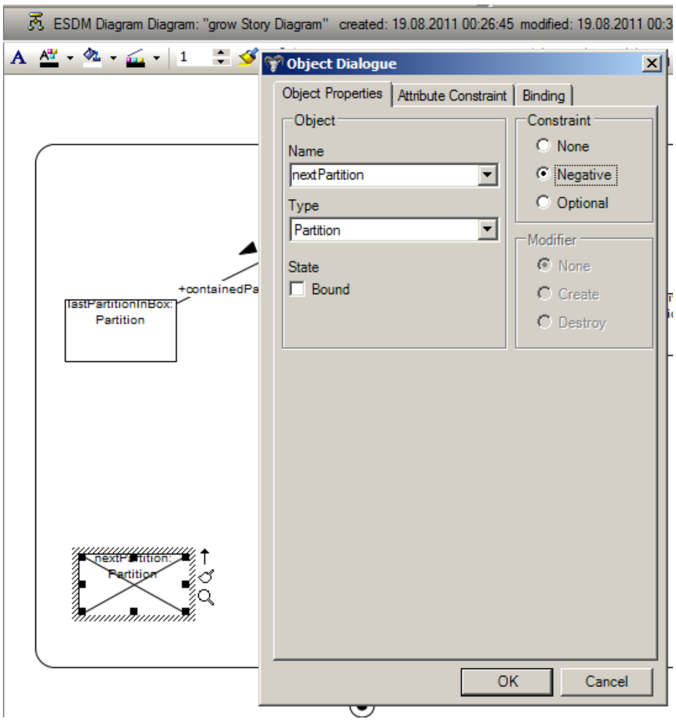



Figure 3.65: Adding a negative element.

Now quick link `nextPartition` to `lastPartitionInBox` and choose the link type carefully, so that `nextPartition` plays the role of `next` with respect to `lastPartitionInBox`. Now complete the story pattern so that it closely resembles Fig. 3.66. The NACs can be interpreted as follows: The first/last partition in the box is *a* partition in the box that has no previous/next partition. The valid matches are made unique and thus deterministic by construction, i.e., if you *grow* the box via this method, there will always be exactly one first and one last partition.

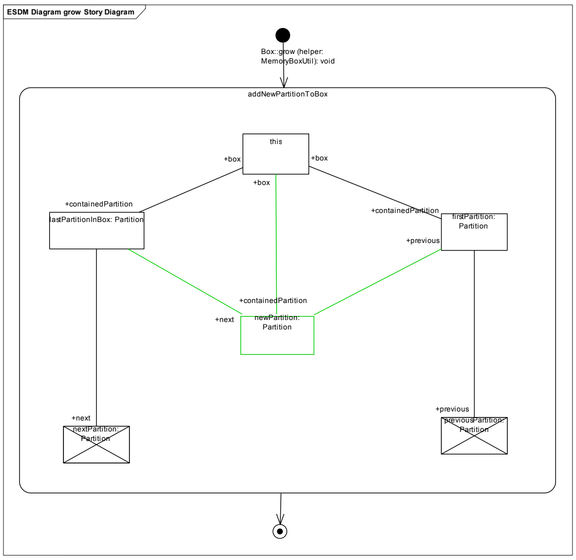



Figure 3.66: Determining the first and last partition with NACs.

Note how the newly created partition `newPartition` is hung into the box (it becomes the next partition of the current last partition and has as its previous partition the first partition in the box) according to the arrows in Fig. 3.1.

All that is missing to complete our SDM is an assignment to set the size of the new partition. We already know that an assignment is an attribute constraint with `:=` as operator so go ahead and invoke the corresponding dialogue. As the new size must be calculated depending on the rest of the partitions in the box (partitions usually get bigger) we call a helper function via a *MethodCallExpression*. A MethodCallExpression is used to invoke a method that is defined in a class in the current EA project. Enter the values in Fig. 3.67 choosing the argument `helper` as target and `determineNextSize` as the method to be invoked. Parameters can be specified by just choosing the appropriate position via the drop-down menu and typing in the value (this is basically a literal expression). Don't forget to press the `Save` button for every parameter and `Add` + `OK` to confirm and close the dialogue.

*MethodCallExpression*

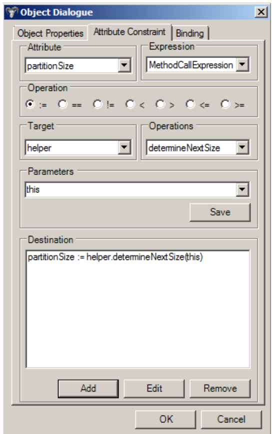



Figure 3.67: Invoking a method via a `MethodCallExpression`.

If you've done everything right, your SDM should now closely resemble Fig. 3.68. As usual, try to export, generate code, inspect the method implementation and write a JUnit test. This time around you also have to implement the helper method `determineNextSize` directly in the generated code (`gen/MemoryBoxLanguage/facade/impl/MemoryBoxUtilImpl`). Don't forget to add `@generated NOT` to the Java doc comment of the method so the code generator preserves your code in future runs.

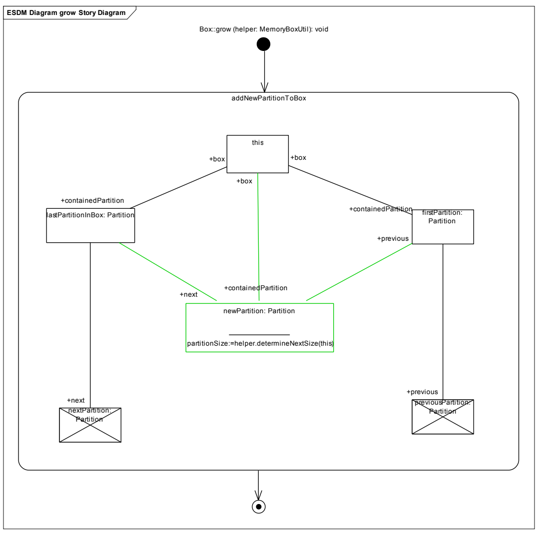



Figure 3.68: Complete SDM for `Box::grow`.

### 3.4.6 A string representation for our memory box

With the next SDM, we shall create a string representation for a complete memory box. To accomplish this, we have to iterate through all cards in all partitions which involves an inner loop nested in an outer loop. SDMs support arbitrary nesting of For Each story nodes via special guards. In Sec. 3.4.3 we already used the `[end]` edge guard to terminate a loop and, as depicted in Fig. 3.69, an `[each time]` guard is used to indicate control flow that is *nested* in the For Each story node and is executed for each match. *[each time]*

Go ahead and create the SDM for `Box::toString` till it closely resembles Fig. 3.69. The first For Each `ForAllPartitions` matches all partitions and each partition is used `[each time]` in `ForAllCards` to match all cards. Note that `partition` in `ForAllCards` is bound and thus refers to the assigned value determined in `ForAllParitions`. When all cards have been matched, `ForAllCards` terminates or `[end]`s and returns to the outer loop `ForAllPartitions`.

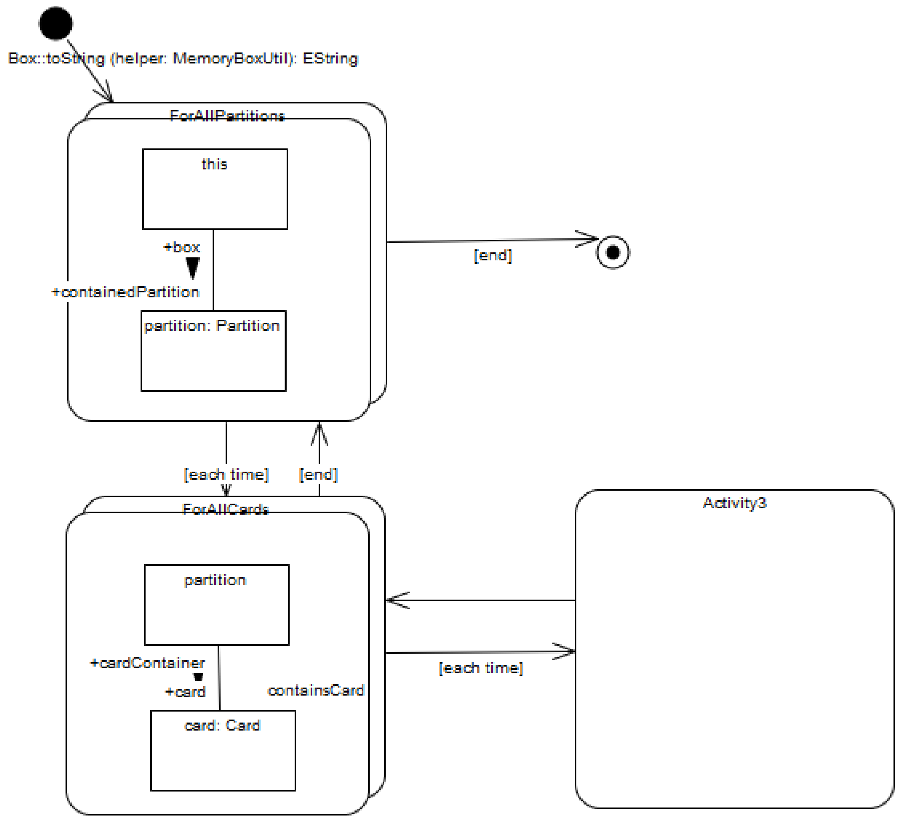



Figure 3.69: Control flow with nested loops.

*Statement Nodes*

To actually do something sensible with each card, double-click the empty activity node that is taken each time a card is matched and invoke the `Edit ActivityNode` dialogue. Now choose `printCard` as the name, and `StatementNode` as the type of the activity node (Fig. 3.70). A statement node is used to invoke a method from a class in any package in the current EA project via a MethodCallExpression. This way, the method invocation is represented as an activity node and is guaranteed to be executed at this point in the control flow.

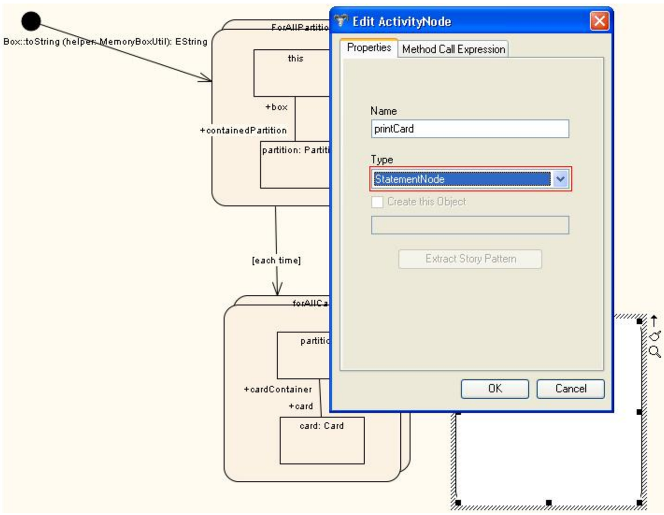



Figure 3.70: Invoking a method in a `StatementNode`.

As we have already used a MethodCallExpression in an attribute constraint (Sec. 3.4.5), go ahead and click the `Method Call Expression` tab and invoke the `printCard` operation on `helper` with the current `card` as its argument (Fig. 3.71).

*Recursion*

Statement nodes should be used to interact with methods that are implemented by hand and provide a means of invoking libraries and arbitrary Java code from SDMs. Please note that we do not differentiate at this point between methods that are implemented via an SDM or by hand and thus, statement nodes can of course be used to invoke other SDMs via a MethodCallExpression. Most importantly, this enables *recursion* as the current SDM can be invoked on `this` with appropriate new arguments.

A final point to note is that the return value of the method is ignored – statement nodes are therefore best used for void methods that either have appropriate side effects (e.g. manipulate their arguments). We shall learn in a few pages how to invoke methods with non-primitive return values (if a method returns a primitive then it can be invoked in an attribute constraint as in Sec. 3.4.5).

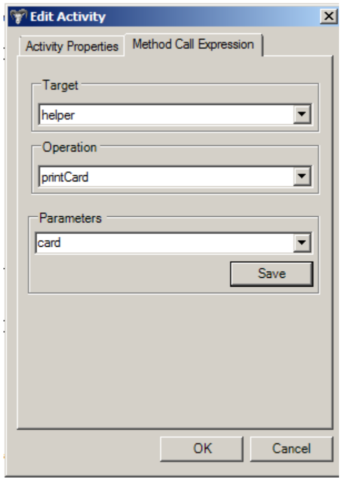



Figure 3.71: Specify a `MethodCallExpression` in the `StatementNode`.

To complete the SDM, retrieve the final string representation from the helper by returning via a MethodCallExpression in the stop node (Fig. 3.72).

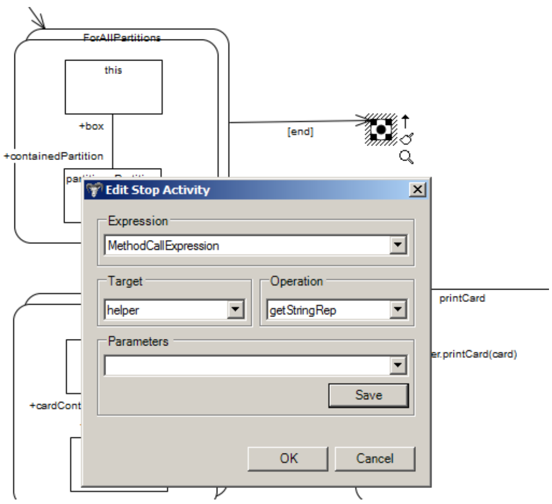



Figure 3.72: Using a `MethodCallExpression` as a return value.

Take some time to compare and reflect on the complete SDM as depicted in Fig. 3.73. The idea was to abstract from the actual text representation of the box and model the necessary traversal of the data structure. The helper methods `printCard` and `getStringRep` could, for example, build up a string buffer and return the final string representation respectively. While modelling this SDM, we have seen that for each story nodes can be nested, and have learnt two new uses of MethodCallExpressions that provide a type safe[13] means of invoking methods from SDMs.

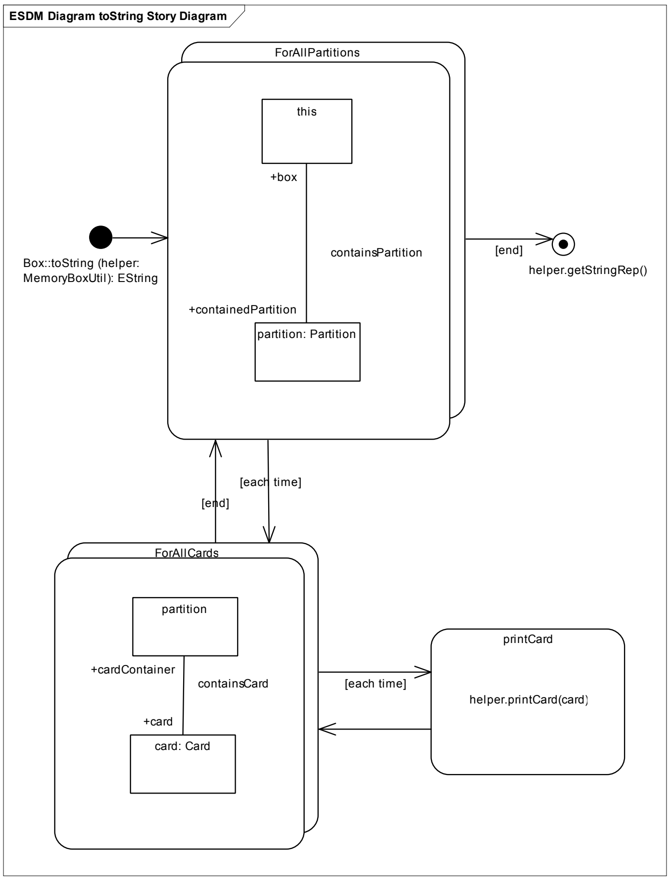



Figure 3.73: The complete SDM for `Box::toString`.

As usual, export, generate code, inspect the generated method implementations and write some tests. Just like in Sec. 3.4.5, don't forget to implement the helper methods!

[13] Apart from the literal expressions used for specifying argument values.

### 3.4.7 Handling "fast" cards

For very simple cards (e.g. words in different language that are so similar), it might be a bit annoying to have to answer these cards again and again in every partition. Such *fast* cards can be marked as such and handled differently: If a fast card is gotten right once it should be immediately moved to the last partition in the box. This way the card is learnt once and is tested only once more before it is finally removed from the box.

To introduce fast cards to our memory box go to the metamodel and create a new eclass `FastCard`. Quick link to `Card` and choose `Inheritance` from the quick link context menu (Fig. 3.74).

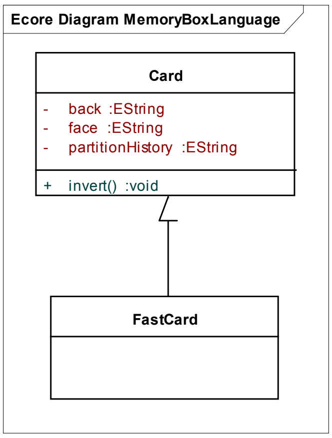



Figure 3.74: Fast cards are a special kind of card.

Now go to the SDM `check` in `Partition` and extend the control flow as depicted in Fig. 3.75. Add new story nodes `is fast card?` and `promote fast card` and drag and drop a bound object variable `fastcard` *of type* `FastCard` into `is fast card?`. What we need to do now is decide, based on the dynamic type[14] of `card` if we must handle a fast card or not.

This can be expressed in SDMs via *BindingExpressions* or just *Bindings*. A binding can be specified for a *bound* object variable and represents the final case where an object variable can be marked as being bound. *Bindings*

To refresh your memory, we have already learnt that a bound object variable is either (1) assigned to `this`, (2) a parameter of the method, or (3) a value determined in a preceding activity node. Bindings represent a fourth possibility of giving a manual binding for an object variable.

[14] In a statically typed langauge like Java, every object has a static type (determined at compile time) and a dynamic type (that can only be determined at runtime).

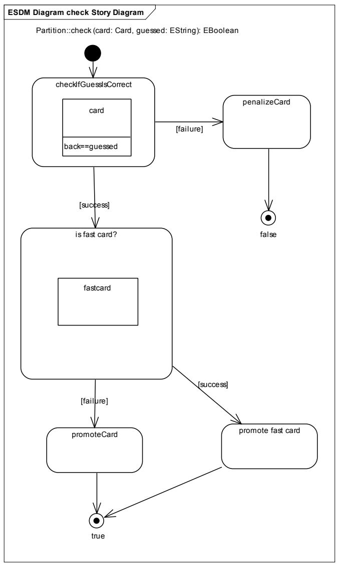



Figure 3.75: Extend check to handle fast cards.

To create a binding for `fastcard`, choose the `Binding` tab in the `Object Variable Properties` dialogue (Fig. 3.76).

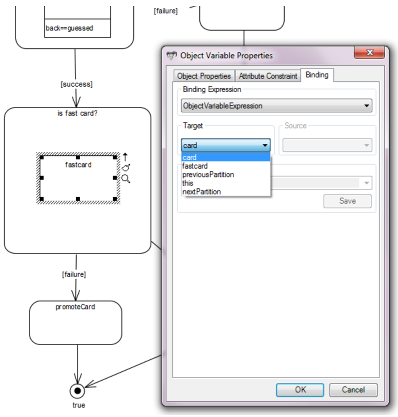



Figure 3.76: Create a binding for `fastcard`.

As usual all our expression types can be used for the `Binding Expression`. Since we already know all the types let's consider what each type would mean in this context:

**MethodCallExpression:**
: This would allow invoking a method and binding its return value to the object variable. This is how non-primitive return values of methods can be used safely in SDMs.

**ParameterExpression:**
: This could be used to bind the object variable to a parameter of the method. If the object variable is of a different type than the parameter (e.g. a subtype) this represents basically a successful typecast if the pattern matches.

**LiteralExpression:**
: As usual this can be anything and is literally copied with a surrounding typecast into the generated code. Using LiteralExpressions too often is usually a sign for not thinking in a *pattern oriented* manner and is considered a *bad smell*.

**ObjectVariableExpression:**
: This can be used to refer to other object variables in preceding story nodes. Just like for ParameterExpressions, this represents a simple typecast if the types of the `target` and the object variable with the binding are different.

In our case, we could use a ParamterExpression or an ObjectVariable-Expression as `card` is indeed a parameter *and* has already been used in `checkIfGuessIsCorrect`. As we haven't used ObjectVariableExpressions before lets try it out! Choose `ObjectVariableExpression` as the type of the binding expression and `card` from the drop-down menu as the target. If you've done everything right, the binding should be visualised concisely as in Fig. 3.77.

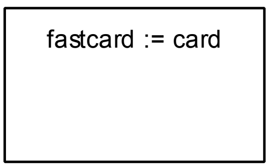



Figure 3.77: Visualisation for binding expression.

To complete the SDM, extract the story pattern of `promote fast card` and specify the pattern according to Fig. 3.78. The fast card is transferred from the current partition `this`, to the last partition in the box, which is identified with an appropriate NAC (already used in Sec. 3.4.5).

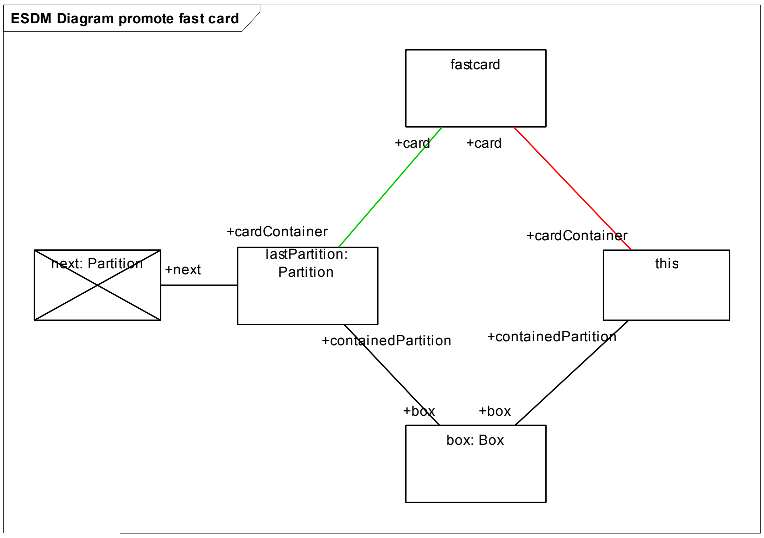



Figure 3.78: Story pattern for handling fast cards.

Export, generate code and inspect the new implementation for `check`. Can you find the generated type casts for `fastcard`? Don't forget to write a few tests and see if fast cards are handled correctly!

# Chapter 4

# Model-to-Text with Moca

When establishing a model-driven solution, *model transformations* usually play a central and important role. Be it for specifying dynamic semantics (like for our memory box) or, more generally, for transforming a certain model to another model to achieve some goal (consistency, adding or abstracting from platform details, ...).

*A Taxonomy of Model Transformations*

There are many *types* of model transformations and [2, 4] give a nice and detailed classification along a set of different dimensions. In this chapter, we shall explore some of these dimensions and learn how *model-to-text* transformations can be achieved with a nice mixture of *string grammars* and *graph grammars*.

*Model-to-Text Transformations*

For the rest of the chapter a model transformation is to be regarded as:

$$\Delta : m_{src} \rightarrow m_{trg}$$

where the source model $m_{src}$ is to be transformed the target model $m_{trg}$.

$\Delta$ is *endogenous*, if $m_{src}$ and $m_{trg}$ conform to the same metamodel. All the SDMs we have treated in the tutorial till now (for our memory box) are examples of endogenous transformations.

*Endogenous Model Transformations*

$\Delta$ operates *in-place*, if $m_{src}$ is destructively transformed to $m_{trg}$. The SDMs for our memory box are also examples for in-place transformations as they perform changes directly to a source model, transforming it destructively into the target model.

*In-Place Model Transformations*

$\Delta$ is *exogenous*, if $m_{src}$ and $m_{trg}$ are instances of different metamodels. In this chapter, we shall complement our memory box with a simple language for *dictionaries*. A dictionary is also used to learn new words but is more suitable to be used as a reference, i.e., one already knows most of the words

*Exogenous Model Transformations*

and only specific words are looked-up now and then. A memory box, on the other hand, is more geared towards supporting the actual memorization process. Ergo? One could start with a memory box and, when all words have been memorized, transform it to a personalized dictionary for future reference. If one notices that too many words have been forgotten (typically after a long break or a lazy spell) a dictionary can be transformed *back* to a memory box. We shall see later on that this transformation is actually quite cool as one could, for example, use the history of cards or their difficulty level (fast cards are very simple) to either annotate entries in a dictionary or pre-place cards appropriately in a memory box.

The memory box to dictionary transformation and vice-versa are examples of exogenous transformations.

*Out-Place Model Transformations*

$\Delta$ is *out-place* if $m_{src}$ is left intact and is not changed by the transformation that creates $m_{trg}$. The memory box to dictionary transformation and vice-versa are also examples of out-place transformations.

Although endogenous + in-place is the natural case for SDMs (like for our memory box), we shall see in a moment that exogenous and/or out-place transformations can also be specified with SDMs.

To twist your brain a bit here are a few interesting statements:

- Out-place transformations can be endogenous or exogenous.
- In-place transformations can usually[1] only be endogenous. Exogenous transformations are, consequently, always out-place. Why?

*Horizontal or Vertical?*

$\Delta$ is further classified as *horizontal* if $m_{src}$ and $m_{trg}$ are on the same *abstraction level* and *vertical* if they are not.

*Abstraction Levels*

This last abstraction-level dimension is unfortunately a bit fuzzy but in a moment we shall explore and work on different abstraction levels by establishing a textual concrete syntax for our dictionaries.

[1] One can always think up crazy examples right?

In the process we shall learn how graph transformations can be used, in combination with parser generators and template languages, to implement model-to-text and text-to-model transformations that are typically vertical (text is normally on a lower abstraction level than a model).

Our memory box to dictionary transformation is, on the other hand, probably horizontal as the models represent the *same* information, albeit differently, and can thus be considered to be on the same abstraction level.

In the following the *Mo*flon *C*ode *A*dapter (*Moca*) framework refers to:

*What is Moca?*

1. the approach we use to integrate string grammars, graph grammars and template languages,
2. how we separate the transformation into different modular steps,
3. the usage of a generic and simple tree to consolidate different platforms, and
4. the actual tool support that acts as glue to hold all the different parts together.

Fig. 4.1 gives a "big picture" of what we plan to achieve in this chapter. All explanations are integrated right in the figure so take your time and let it sink in. We'll be zooming in on bits and pieces in the following sections to make things clearer and more concrete.

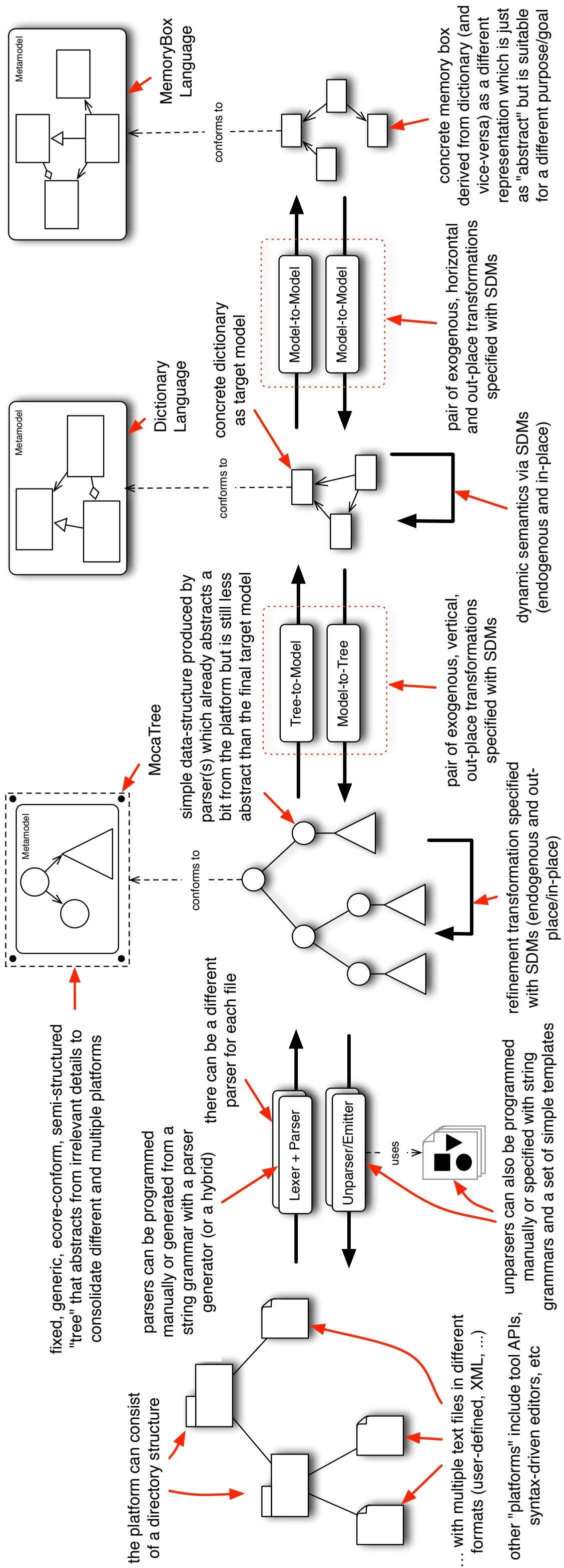



Figure 4.1: Overview of model-to-text with the MOCA framework

## 4.1 Set up ANTLR

Our first step will be to install and set up a *parser generator*. Nowadays, *no one* really writes a complex parser completely by hand. Although this is sometimes still necessary[2] most parsers can be whipped up pretty quickly using context-free *string grammars*[3] typically in EBNF[4]. ANTLR [5] is a tool that can generate a parser from a compact EBNF specification for a host of target programming languages, including Java. Although ANTLR might not be the most efficient or powerful parser generator out there, it is open-source, well documented and supported, and allows for a pragmatic and quite elegant *fallback* to Java when things get nasty and we have to resort to some dirty tricks.

- Install the ANTLR-IDE[5] Eclipse plugin from:
  `http://antlrv3ide.sourceforge.net/updates`.

  We suggest you use ANTLR-IDE as it integrates nicely with eMoflon (the same build function for generating code also triggers the parser generator) and offers adequate editor functionality.

- Download ANTLRWorks from:
  `www.antlr.org/download/antlrworks-1.4.3.jar`.

  ANTLRWorks[6] is the IDE recommended by [5]. It offers a nice debugger and visualization of parse trees and abstract syntax trees. You are free to use ANTLRWorks either together with ANTLR-IDE or as an alternative. For all screenshots and explanations, however, we shall assume you choose to work with ANTLR-IDE.

  Now choose `Directory` and browse to the directory with the downloaded ANTLRWorks jar file (Fig. 4.2).

- Configure ANTLR for Eclipse:
  Go to "Window/Preferences/ANTLR/Builder" and choose `Add` as depicted in Fig. 4.3.

  As ANTLR-IDE does not provide the actual jars for the parser generator, these have to be downloaded and referenced from the plugin.

[2] Some languages are syntactically quite challenging.
[3] For simple cases, *regular expressions* can also be used.
[4] Extended Backus-Naur Form
[5] `http://antlrv3ide.sourceforge.net/`
[6] `http://www.antlr.org/works/index.html`

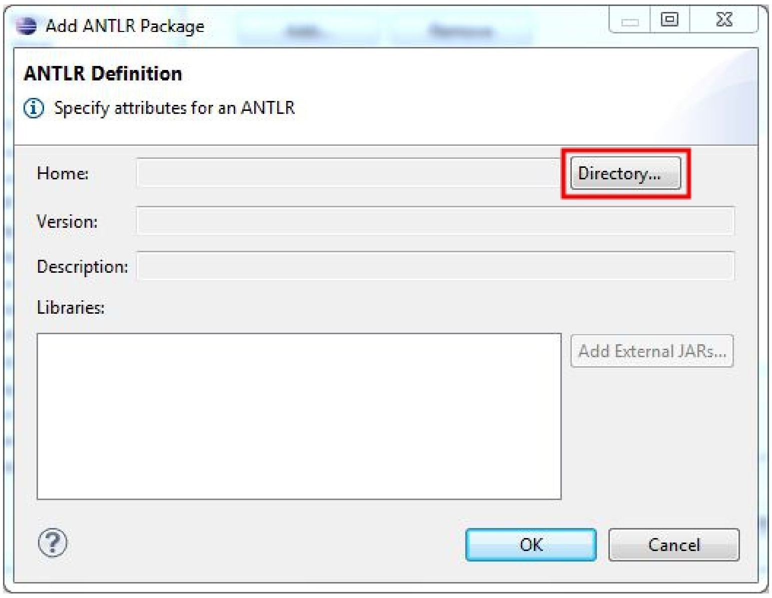



Figure 4.2: Dialogue for choosing ANTLRWorks as Builder

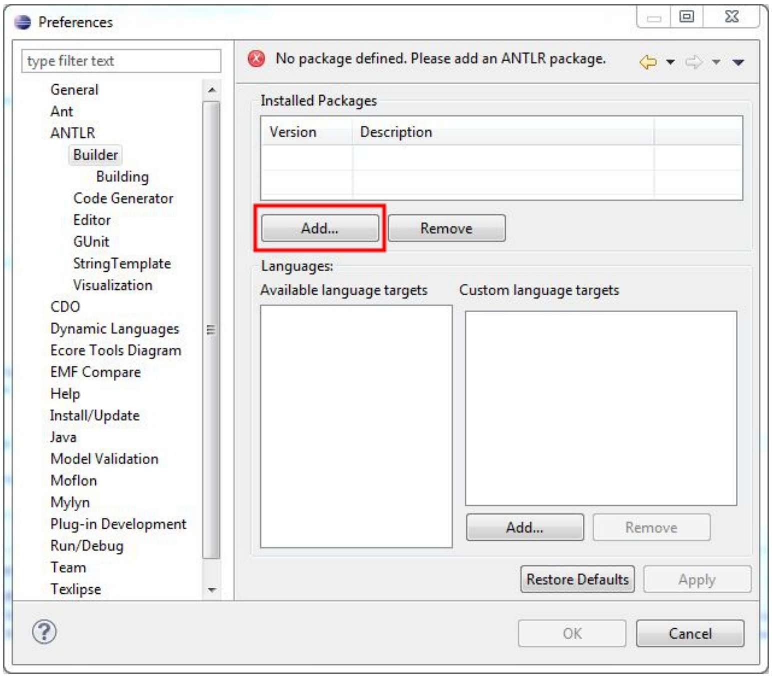



Figure 4.3: Builder Preferences for ANTLR IDE

## 4.2 Set up your M2T Workspace

The next step is preparing an Eclipse workspace according to our suggested workflow.

▶ In Eclipse (preferably with an empty workspace), switch to our Moflon perspective and invoke the `New Metamodel` wizard (Fig. 4.4).

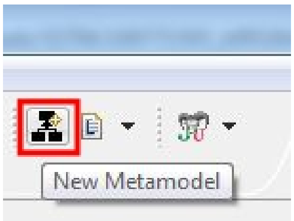



Figure 4.4: Invoking the `New Metamodel` wizard.

▶ Choose "Dictionary" as project name and select `Add Moca Support` as depicted in Fig. 4.5.

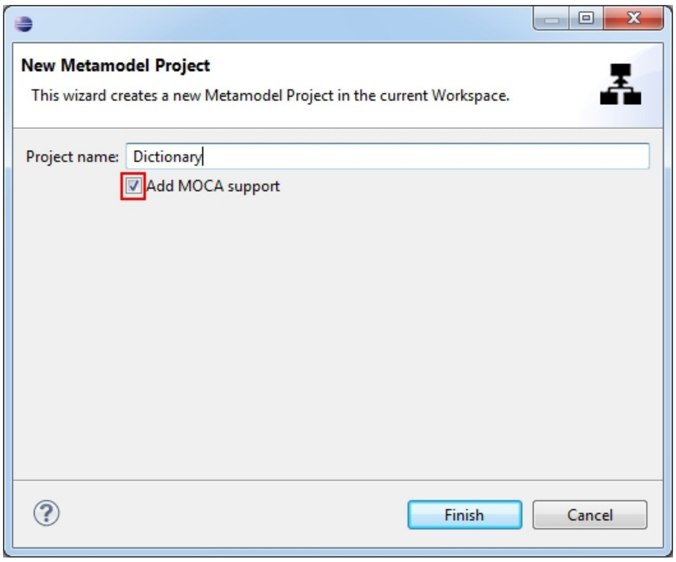



Figure 4.5: Add Metamodel project with MOCA support

- After the project is created as usual (Fig. 4.15) double-click the EAP file to open it.

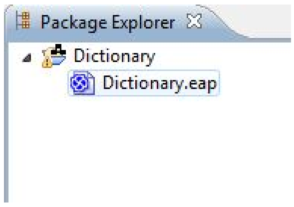



Figure 4.6: Eclipse workspace after creating the `Dictionary` project

- In EA, the project is already populated with the metamodel for our generic tree. To differentiate this from other trees (ANTLR parse tree and abstract syntax tree, XML DOM tree, ...) we shall refer to it as `MocaTree` (Fig. 4.7). Note that the `MocaTree` package has a special tagged value `Moflon::Export` that is set to `false`[7]. This ensures that the package is *ignored* when exporting. As with all standard metamodels (e.g. Ecore or the SDM metamodel) the `MocaTree` package in EA should be regarded as read-only and is only required in the EA project so that SDMs can refer to the classes defined in the package. The corresponding Java code is provided by our Eclipse plugin and is added automatically to the Java build path whenever necessary.

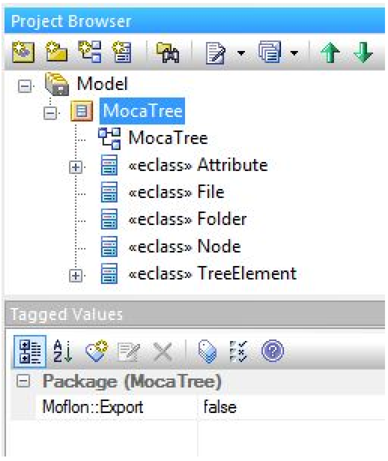



Figure 4.7: `MocaTree` in default EA project

[7]Tagged values can be viewed in the `Tagged Values` view in EA (Fig. 4.7).

Go ahead and inspect the `MocaTree` metamodel (Fig. 4.8). It basically combines concepts from a filesystem (folders and files), XML concepts (text-only nodes and attributes), and a general indexed node containment hierarchy.

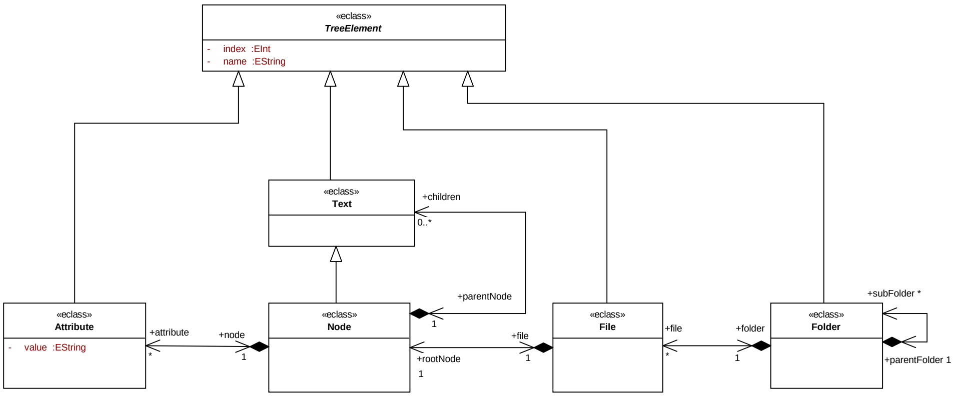



Figure 4.8: Mocatree

- ▶ Add a new package `DictionaryLanguage` and model the required classes and relationships for our dictionary language (Fig. 4.9).

  Every dictionary has a title and consists of entries. Entries have a content and a level that indicates how difficult the entry is. Dictionaries can be organized in shelves that have a description and shelves can be collected in a library. To make things interesting, each dictionary has an author. Note that arbitrary many different dictionaries, irrespective of their shelves, can share the same author.

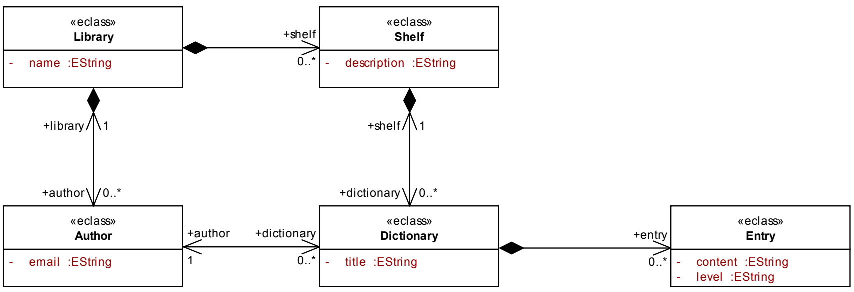



Figure 4.9: Dictionary metamodel

► For the moment, add an empty package in EA named `Dictionary-CodeAdapter` so that your EA workspace closely resembles Fig. 4.10.

According to our conventions and workflow, a *code adapter* is a package that contains the tree-to-model transformation logic. This could of course be integrated directly in the corresponding metamodel (`DictionaryLanguage` in our case), but a separation makes sense as there could be *different* code adapters for the *same* language.

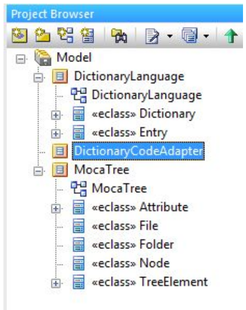



Figure 4.10: EA workspace before exporting

► Export as usual and ensure that your Eclipse workspace closely resembles Fig. 4.11. Note especially the library nodes (`Moflon` and `Moca`) that reference jars for all required dependencies.

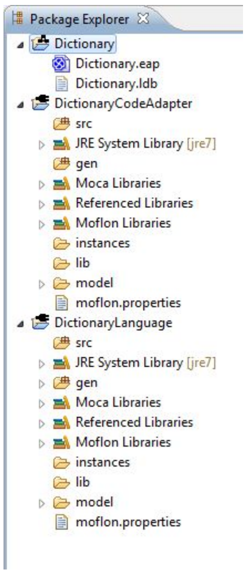



Figure 4.11: Workspace after export to Eclipse

- ▶ Right-click on the project `DictionaryCodeAdapter` and choose "Convert to ANTLR Project . . . " from the Configure context menu (Fig 4.12).


Figure 4.12: Convert CodeAdapter project to an ANTLR project

- ▶ Right-click on `DictionaryCodeAdapter` one more time and choose "Add Parser/Unparser", this time from the eMoflon context menu (Fig 4.13).

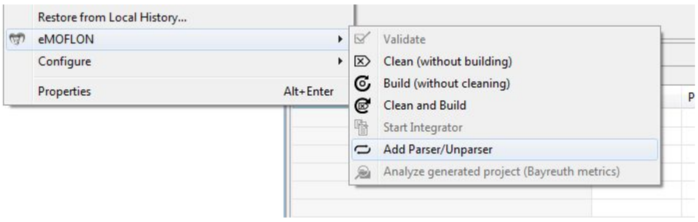



Figure 4.13: Invoking the `Add Parser/Unparser` wizard

- ▶ In the wizard dialogue (Fig 4.14), enter "dictionary" as file extension, and check the boxes `Create Parser` and `Create Unparser` with `ANTLR` chosen as corresponding technology in both cases. Click `Finish`.

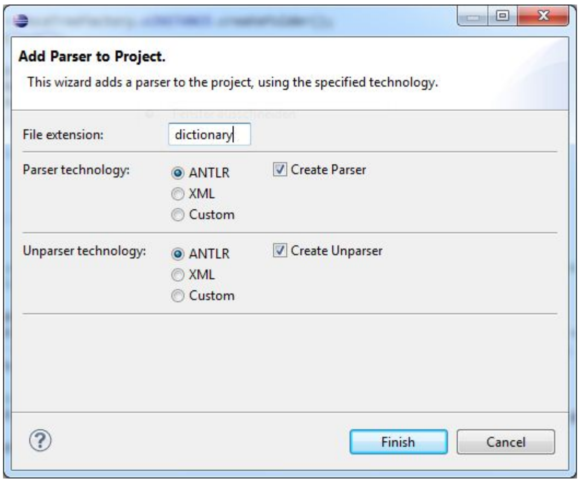



Figure 4.14: Add Parser/Unparser

If everything has been installed and set up properly, parser and unparser stubs should be generated and ANTLR should automatically build the corresponding Java code as depicted in Fig. 4.15.

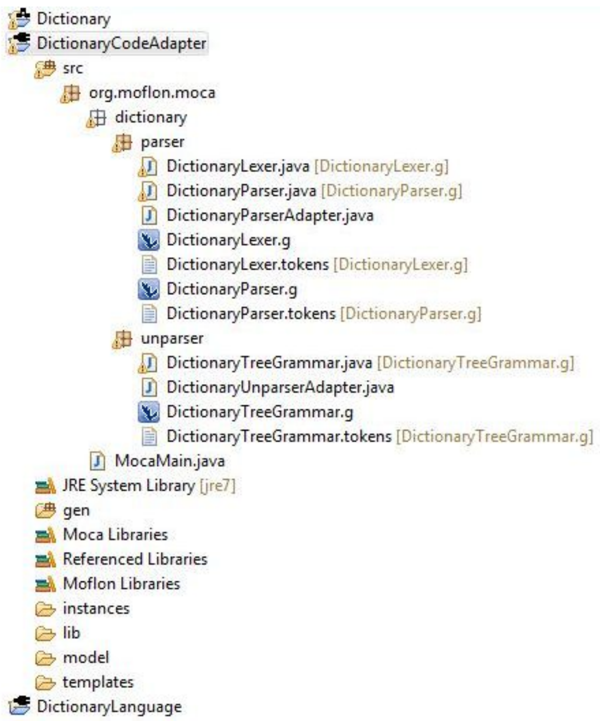



Figure 4.15: Workspace after wizard finishes

## 4.3 Text-to-Tree Transformation

As we shall see in a moment, libraries and shelves correspond to a folder structure while the contents for a single dictionary are specified in a file. Figure 4.16 depicts a small sample of the textual syntax used to specify a dictionary. On the way to an instance model of our dictionary metamodel, the very first step is to create nice *chunks* of characters. This is called *lexing* and is a step that simplifies the actual comprehension of the complete text. Interestingly human beings actually comprehend text in a similar manner, one recognizes whole words without “seeing” every individual character. This is the reason why you can siltl raed tihs sneentce alsomt eforftlsesly. A lexer recognizes these chunks or *tokens* and passes them on as a token stream to the *parser* that does the actual work of recognizing complex hierarchical and recursive structures.

To recognize the tokens as indicated in Fig. 4.16, `ANTLR` can automatically generate a lexer in Java from a compact specification as depicted in Fig. 4.17. This is actually a DSL for lexing and is explained in detail in [5]. If you do not know what EBNF is and have problems understanding the lexer grammar then make sure you at least go through the documentation on `www.antlr.org` or read relevant chapters in [5].

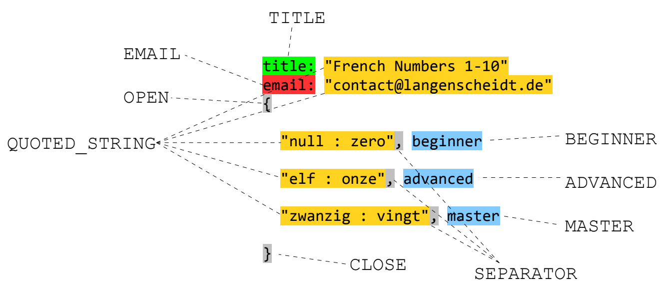



Figure 4.16: Identified tokens in a dictionary file.

► Edit `DictionaryLexer.g` so it closely resembles Fig. 4.17. Be careful to avoid any typos and mistakes. Save and make sure it compiles.

```
lexer grammar DictionaryLexer;

@header {
package org.moflon.moca.dictionary.parser;
import org.moflon.moca.MocaUtil;
}

WS: (' ' | '\t' | '\n' | '\r')+  { skip(); };

TITLE: 'title:';

EMAIL: 'email:';

QUOTED_STRING:   '"' .* '"' { MocaUtil.trim(this, 1, 1); };

BEGINNER: 'beginner';

ADVANCED: 'advanced';

MASTER: 'master';

OPEN: '{';

CLOSE: '}';

SEPARATOR: ',';

DICTIONARY: 'DICTIONARY';

ENTRY: 'ENTRY';
```

Figure 4.17: Lexer grammar

The next step is to form the stream of tokens from the lexer into a *tree*. In this context, a tree is an acyclic, hierarchical, recursive structure as depicted in Fig. 4.18. Depending on what the tree is to be used for, it can be organized very differently with extra *structural* nodes like `DICTIONARY` or `ENTRY` that were not present in the textual syntax and are used to give additional meaning to the tree.

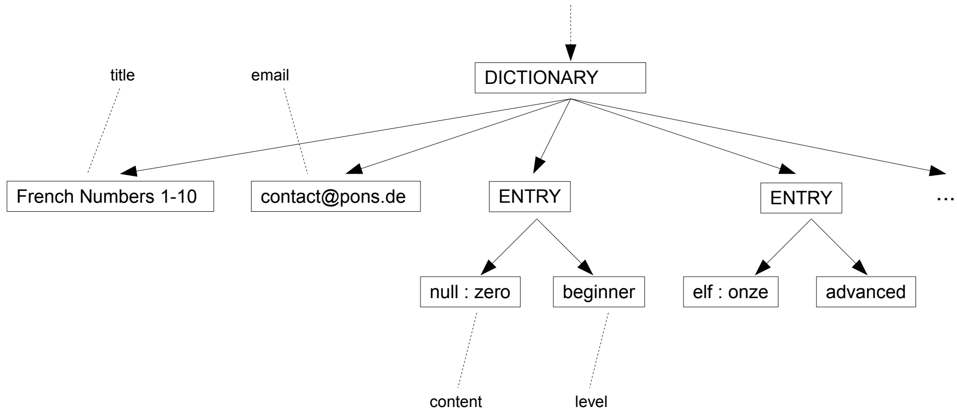



Figure 4.18: MocaTree structure

► Edit `DictionaryParser.g` so it closely resembles Fig. 4.19. As with the lexer, avoid typos and mistakes and make sure it compiles.

```
parser grammar DictionaryParser;

options {
  language   = Java;
  tokenVocab = DictionaryLexer;
  output     = AST;
}

@header {
package org.moflon.moca.dictionary.parser;
}

main: title email? OPEN entry+ CLOSE  -> ^(DICTIONARY title email? entry+ );

title: TITLE QUOTED_STRING -> QUOTED_STRING;

email: EMAIL QUOTED_STRING -> QUOTED_STRING;

entry: QUOTED_STRING SEPARATOR level  -> ^(ENTRY QUOTED_STRING level);

level: ( BEGINNER | ADVANCED | MASTER );
```

Figure 4.19: Parser grammar

The parser grammar is quite similar to the lexer grammar, but there are *parser actions* after `->` to build up the tree. Using this simple tree language, one can (1) abstract from tokens like { or }, which are just *syntactical noise*[8] and (2) enrich the tree with imaginary nodes like `ENTRY`, which add explicit structure to the tree. Please refer to [5] and online resources for a detailed explanation of the syntax and semantics of the parser grammar supported by `ANTLR`.

```
1  package org.moflon.moca;
2  import java.io.File;
11
12 public class MocaMain
13 {
14
15   private static CodeAdapter codeAdapter;
16
17   public static void main(String[] args)
18   {
19     BasicConfigurator.configure();
20
21     // Register parsers and unparsers
22     codeAdapter = MocaFactory.eINSTANCE.createCodeAdapter();
23     codeAdapter.getParser().add(new DictionaryParserAdapter());
24     codeAdapter.getUnparser().add(new DictionaryUnparserAdapter());
25
26     // Perform text-to-tree
27     Folder tree = codeAdapter.parse(new File("instances/in/"));
28
29     // Save tree to file
30     eMoflonEMFUtil.saveModel(MocaTreePackage.eINSTANCE, tree, "instances/tree.xmi");
31
32     // Transform tree-to-model
33     //TODO
34
35     // Save model to file
36     //TODO
37
38     // Transform model-to-tree
39     //TODO
40
41     // Perform tree-to-text (using initial tree)
42     codeAdapter.unparse("instances/out", tree);
43   }
44 }
```

Figure 4.20: Generated main method

Before we take our lexer and parser for a spin, open `MocaMain.java` and inspect it. If everything went right it should bear a striking resemblance to Fig. 4.20. For the moment we do not need to adjust anything, just note how the parser is added to the Moca framework (line 23) via an adapter (`DictionaryParserAdapter`). Go ahead and look at what the adapter exactly does. All the code can be adjusted and used to, for example, define which files the parser is to be used for (per default the adapter registers for `*.dictionary` files). The main job of the adapter is to hide `ANTLR` specifics

[8]In this context, content that is irrelevant for our model.

so the framework remains (parser) technology agnostic. If you decide to use a different parser generator or write the parser by hand you would need to implement a corresponding adapter from scratch.

On line 27, the input for the framework is set, meaning that all folders in `./instances/in` are parsed. In a nutshell, each folder is taken as a root of a tree and the folder and file structure is reflected as a hierarchy of (children) nodes in the tree. For each file, the framework searches for a registered parser that is responsible for the particular file, passes the content on to the parser and plugs in the tree from the parser as a single subtree of the corresponding file node in the overall tree. Take a look at Fig. 4.1 again and review the parts we have covered.

The final step is now to prepare some input for the framework:

- ▶ Create the directory structure depicted in Fig. 4.21 in `Dictionary-CodeAdapter` and enter the contents from Table 4.1 for each of the four `dictionary` files[9].

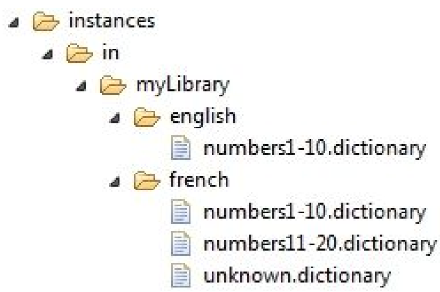



Figure 4.21: Input directory structure.

- ▶ After creating all the dictionaries, run `MocaMain.java` as a normal Java application. If everything works out right, a `tree.xmi` and an `out` folder should be created in the `instances` directory. Inspect their contents and compare to Fig. 4.22. The unparsed files are obviously empty as we haven't implemented an *unparser* yet. Don't be irritated by the fact that an `out/in` is created, this can all be configured in `MocaMain` but the default assumes `in` contains multiple folders – in our case libraries – and it is therefore treated as the root of the tree. One could also unparse directly in `instances` but the unparsed `in` would have to be renamed in `MocaMain`.

[9]You can just copy&paste right from this PDF file

**english/numbers1-10.dictionary:**

```
title: "English Numbers 1-10"
email: "contact@langenscheidt.de"
{
  "null : zero", beginner
  "eins : one", beginner
  "zwei : two", beginner
  "drei : three", beginner
  "vier : four", beginner
  "fuenf : five", beginner
  "sechs : six", beginner
  "sieben : seven", beginner
  "acht : eight", beginner
  "neun : nine", beginner
  "zehn : ten", beginner
}
```

**french/numbers11-20.dictionary:**

```
title: "French Numbers 11-20"
email: "contact@pons.de"
{
  "elf : onze", advanced
  "zwoelf : douze", advanced
  "dreizehn : treize", advanced
  "vierzehn : quatorze", advanced
  "fuenfzehn : quinze", advanced
  "sechzehn : seize", master
  "siebzehn : dix-sept", master
  "achtzehn : dix-huit", master
  "neunzehn : dix-neuf", master
  "zwanzig : vingt", master
}
```

**french/numbers1-10.dictionary:**

```
title: "French Numbers 1-10"
email: "contact@pons.de"
{
  "null : zero", beginner
  "eins : un/une", beginner
  "zwei : deux", beginner
  "drei : trois", beginner
  "vier : quatre", beginner
  "fuenf : cinq", beginner
  "sechs : six", beginner
  "sieben : sept", beginner
  "acht : huit", beginner
  "neun : neuf", beginner
  "zehn : dix", beginner
}
```

**french/unknown.dictionary:**

```
title: "unknown"
{
  "unbekannt", beginner
}
```

Table 4.1: Input files containing dictionaries.

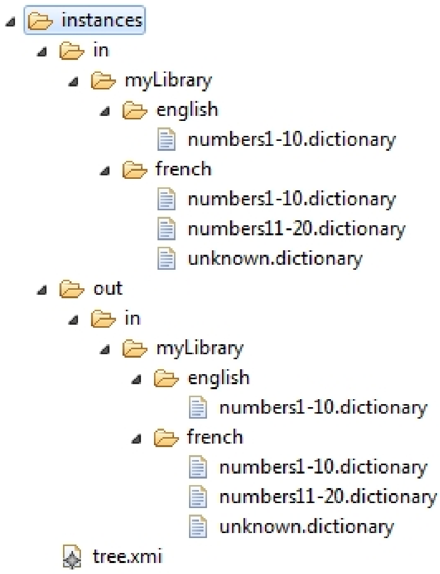



Figure 4.22: Directory `instances` after parsing

▶ Double-click `tree.xmi`[10] and compare the contents to Fig. 4.23. At this point, you can reflect on the structure of the tree and note the directory structure, file nodes and the subtrees from the parser.

This is an important point to understand; the directory structure is transformed to a corresponding hierarchy of `Folders` and `Files`. The actual *textual content* of each file is then transformed to a subtree using a registered, suitable parser. The resulting subtree from the parser is then plugged into the tree by setting its root as the single child node of a `File`.

If everything worked out then well done! We now have a nice tree that we can work on with SDMs and transform in a few simple steps to an actual instance of our Dictionary metamodel.

[10] Depending on the plugins you have installed, you might have to explicitly choose `Open With/Sample Reflective Ecore Model Editor`.

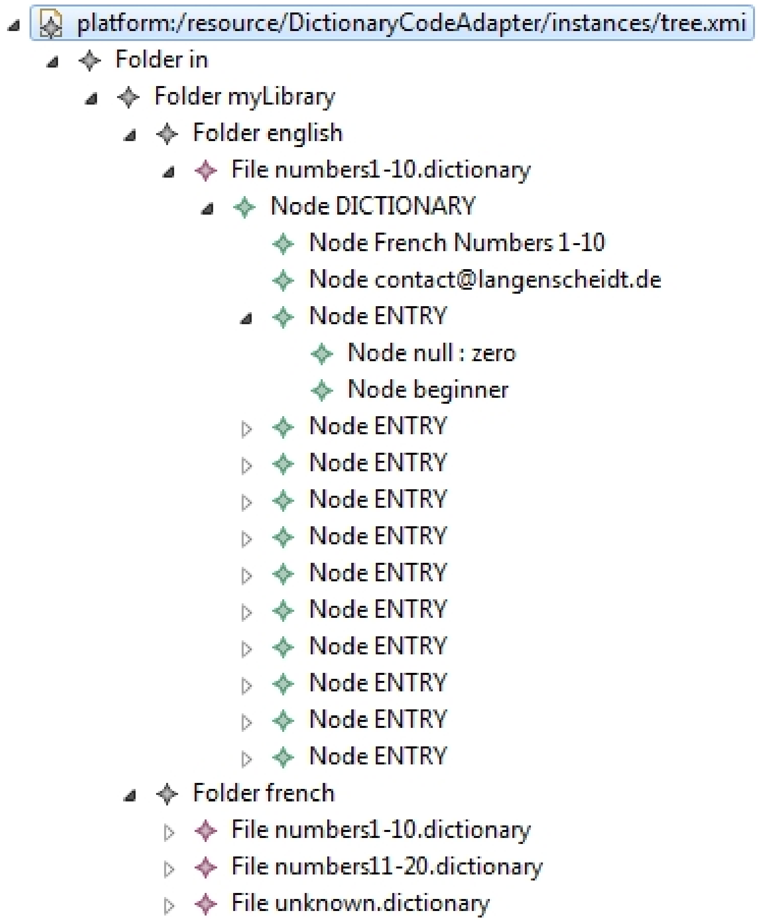



Figure 4.23: MocaTree created by the framework using our parser

## 4.4 Tree-to-Model Transformation with SDMs

The next step in the text-to-model transformation is a set of SDMs that transform our tree to an instance of our dictionary metamodel. Take a look at the overview (Fig. 4.1) again and try to identify which arrow depicts exactly this *tree-to-model* transformation. Just a short comment; all SDMs in this section depict story patterns directly in their story nodes. Please do not take this as a *best practice*, in fact we actually recommend always extracting story patterns. It's just easier for the tutorial to fit each SDM on a single page!

▶ In the code adapter project `DictionaryCodeAdapter`, create a `Transformer` class with the methods and reference to the `Library` class as depicted in Fig. 4.24.

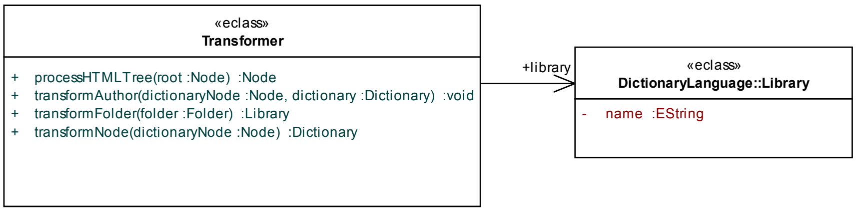



Figure 4.24: Transformer class with methods for SDMs

▶ The first SDM `transformFolder: Folder` → `Library` is depicted in Fig. 4.25 and creates a corresponding library with shelves for the given folder, delegating the creation of dictionaries to `transformNode`. The SDM uses features that we have treated in previous chapters. Note that the exogenous transformation is specified by simply drag & dropping in elements from *different* metamodels as required! All dependencies are automatically added when exporting to Eclipse – now isn't that as simple as ABC?

▶ The next SDM, `transformNode: Node` → `Dictionary` (Fig. 4.26) takes a node, representing a dictionary, and builds up a dictionary object adding entries appropriately. It further delegates creation of authors to `transformAuthors`. Note how *indices* are used in the story node `match entry node` to decide, according to convention (how we built the tree), which node in the tree is to be interpreted as content and which as the level of the entry.

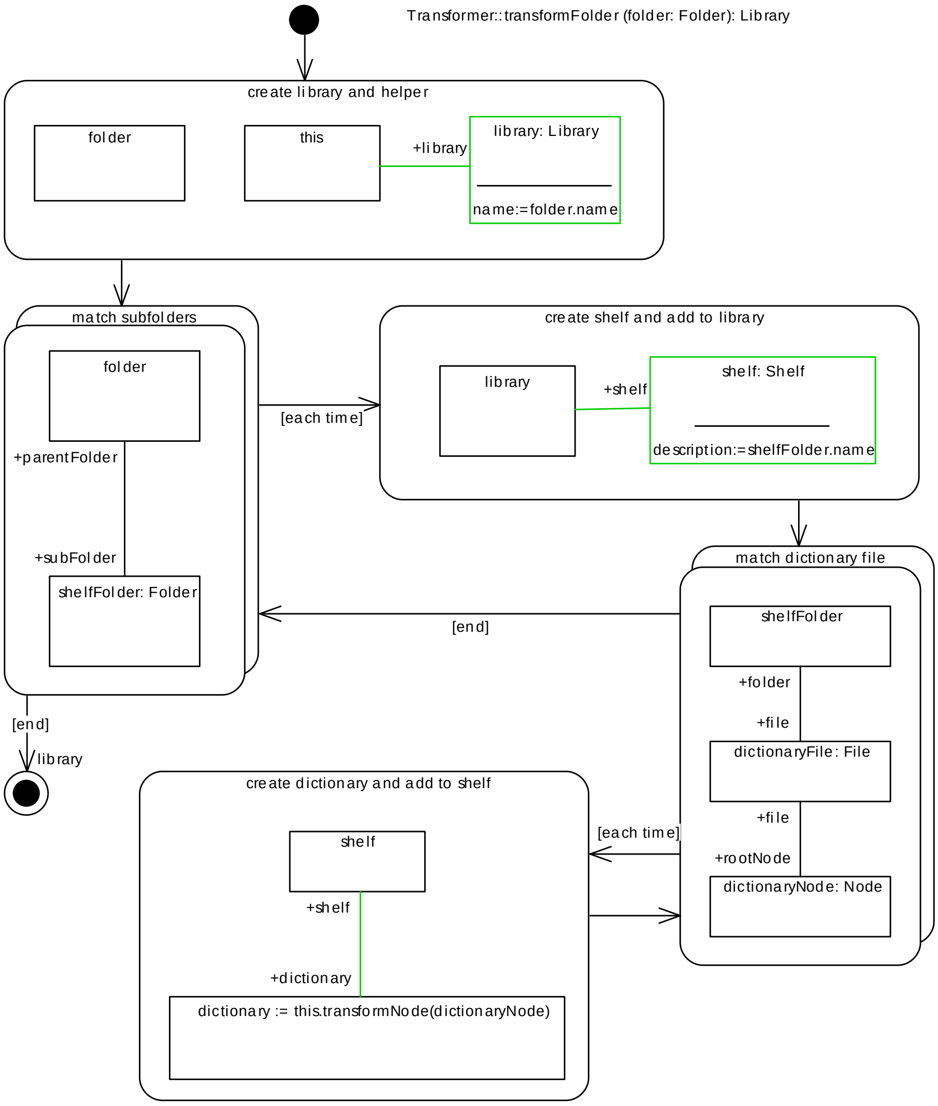



Figure 4.25: Transforming the outermost folder into a dictionary

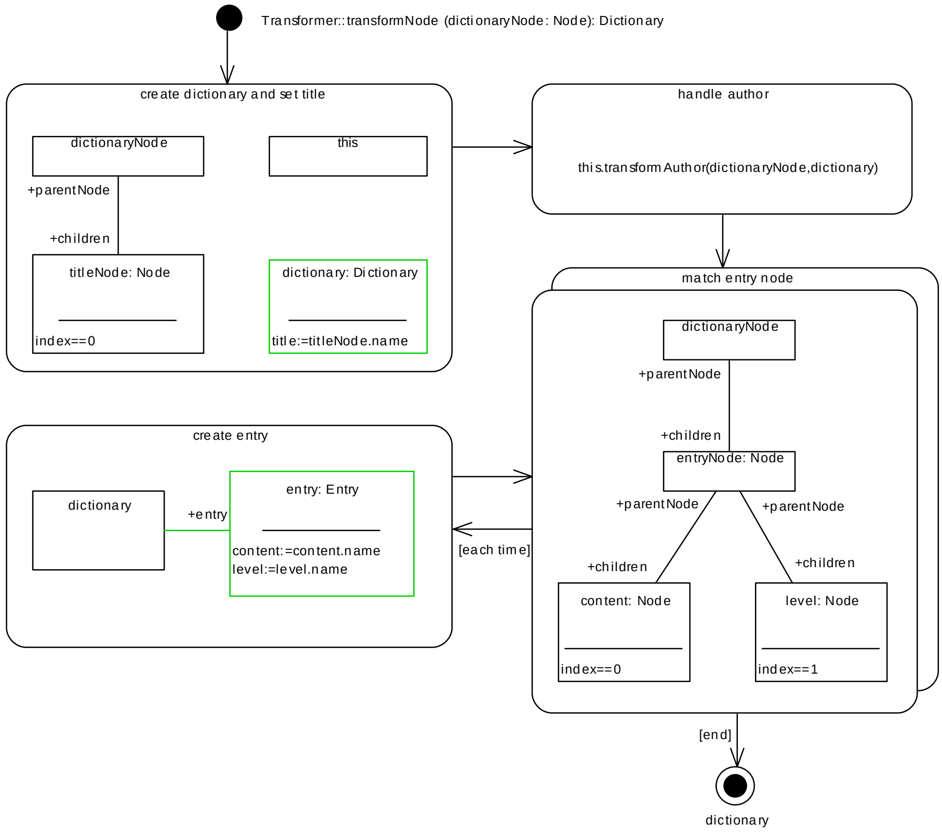



Figure 4.26: Creating dictionaries from dictionary nodes

▶ To wrap things up, create the last SDM, `transformAuthor: Node, Dictionary` → `()` as depicted in Fig. 4.27. This SDM checks in `match author node` if the node with `index` 1 is *not* an entry. Again according to convention, this would be an author node which is optional. If no such node exists we do not create an author and simply return. If such a node does exist, a further complication is that the author might already be known in the library. In order to avoid multiple, actually identical authors, the author object variable in `create author` is set to `optional` and to `create`. *Optional Create*

The semantics of *optional create* is the following: if a match for the object variable with the specified attribute *constraints* is found, it is used. If no match can be found then the object variable is created and the specified attribute assignments are carried out.

This is exactly how we need to handle authors – cool right?

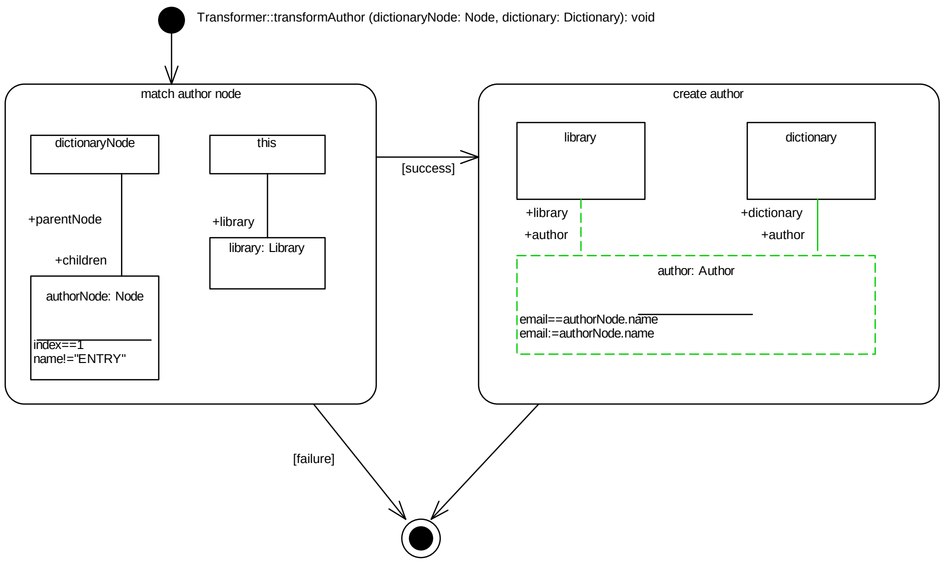



Figure 4.27: SDM transformAuthor

► As a final step, open `MocaMain.java` (Fig. 4.20) and edit lines 32 – 39 as follows:

```
// Transform tree-to-model
Transformer transformer =
  DictionaryCodeAdapterFactory.eINSTANCE.createTransformer();
Library library = transformer.transformFolder(tree);

// Save model to file
eMoflonEMFUtil.saveModel(DictionaryLanguagePackage.eINSTANCE,
  library, "./instances/library.xmi");
```

Open and inspect the library model using the reflective model browser and note especially the cross-tree references between authors and their dictionaries.

# Chapter 5

# Conclusion

Wow! If you've really worked through everything till now (and wrote all those tests), then you can consider yourself a *bona fide* SDM wizard. Get yourself a nice cool beer – you've earned it!

We hope you enjoyed the trip, if you have any suggestions, questions, feedback or corrections (all those screenshots get outdated so quickly!), please contact us `contact@moflon.org`.

Our tool *eMoflon* is constantly evolving so don't forget to check out what's new at `www.moflon.org`

# Appendix A

# References

[1] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 2005.

[2] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Online Proc. of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Anaheim, California, USA, October 2003.

[3] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, Berlin, 2006.

[4] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(GraMoT):125–142, 2006.

[5] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

# Appendix B

# GNU GENERAL PUBLIC LICENSE

GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007



Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work’s users, your or third parties’ legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAM-