An Introduction to Metamodelling and Graph Transformations

with eMoflon

Version 3.3

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

The eMoflon team
Darmstadt, Germany (January 2013)

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.

As the tutorial is quite a few pages long, here are a few suggestions of what you must read or can skip depending on what you're interested in:

Chapter 2 provides 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.

This chapter can be considered *mandatory* if you are new to eMoflon and we recommend working through it in any case. It's also kept as minimal as possible and should only take a few minutes really.

Chapter 3 is the main chapter and takes you 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.

You should definitely work through this chapter if you're new to metamodelling in general (using Ecore/EMF) and if you're interested in model transformation via graph transformation using *Story Driven Modelling* (SDM).

Chapter 4 gives a few tips and tricks for using our frontend Enterprise Architect (EA) effectively and avoiding typical mistakes.

If you're in a hurry this chapter can be skipped and used as a reference when you get stuck with EA or can't figure out how to do something.

Chapter 5 treats *model-to-text* transformation and how eMoflon can be used together with parsers and template languages.

If you're only interested in model-to-model transformation then you can safely skip this chapter and come back to it only when you need to

generate code or extract your models from XML or some other textual format.

Chapter 6 introduces *Triple Graph Grammars* (TGGs) used for bidirectional model transformation.

Feel free to jump directly here after working through at least Chap. 2 if you're mainly interested in TGGs. Although the example builds up on parts constructed in previous chapters, we provide a "cheat" package that you can use to get started directly.

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 *extensions* to support new modelling languages.

- Download EA for Windows from `http://www.sparxsystems.com/` to get a free 30 day trial and follow installation instructions (Fig. 2.1).

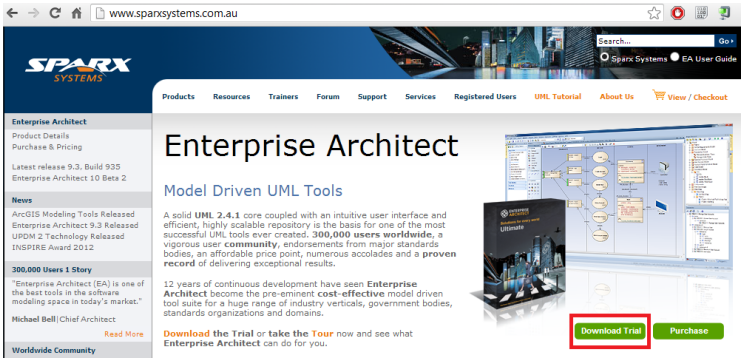


Figure 2.1: Download Enterprise Architect

- 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`.

[1] Unified Modelling Language

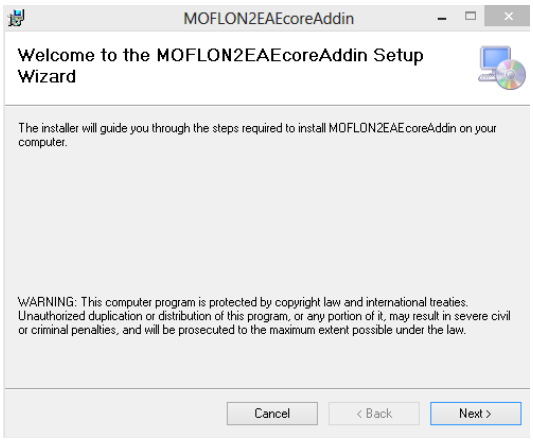


Figure 2.2: Install our extension for EA

2.2 Install our plugin for Eclipse

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

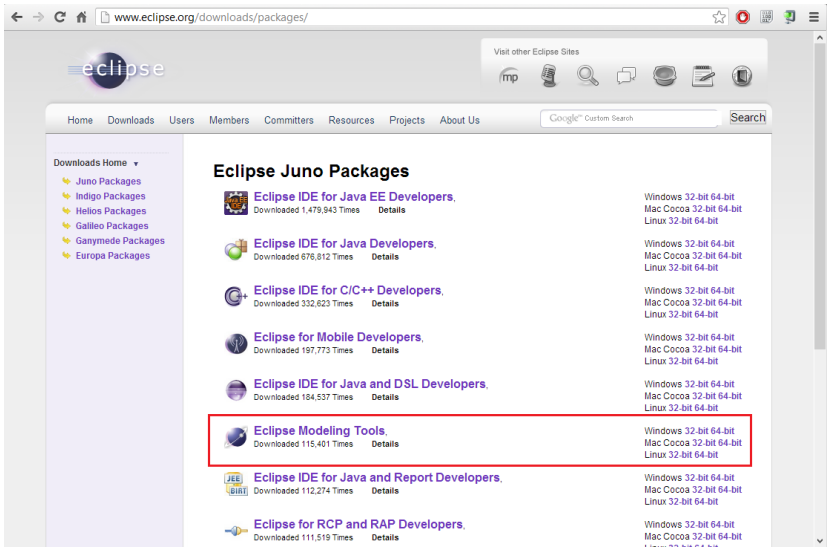


Figure 2.3: Download Eclipse Modeling Tools.

[2]Please note that you *have to* install *Eclipse Modeling Tools* or nothing will work. Do not choose a different Eclipse package! Although different versions and constellations might work, eMoflon is currently tested for Eclipse Juno and Java 1.7.

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

2.3 Get a simple demo running

- Go to "Window/Open Perspective/Other..."[5] and choose Moflon (Fig. 2.4).

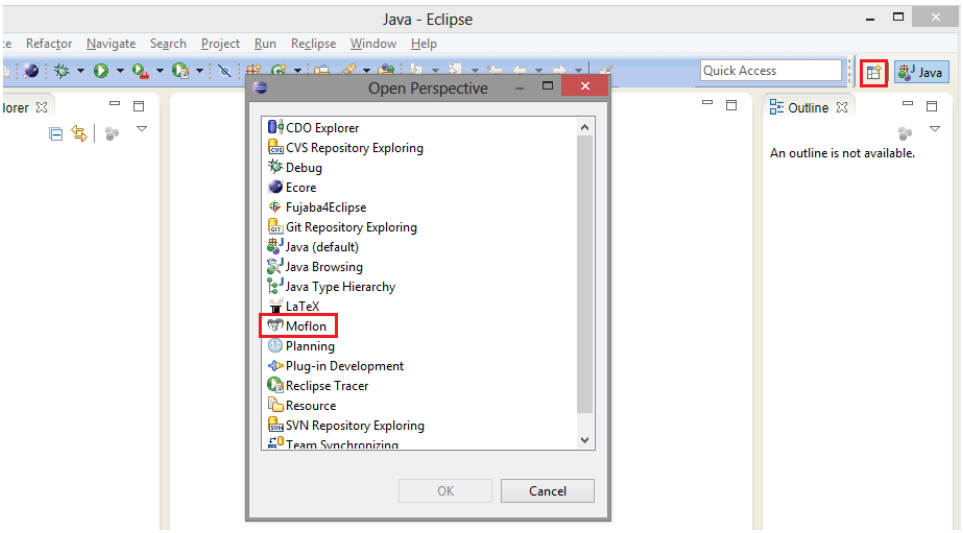


Figure 2.4: Choose the Moflon perspective.

- 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!).
- 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.

[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.

[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.

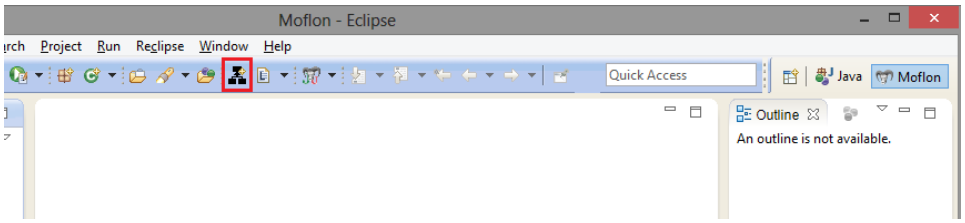


Figure 2.5: Eclipse "New Metamodel"

- 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.

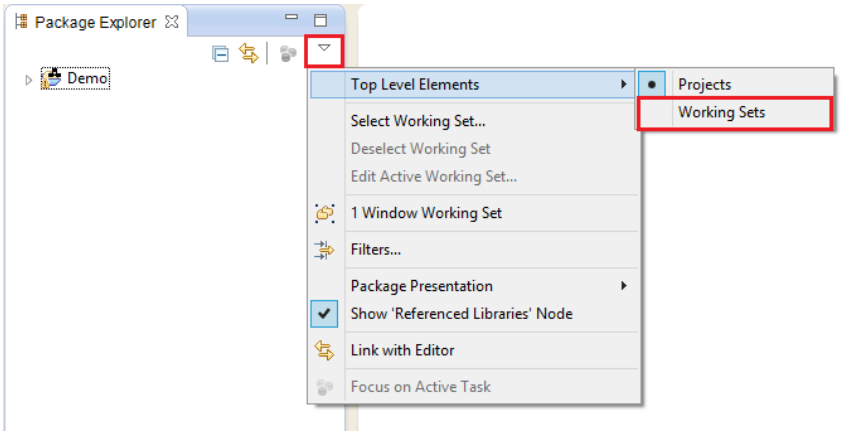


Figure 2.6: Choose Working Sets as Top Level Elements.

- 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.

- 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.

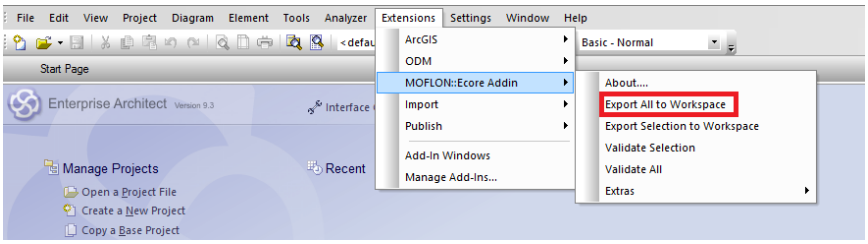


Figure 2.7: Export from EA using our extension

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

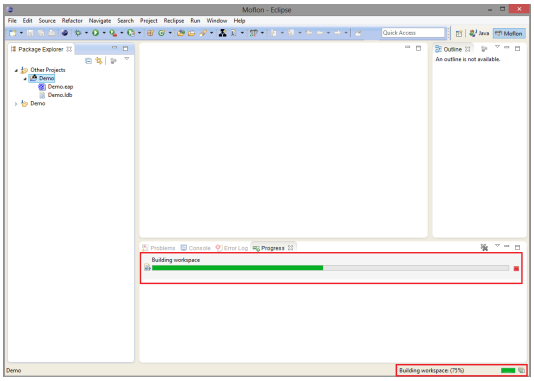


Figure 2.8: Automatically building the workspace after a refresh.

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.

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

- Right-click on the Testsuite project and select "Run as/JUnit Test". If anything goes wrong, try refreshing your whole workspace by choosing projects and pressing `F5` or right-clicking and selecting "Refresh".

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!

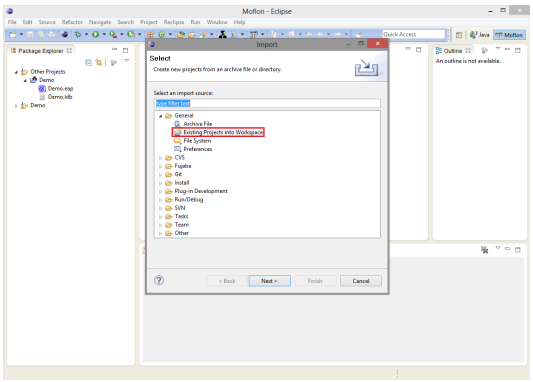


Figure 2.9: Import our Testsuite as an existing project.

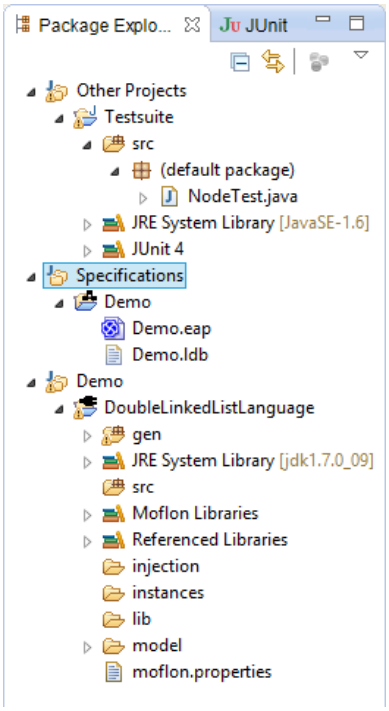


Figure 2.10: Workspace in Eclipse.

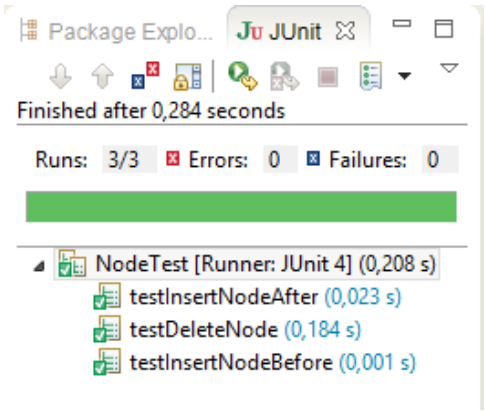


Figure 2.11: All's well that ends well...

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.

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.

[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.

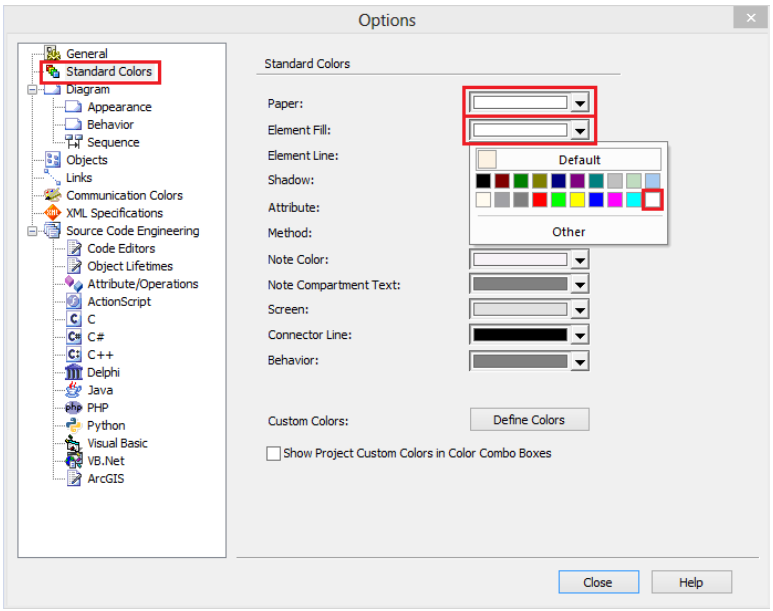


Figure 2.12: Our choice of standard colours for diagrams in EA.

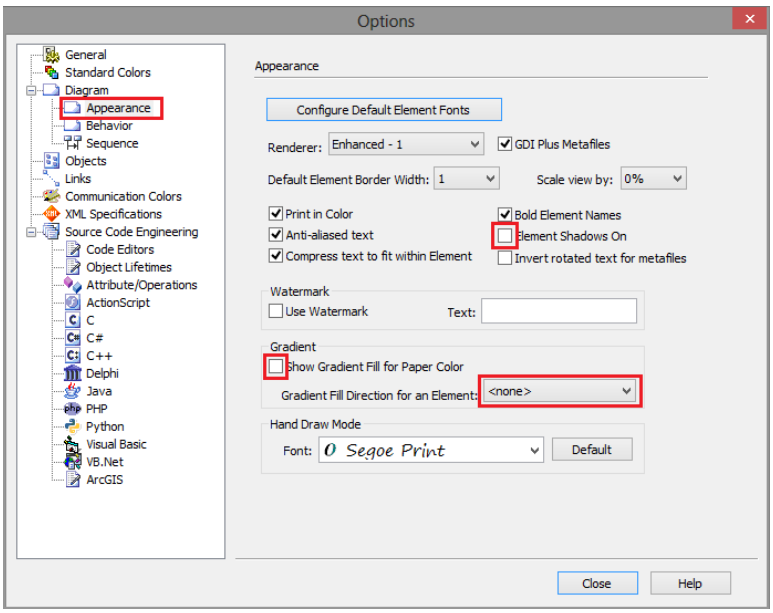


Figure 2.13: Our choice of the standard appearance for model elements.

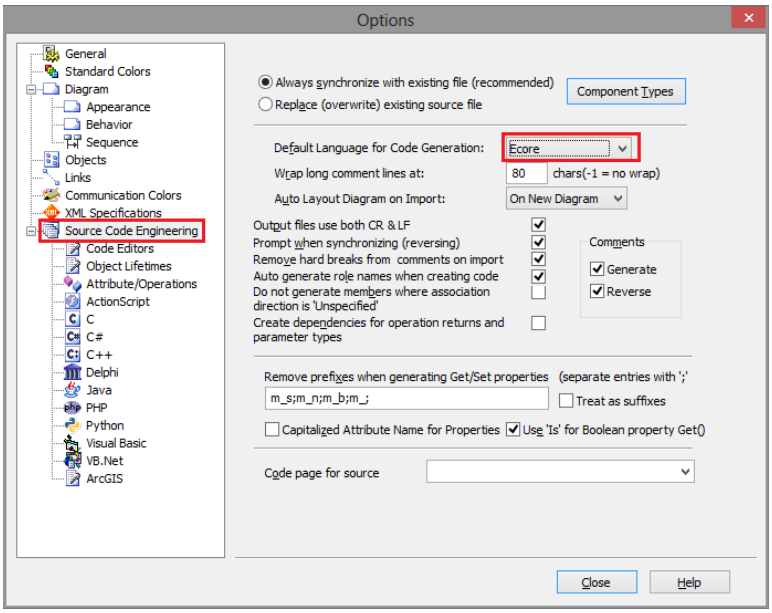


Figure 2.14: Make sure you set the standard language to Ecore.

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 depicts how the Eclipse working set `Demo` and its contents were generated from the EA model `Demo`. Every model in EA is mapped to a working set in Eclipse with the same name. From every package in the EA model, an Eclipse project is generated, also with the same name. These projects, however, are of a different *nature* than for example metamodel projects or normal Java projects, and are called *repository projects*. A nature is Eclipse lingo for "project type" and is visually indicated by a corresponding nature icon on the project folder. Our metamodel projects sport a 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".

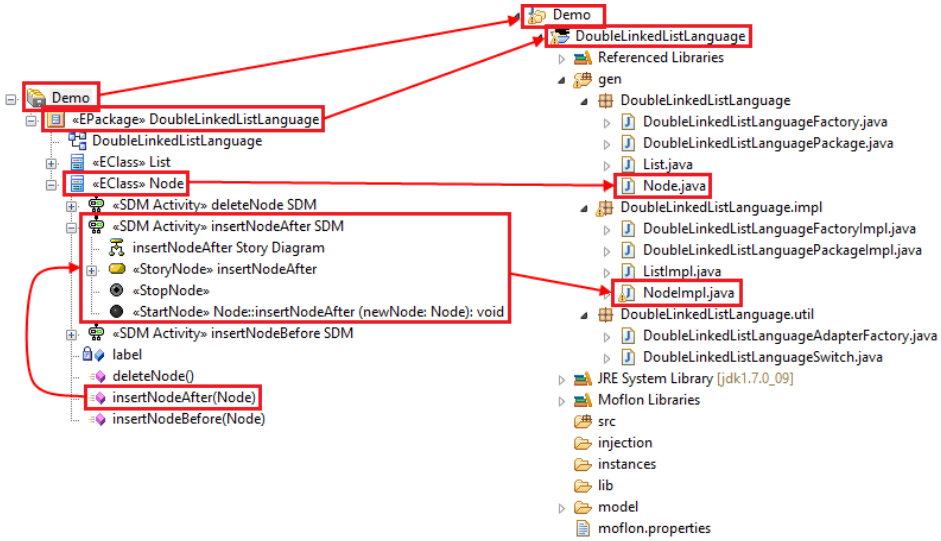


Figure 2.15: From EA to Eclipse

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 such as 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`, `injection` and `gen`, we advocate a clean separation of hand-written (this should go in `src` and `injection`) and generated code (lands automatically in `gen`). As we shall see later in the tutorial, hand-written code can be integrated in generated classes vio Injections. This is sometimes more elegant for small helper functions or necessary for String manipulation for instance.

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

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

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.

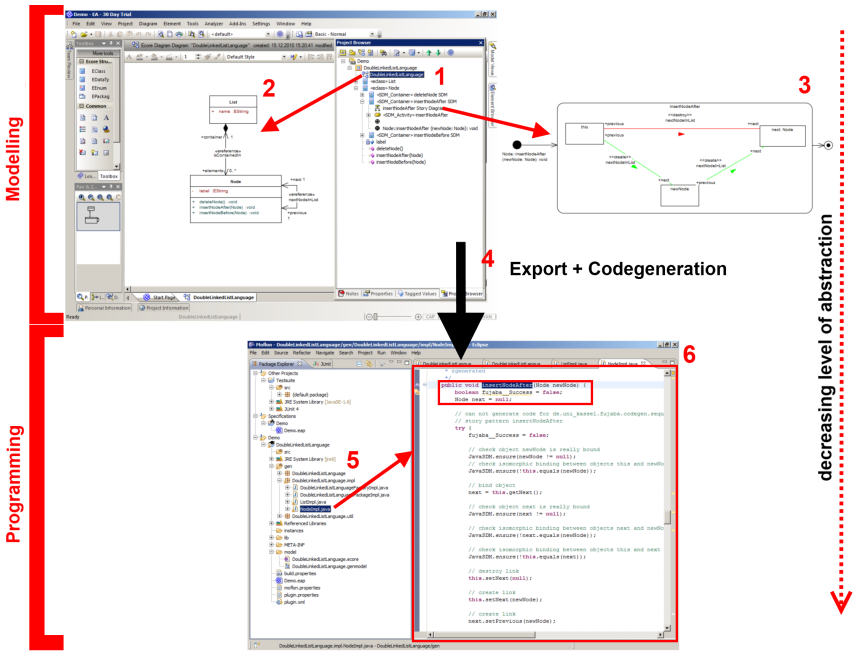


Figure 2.16: Overview

Chapter 3

Leitner's Learning 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 *Leitner's learning box*[1] is a simple but ingenious little contraption to support this tedious process of memorization.

As depicted in Fig. 3.1, it consists of a series of compartments or partitions usually of increasing size. The content to be memorized 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 memorized 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.

[1] http://en.wikipedia.org/wiki/Leitner_system

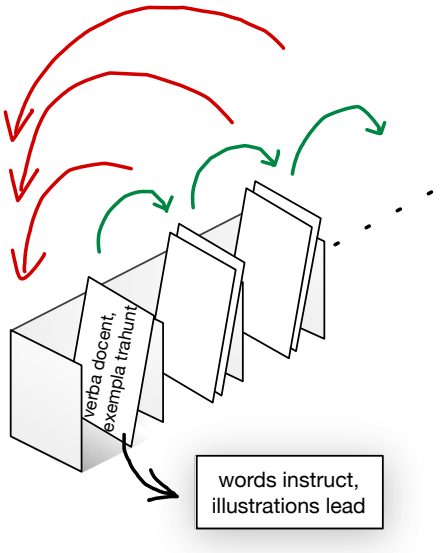


Figure 3.1: Possible *Concrete Syntax* of a Leitner's Learning Box.

A learning 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 learning box from scratch in a model-driven fashion and use it to introduce fundamental concepts in metamodelling and *Model-Driven Software Development* (MDSD) in general.

3.1 A language definition problem?

As 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 learning 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.

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 learning box, Fig. 3.1 can be viewed as a possible concrete syntax. As we are however building a learning box as a software system, our actual concrete syntax will probably be composed of GUI elements like buttons, drop-down menus and text fields.

Concrete Syntax

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.

Grammar

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 learning 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 learning 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.

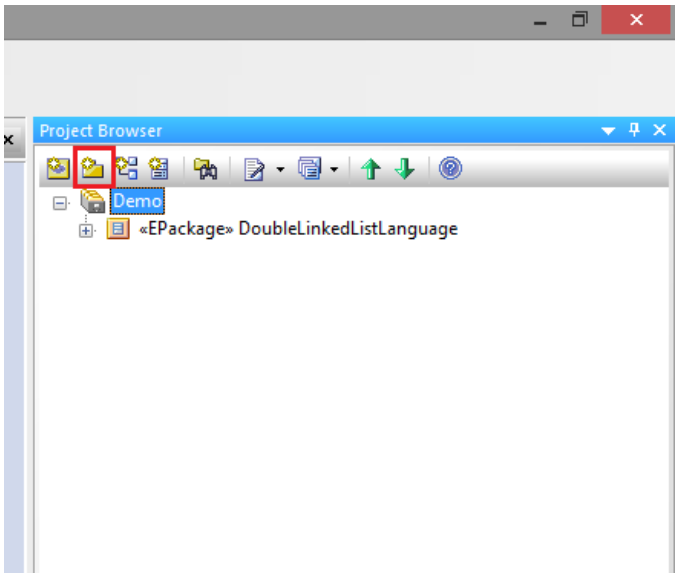


Figure 3.2: Add a new package to `Demo`.

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

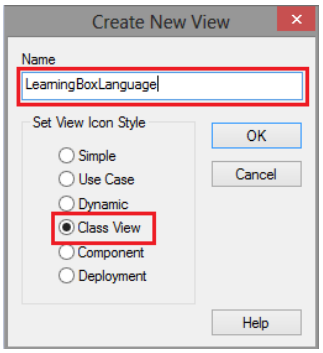


Figure 3.3: Enter the name of the new package.

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

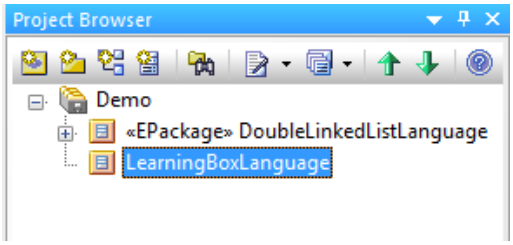


Figure 3.4: State after creating the new package.

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

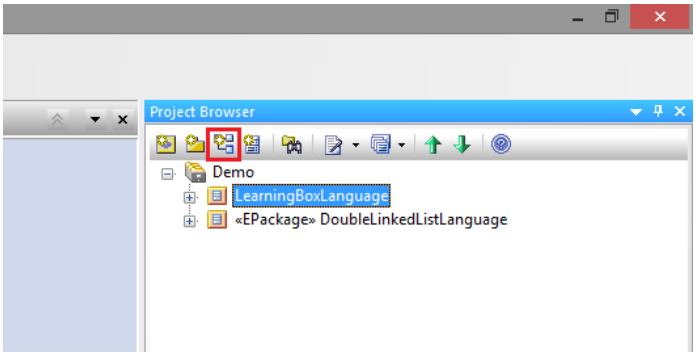


Figure 3.5: Add a diagram.

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

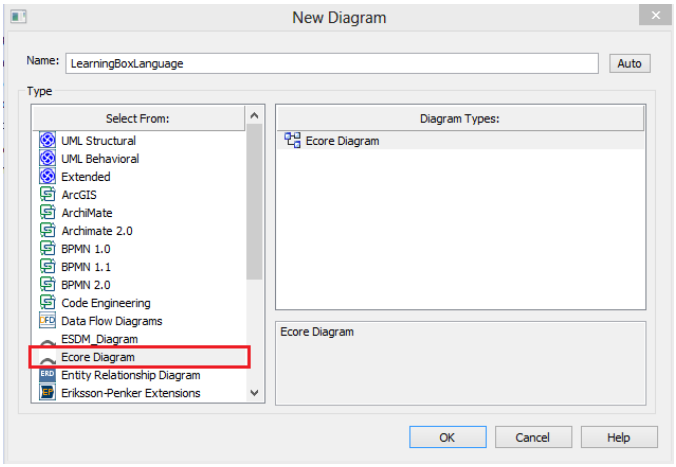


Figure 3.6: Choose type of diagram.

Unification

Meta-metamodel
Meta-Language
Modelling Language

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.

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.
- To the left of the workbench in EA, a *Toolbox* should have appeared[2] 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).
- 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.
- After creating `Box`, your EA workspace should resemble Fig. 3.10.

[2]If not, choose "View/Diagram Toolbox" to show the current toolbox.

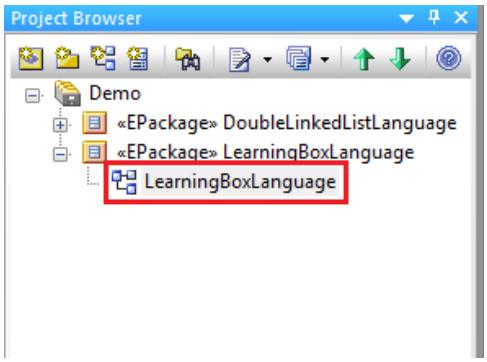


Figure 3.7: State after creating diagram.

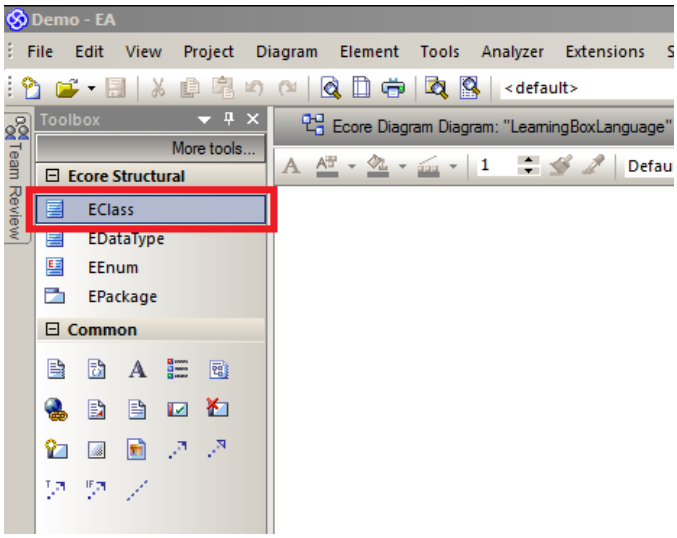


Figure 3.8: Create an EClass.

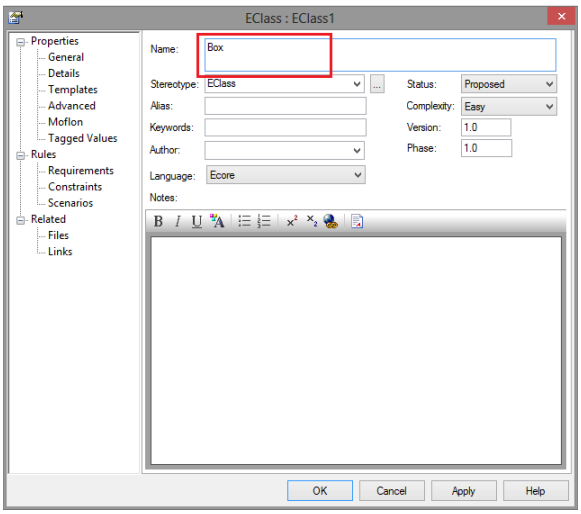


Figure 3.9: Enter properties of EClass.

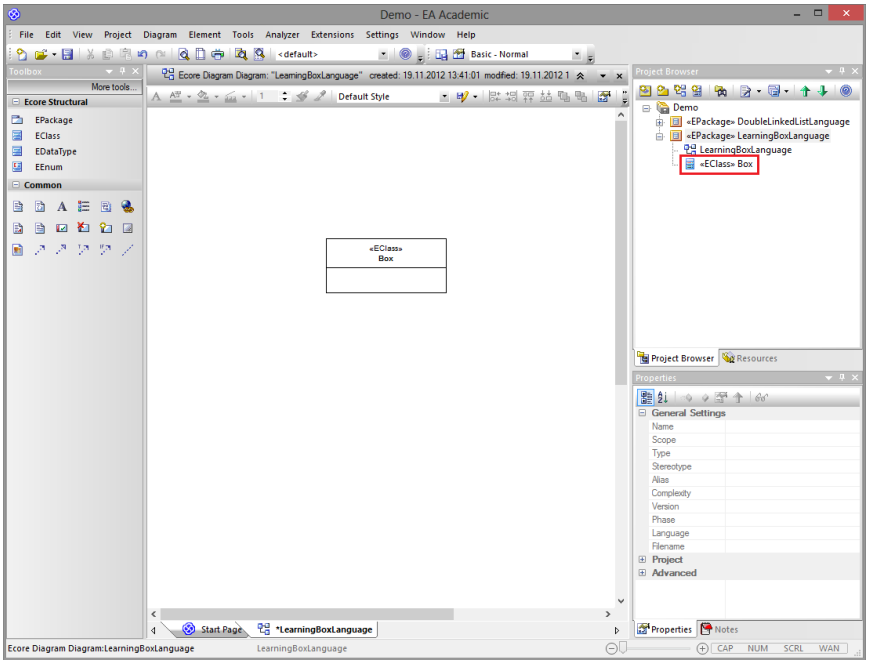


Figure 3.10: State after creating `Box`.

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

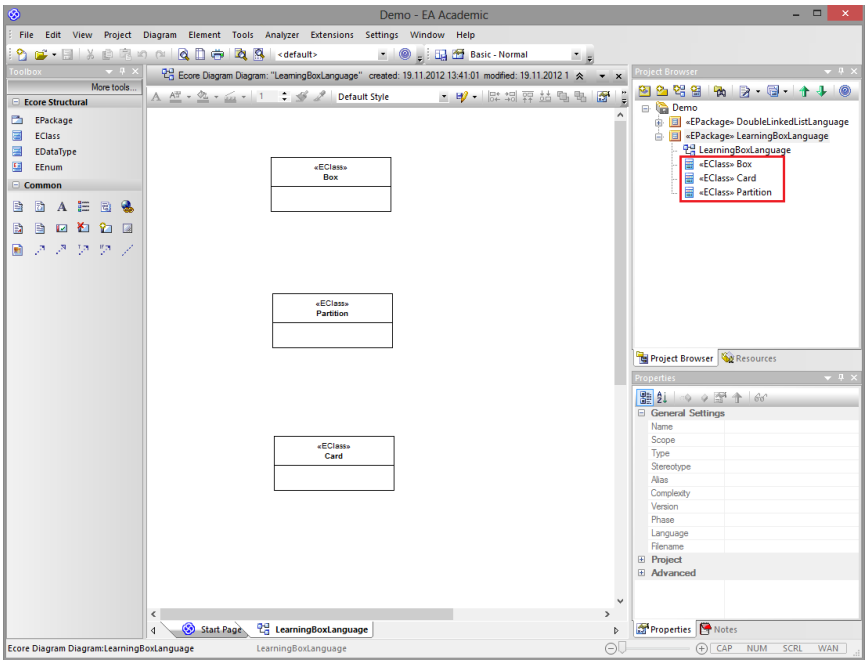


Figure 3.11: Main classes in our metamodel.

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

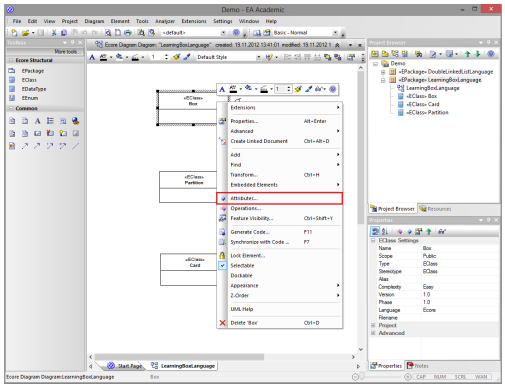


Figure 3.12: Context Menu for a class.

▶ 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`.

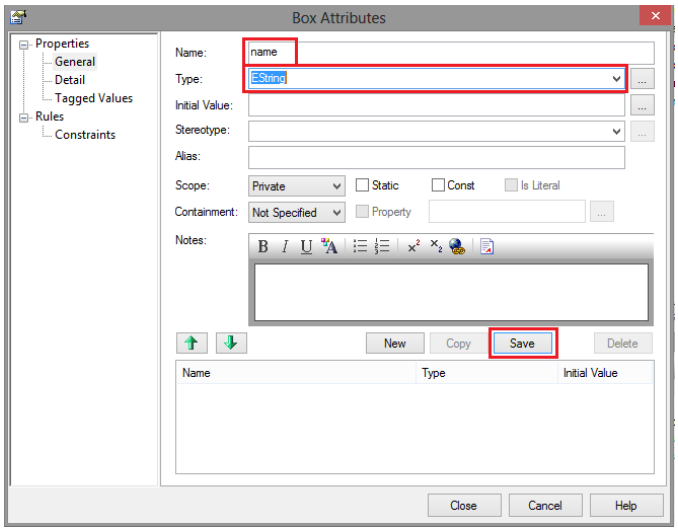


Figure 3.13: Adding attributes to a class.

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

Figure 3.14: Main classes with attributes.

- 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. 3.15).

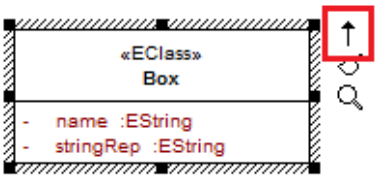


Figure 3.15: Quick Link is a central gesture in EA.

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. 3.16).

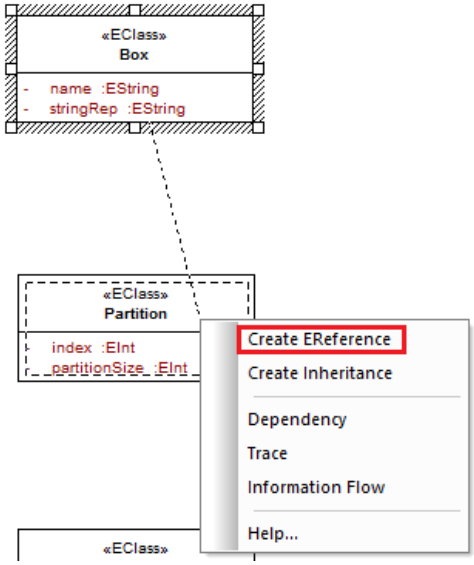


Figure 3.16: Create a reference via Quick Link.

- Double click the reference to invoke a dialogue (Fig. 3.17), 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.
- In the same dialogue choose `Source Role` and enter the values in Fig. 3.18 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`.

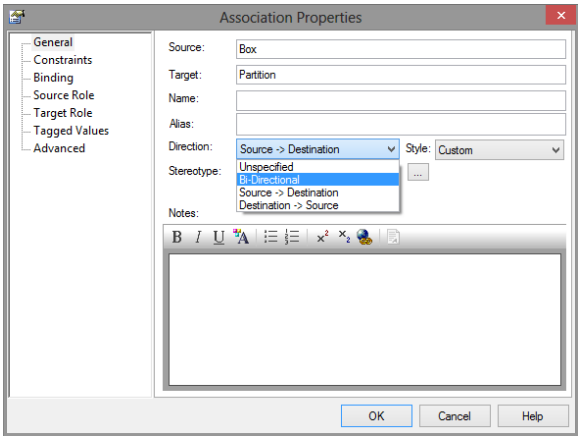


Figure 3.17: Enter properties of the reference.

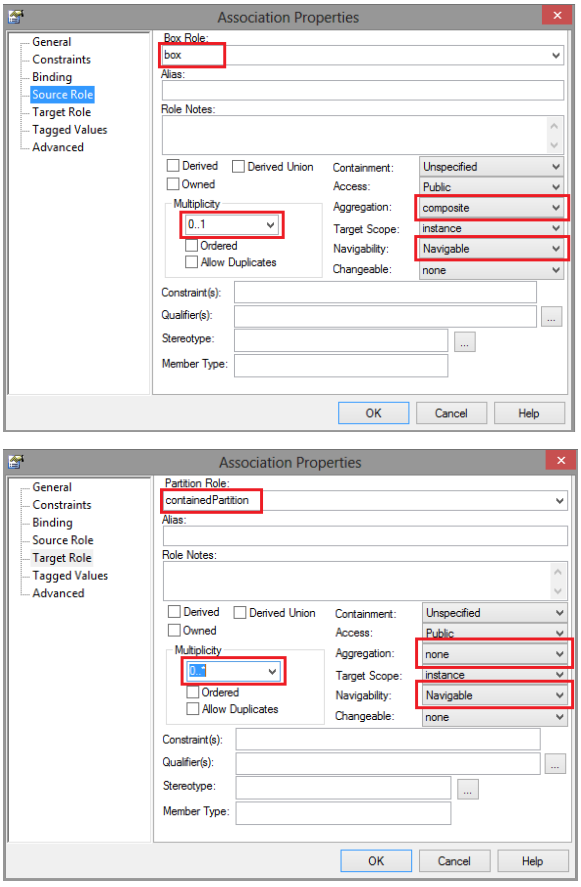


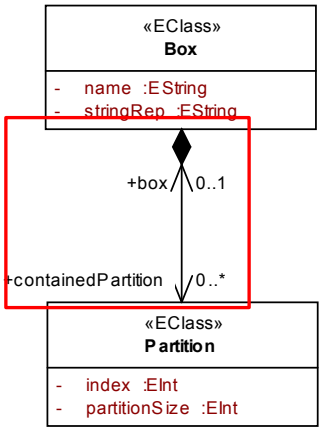
Figure 3.18: Enter properties for source and target of reference.

Navigable ends are mapped to class attributes with getters and setters in Java and therefore *must* have a specified name and multiplicity for successful code generation. 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. 3.19 with a relation between `Box` and `Partition`.



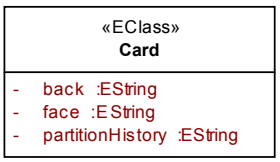


Figure 3.19: `Box` contains `Partitions`.

- Create a bidirectional reference[3] between `Partition` and `Card` and two unidirectional self-references for `Partition` according to Fig. 3.20[4].

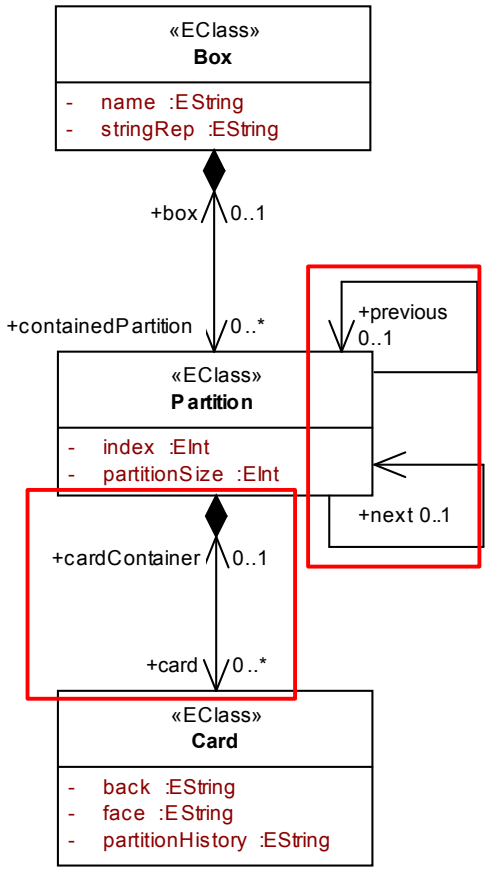


Figure 3.20: All relations in our metamodel.

Dynamic Semantics

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.

[3] 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.

[4] If you have difficulties deciphering the role names and other details in the screen shot please refer to Fig. 3.25 for a better diagram of the metamodel.

In the next few steps we shall define the *signatures* of some operations for our learning box. We will of course use SDMs to *implement* the methods later.

- Right-click `Partition` to invoke the context-menu depicted in Fig. 3.21 and choose `Operations...`.

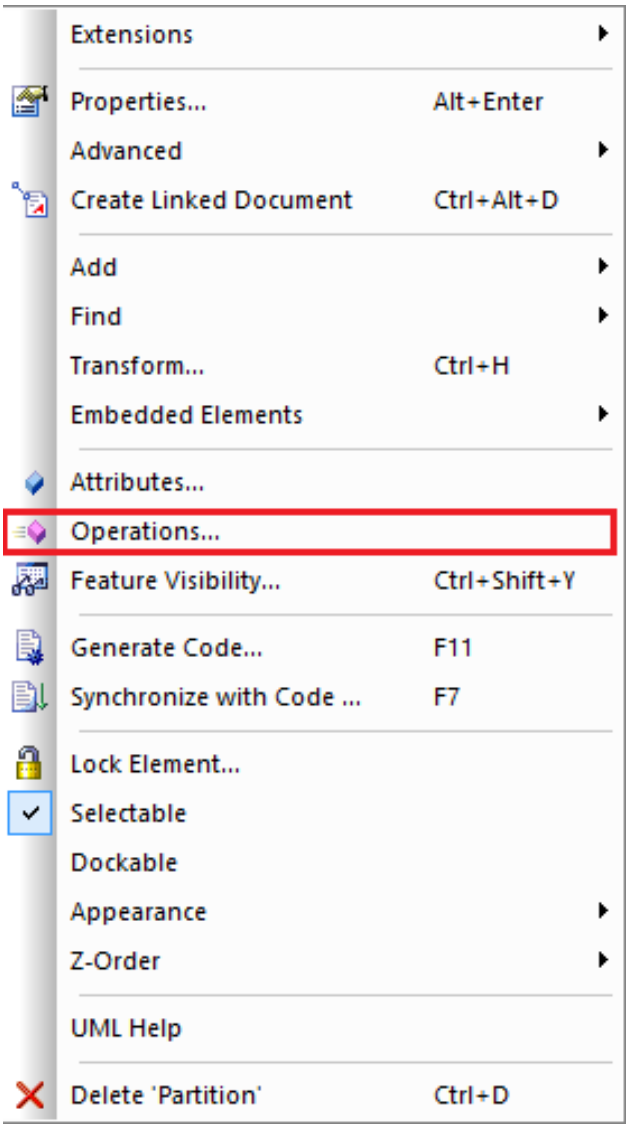


Figure 3.21: Add an operation.

- In the dialogue that pops-up (Fig. 3.22), enter `empty` as the `Name` of the operation, leave the `Return Type` as `void`. Press `Save`.

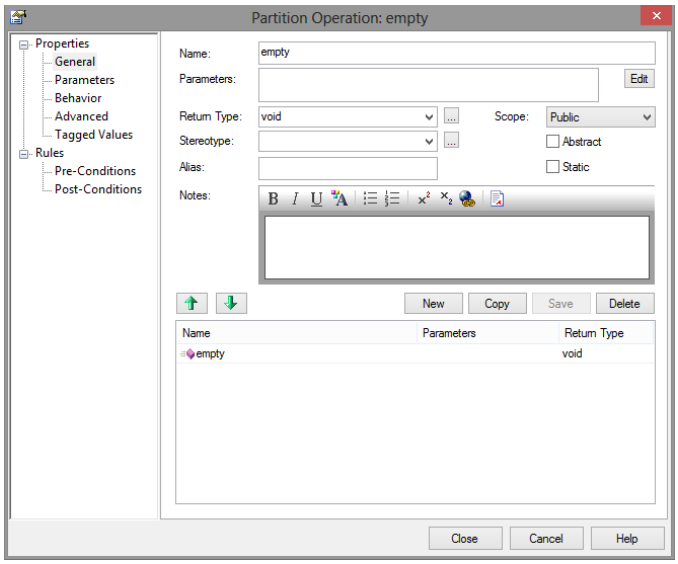


Figure 3.22: Properties for operation.

- In the same dialogue, press `New` to add further operations and enter the values in Fig. 3.23. Parameters can be added by pressing `Edit` and entering the name and choosing the type of each parameter in a separate dialogue.

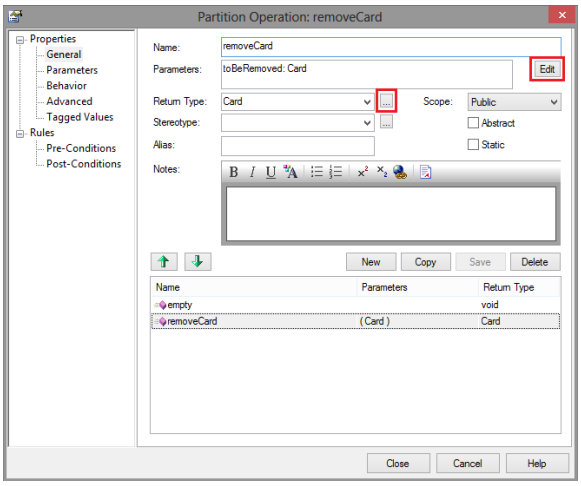


Figure 3.23: Parameters and Return Type.

- Repeat the process for the values in Fig. 3.24. The `Return Type` can be chosen via the drop-down menu for primitives (e.g. `EBoolean`), or via the `...` button (indicated in Fig. 3.23) 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!

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. 3.24.

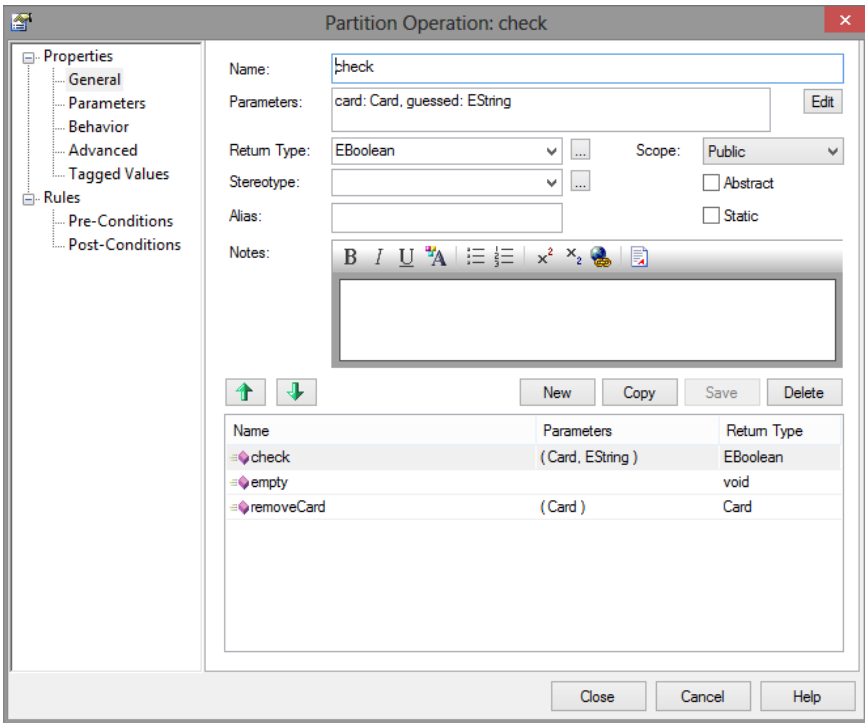


Figure 3.24: All operations in `Partition`.

Add all operations analogously for `Box` and `Card`, so that your metamodel closely resembles Fig. 3.25.

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

A `Box` has a `name`, and can be extended by calling `grow`. A `Box` can print out its contents via `toString`.

The main method of the learning 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.

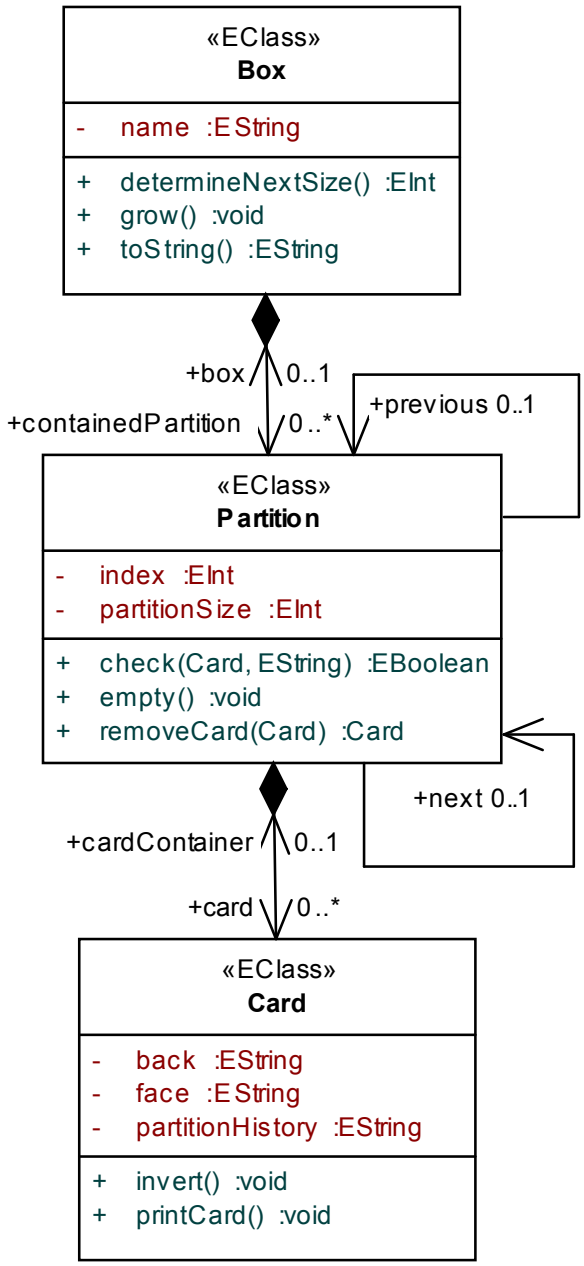


Figure 3.25: Complete metamodel for our learning box (note that names of method parameters are not displayed by default in EA but are shown here for presentation purposes).

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 `partitionHistory` which can be used to keep track of how often a `Card` has been answered correctly/wrongly. This might indicate how difficult the `Card` is for a specific user. When learning a language, it makes sense to be able to swap the target and source language and this is supported by `Card` via `invert` (turns the card around).

Now try to export the metamodel for code generation in Eclipse (if problems occur during the following steps, we recommend to read section 3.3. This might help you to find your mistakes).

- To do this right-click on `LearningBoxLanguage` and choose "Extensions/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 `LearningBoxLanguage` 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. 3.25. 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 code generator actually supports injecting hand-written implementations of methods into generated methods and classes. 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 Validating your metamodel

Our EA extension provides rudimentary support for validating the static semantics (Ecore) and dynamic semantics (SDM) of metamodels. Validation results are displayed and, in some cases, even "quick fixes" to automatically solve the problems are offered. Our validation framework is still work in progress so if you find an error that is not/wrongly detected feel free to send us a brief description.

- To make the validation output window visible in EA, choose “Extensions/-Add-In Windows”. This should display a new output window as depicted in Fig. 3.26. This control panel also contains shortcuts for all the functionality available via the extensions file menu and many users actually prefer this.

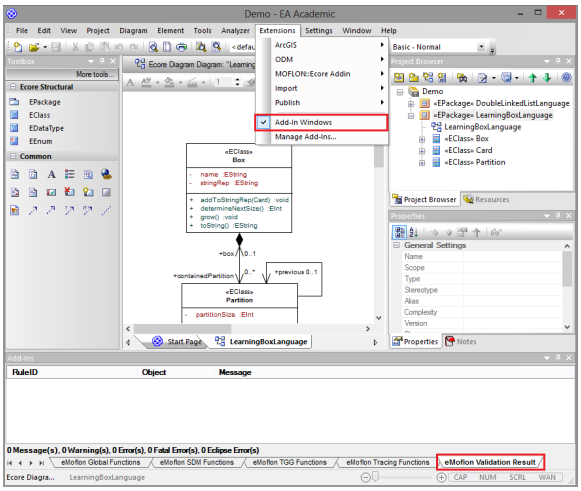


Figure 3.26: Activating the validation output window

- To start the validation choose “Extensions/MOFLON::Ecore Addin/-Validate all” in EA (Fig. 3.27). If you haven’t already made any mistakes while modelling your `LearningBoxLanguage` in the last chapter, the output should resemble Fig. 3.26, indicating that your metamodels are free of errors (at least according to our validation).

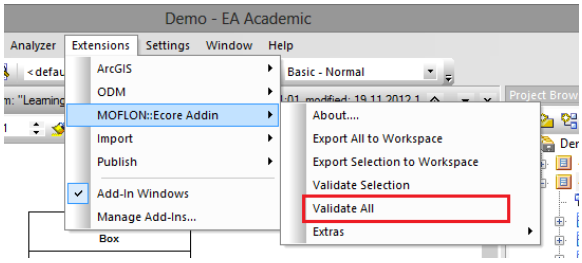


Figure 3.27: Starting the validation

To get familiar with our validation and quick fix features, let’s add two small modelling errors in `LearningBoxLanguage`.

- Create a new class in the `LearningBoxLanguage` diagram. You can retain the default name `EClass1`. Let’s assume, you wish to delete this class from your metamodel.

- Select it (`EClass1`) in the diagram and press the `Delete` button. Note that `EClass1` still exists in the project browser (and thus in your metamodel) and that you have one new `Information` message in the validation output (Fig. 3.28).

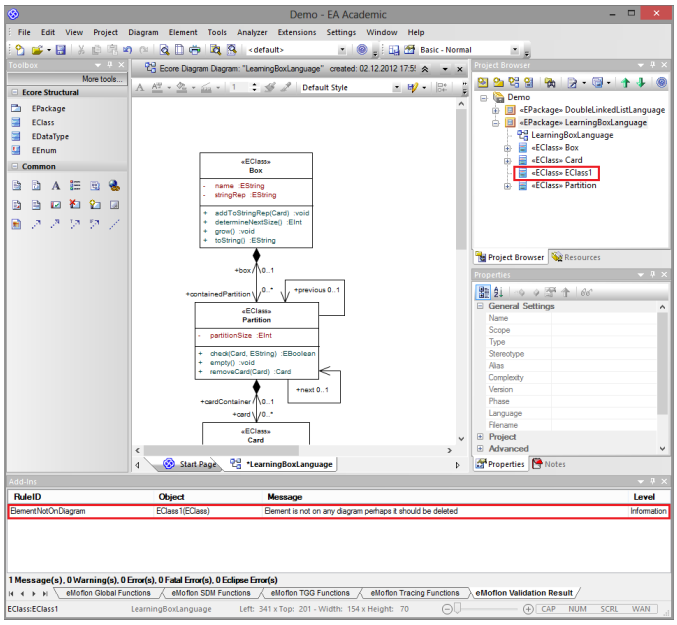


Figure 3.28: Output indicating that an element is present in the metamodel, which is not in any diagram and should probably be deleted completely

The message informs you that `EClass1` is not on any diagram, and as it is still in the model, that this could be a mistake. Just pressing the `Delete` button is apparently not the proper way of *deleting* `EClass1` from the metamodel and only removes it from the current diagram. Deleting elements properly and other EA specific aspects are discussed in detail in Chapter 4.

- To navigate to the problematic element click once on the information message in the output window. EA should navigate automatically to `EClass1` and highlight it in the project browser.

- To check if there are any quick fixes available, double click the information message to invoke the quick fix dialogue. In this case, there is one quick fix which suggests simply deleting the element from the model (Fig. 3.29) as this is probably what was intended.

- Click `OK` and `EClass1` will be deleted correctly from your model. Your metamodel should now be error-free again as indicated by the validation output window.

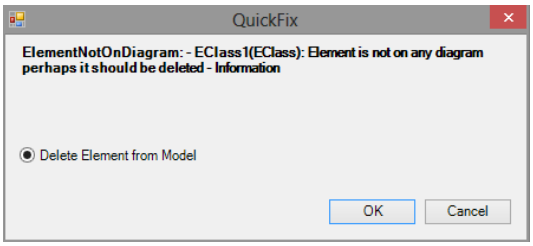


Figure 3.29: Quick fix for elements that are not on any diagram

- To make an error that leads to a more critical message than "information", double click the navigable reference end `previous` of the class `Partition`, and delete its role name as depicted in Fig. 3.30. Affirm with `OK`.

  You should now see a new `Fatal Error` in the validation output, stating that a navigable end *must* have a role name (Fig. 3.31). As navigable references are mapped to data members in a Java class, omitting the name of a navigable reference makes code generation impossible (data members, i.e., class variables must have a name).

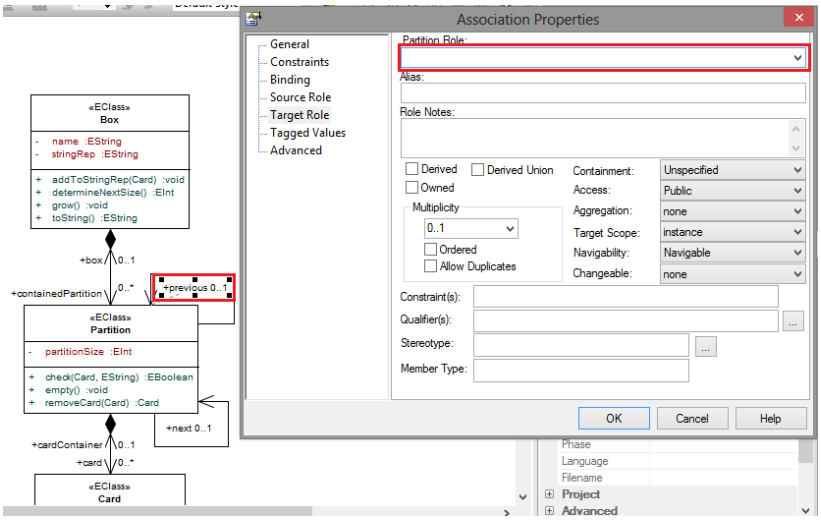


Figure 3.30: Deleting a navigable role name of a reference

- As before, clicking the error message selects the problematic connector in the diagram, and double clicking reveals that there are no quick fixes for this problem (Fig. 3.31).

- Correct your metamodel manually by setting the name of the navigable reference back to `previous`.

- Ensure that your metamodel closely resembles Fig. 3.25 again, and that there are no error messages before you proceed with the rest of the chapter.

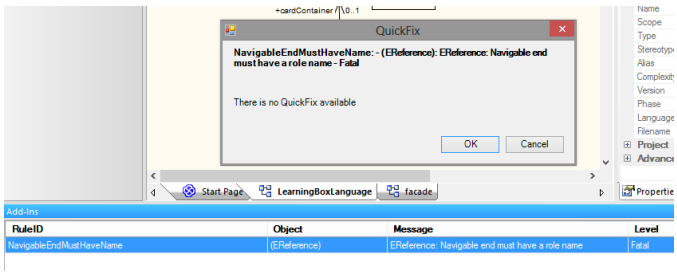


Figure 3.31: Fatal Error after deleting a navigable role name

As you've probably already noticed, we distinguish between five different types of validation messages:

Information:
This is only a hint for the user and can be safely ignored if you know what you're doing. Export and code generation should be possible, but certain naming/modelling conventions are violated, or a problematic situation has been detected.

Warning:
Export and code generation is possible, but only with defaults and automatic corrections applied by the code generator. As this might not be what the user wants, such cases are flagged as warnings (e.g., omitting the multiplicity at references which is automatically set by the code generator to 1). Being as explicit as possible is often better than relying on defaults.

Error:
Although the metamodel can be exported from EA, it is not Ecore conform and code generation will not be possible.

Fatal Error:
The metamodel cannot be exported as required information such as names or classifiers of model elements are incorrectly set or missing.

Eclipse Error:
Error messages produced by our Eclipse plugin after an unsuccessful attempt to generate code. This is currently not actively used.

3.4 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 Sect. 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 `LearningBoxLanguage` project. Double-click the `LearningBoxLanguage.ecore` model to invoke the *Ecore model editor*. Expand this tree to view the different classes and packages you modelled with EA in Sect. 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. 3.32.

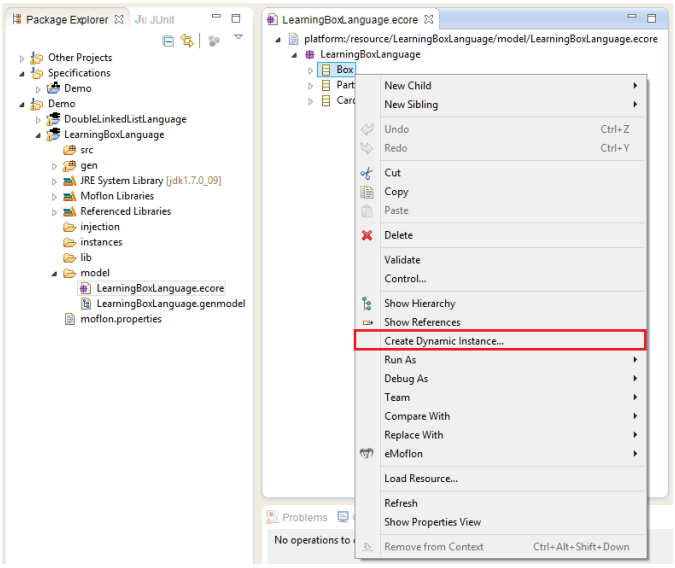


Figure 3.32: Context menu of Ecore model in Eclipse

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. 3.33).

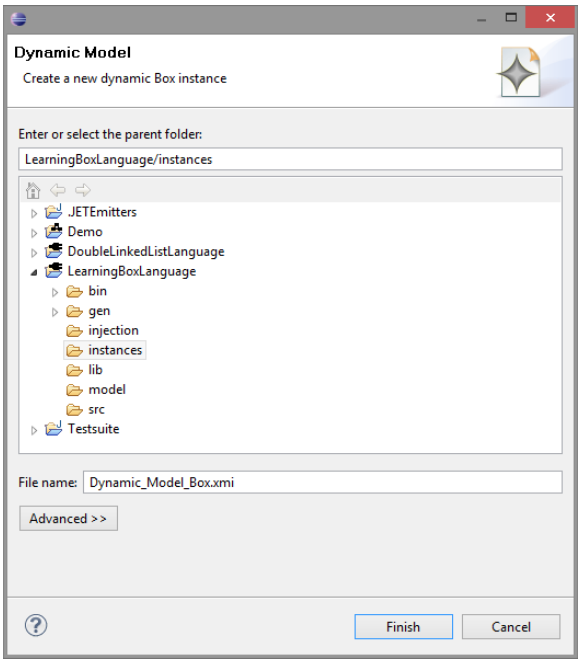


Figure 3.33: Dialogue for creating a dynamic model instance

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. 3.34. 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.

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 learning box!

3.5 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 $\Delta : S \xrightarrow{\Delta} S^*$. In this light, dynamic or behavioural aspects of a model

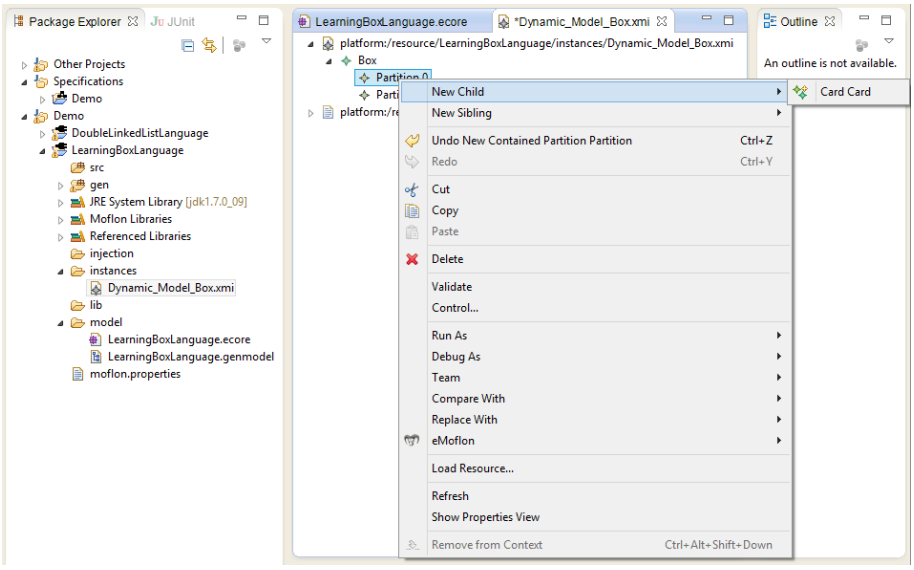


Figure 3.34: Context menu for creating model elements

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. 3.35.

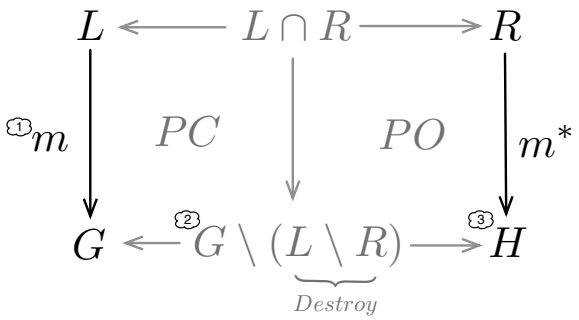


Figure 3.35: Applying a rule $r : (L, R)$ to G to yield H

(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 learning box. How would a rule look like for moving a card from one partition to the next? Fig. 3.36 depicts the rule *moveCard*.

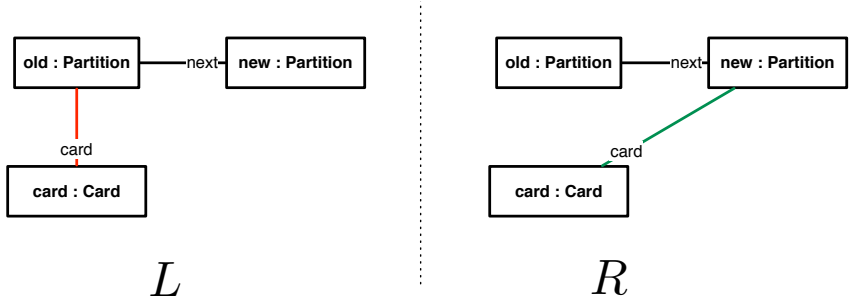


Figure 3.36: Rule *moveCard* as a graph transformation rule.

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. 3.37).

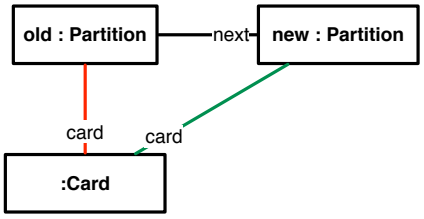


Figure 3.37: Compact representation of *moveCard* as a Story Pattern.

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 learning box according to steps (1) – (3) is depicted in Fig. 3.38.

One last thing before we continue with our learning 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...

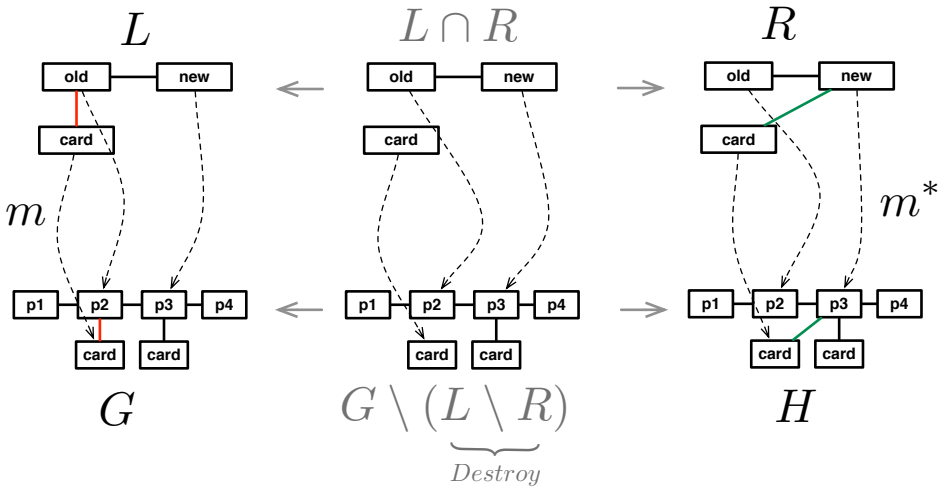


Figure 3.38: Applying *moveCard* to a learning box.

3.5.1 Removing cards from a partition

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. 3.39). (3) *Double-click* on the chosen method to indicate that you want to implement it. *Create an SDM*

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. 3.40. 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[5], 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*

Now it's time to complete our very first activity. Choose the start node, and note the small black arrow that appears (Fig. 3.41).

[5] 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.

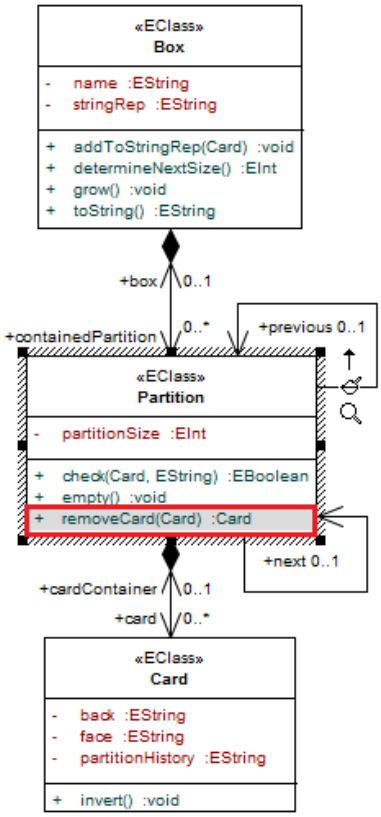


Figure 3.39: Double-click a method to implement it.

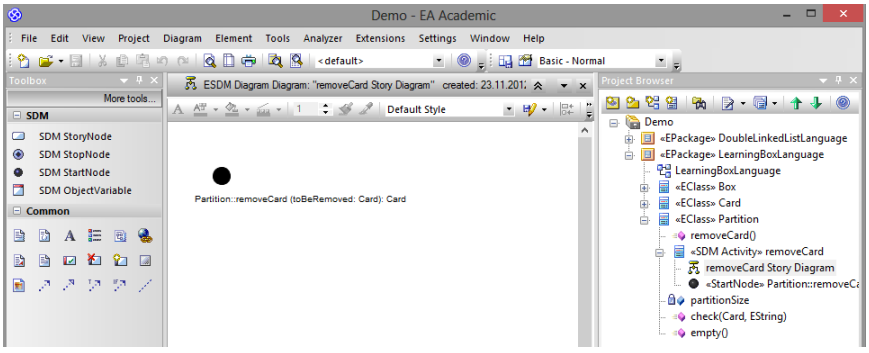


Figure 3.40: Generated SDM diagram and start node.

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.

Quick Create

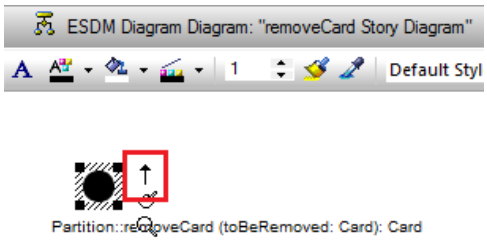


Figure 3.41: Quick link in SDM diagram to create new activity node.

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.

As indicated in Fig. 3.42 choose `Append StoryNode` 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.

Activity
Activity Node
Activity Edge

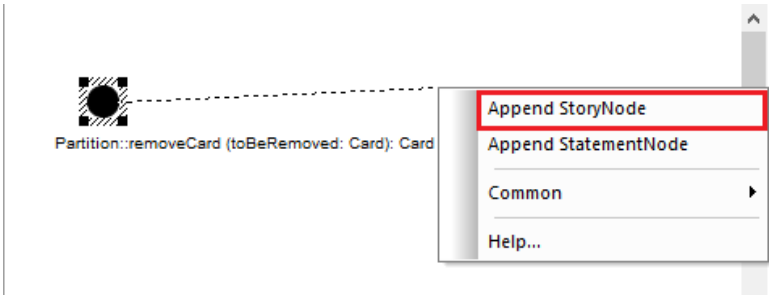


Figure 3.42: Create new activity node.

Complete the activity by quick creating a stop node as depicted in Fig. 3.43.

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

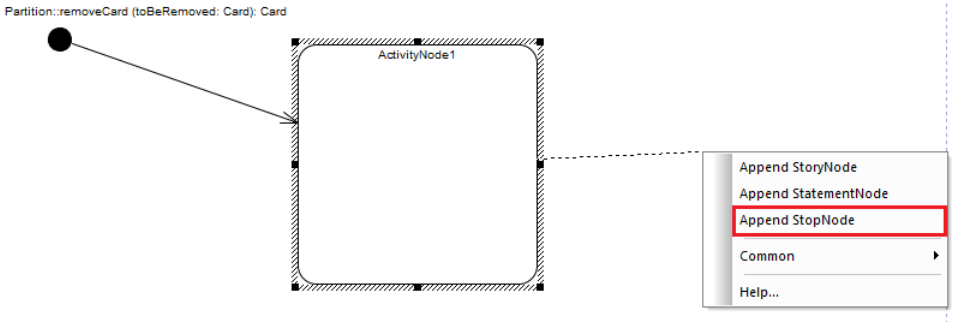


Figure 3.43: Complete activity with a stop node.

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. 3.44 now with the activity node connected via an activity edge to the newly created stop node.

Story Pattern

Story 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. 3.5. As not all activity nodes can contain story patterns (e.g. start and stop nodes), those that can are called *story nodes*.

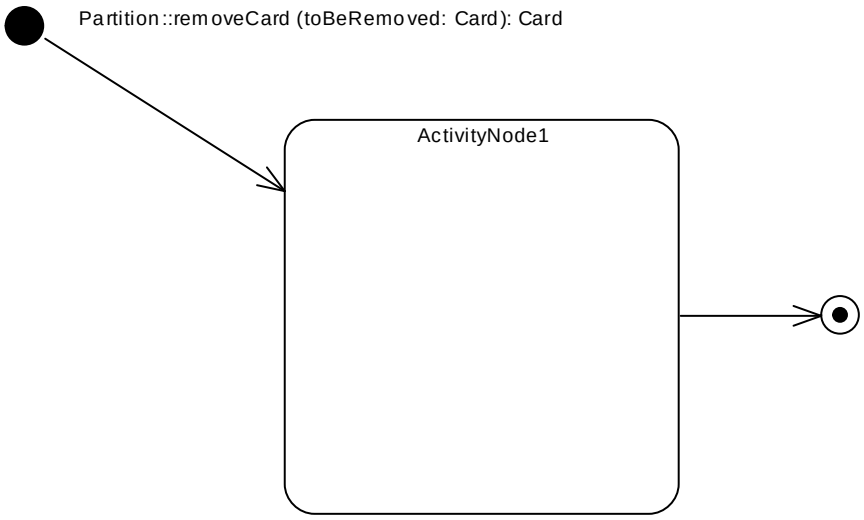


Figure 3.44: Control flow modelled as a simple activity diagram.

To create a story pattern, double click the story node `ActivityNode 1` in Fig. 3.44 to prompt the dialogue depicted in Fig. 3.45. Enter `removeCardFromPartition` as the name of the story node, check `Create this Object` and click `OK`.

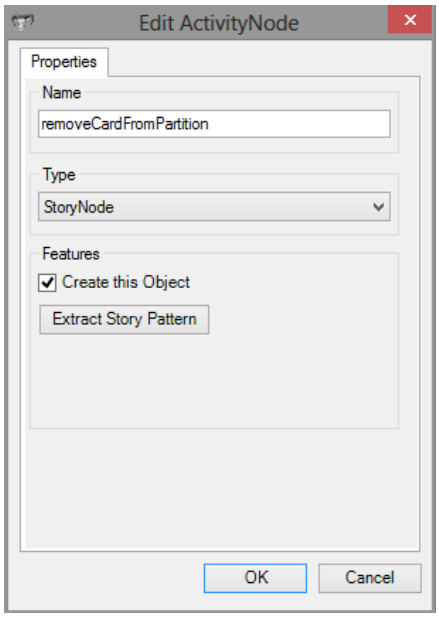


Figure 3.45: Start modelling story pattern in activity node.

Object Variable

Pattern Matching

The activity node should now have a single *object variable* `this` (Fig. 3.46). 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[6]. In our 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).

To create an object variable that can be assigned to other objects, choose `SDM ObjectVariable` from the toolbox as indicated in Fig. 3.46 and click *in the* activity node `removeCardFromPartition` (Fig. 3.47).

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.

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

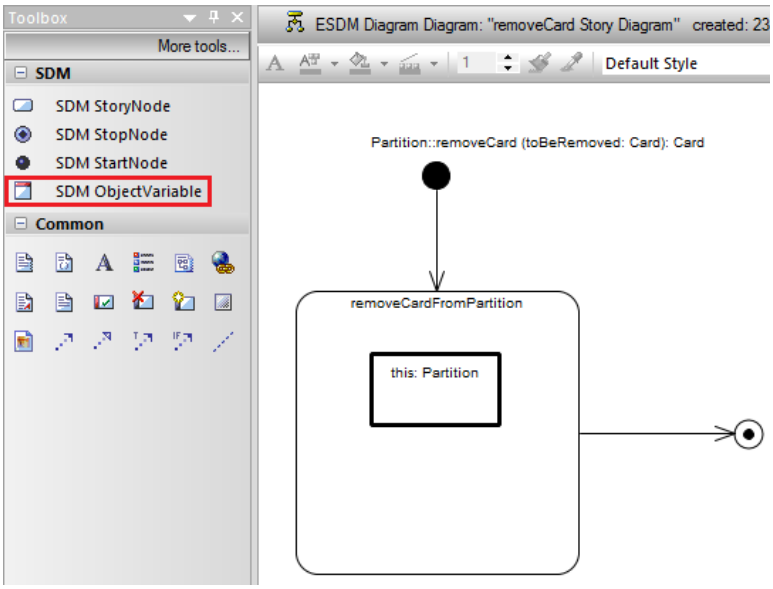


Figure 3.46: Add a new object variable from the tool-box.

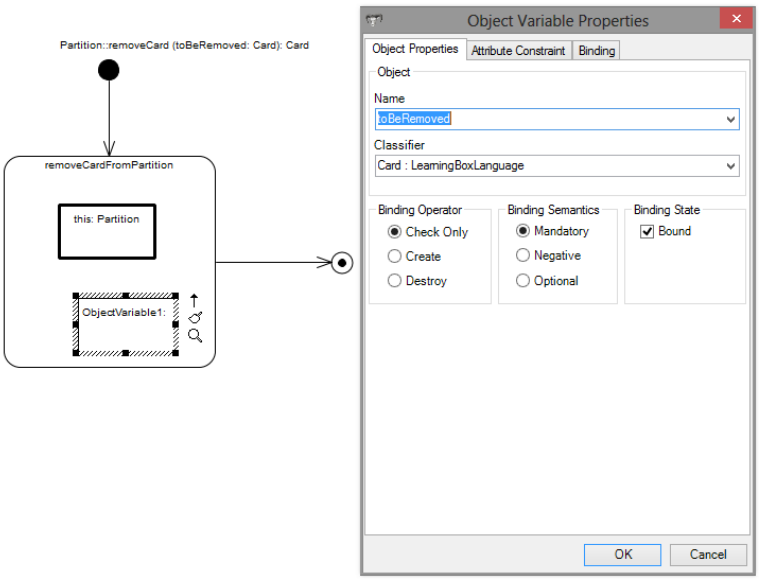


Figure 3.47: Specify properties of the added object variable.

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.

Binding State

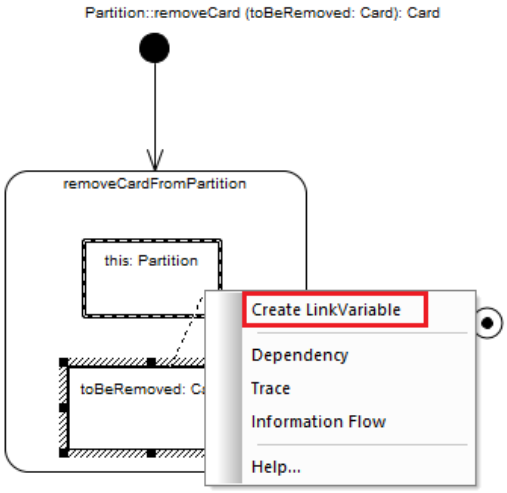


Figure 3.48: Create a link variable.

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. 3.48).

Link Variable

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. 3.49). Every object or link variable's binding operator can be set to one of `Check Only, Create, Destroy`.

Binding Operator

For a rule $r : (L, R)$, as discussed in Sec. 3.5, 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 ($L \setminus R$).

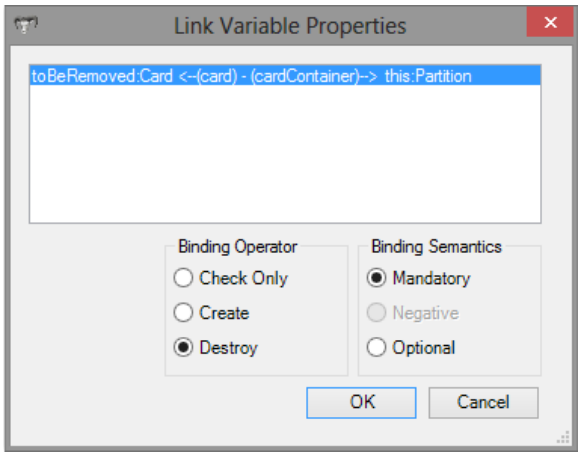


Figure 3.49: Specify properties for created link variable.

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. 3.50, 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[7]. 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.

Parameter Expression

The alternative approach would be to support arbitrary expressions, for example, in a script language like JavaScript or in an appropriate DSL[8] 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*

[7] We also do not support nesting expressions

[8] 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.

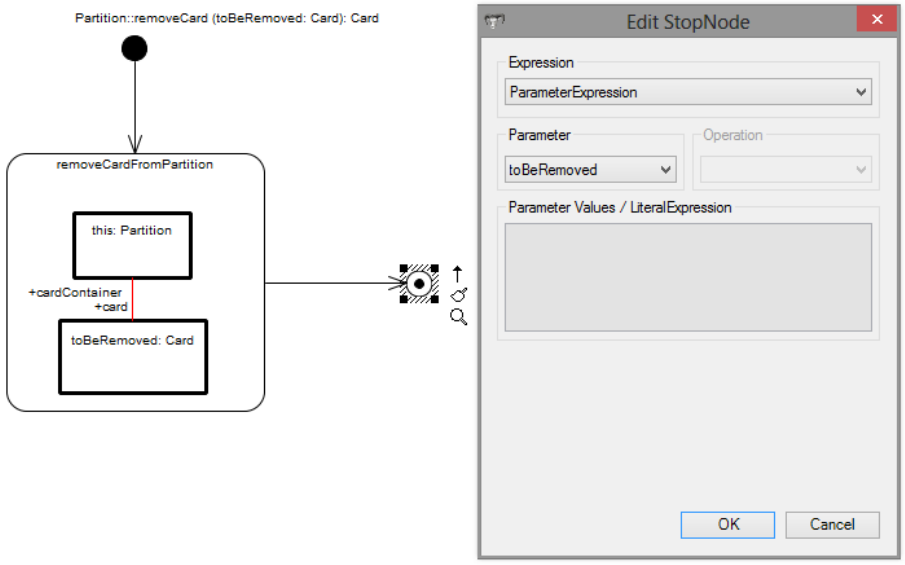


Figure 3.50: Adding a return value to the stop node.

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. 3.51 with the return value indicated below the stop node.

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[9] and see if you can get a feel for what 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!

[9]The generated method is in `/LearningBoxLanguage/gen/LearningBoxLanguage/-impl/PartitionImpl.java/removeCard`

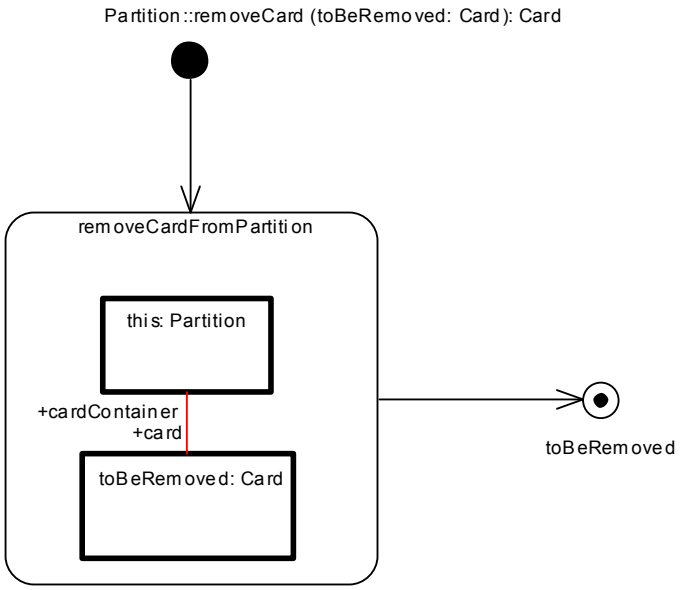


Figure 3.51: Complete SDM for `Partition::removeCard`.

If you're unable to export or generate code successfully, compare your SDM carefully with Fig. 3.51 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.

3.5.2 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 learning 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 *penalized* 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. 3.52.

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 learning box. Remember that this binding is implicitly specified by choosing the name of the argument as the name of the object variable (Fig. 3.53).

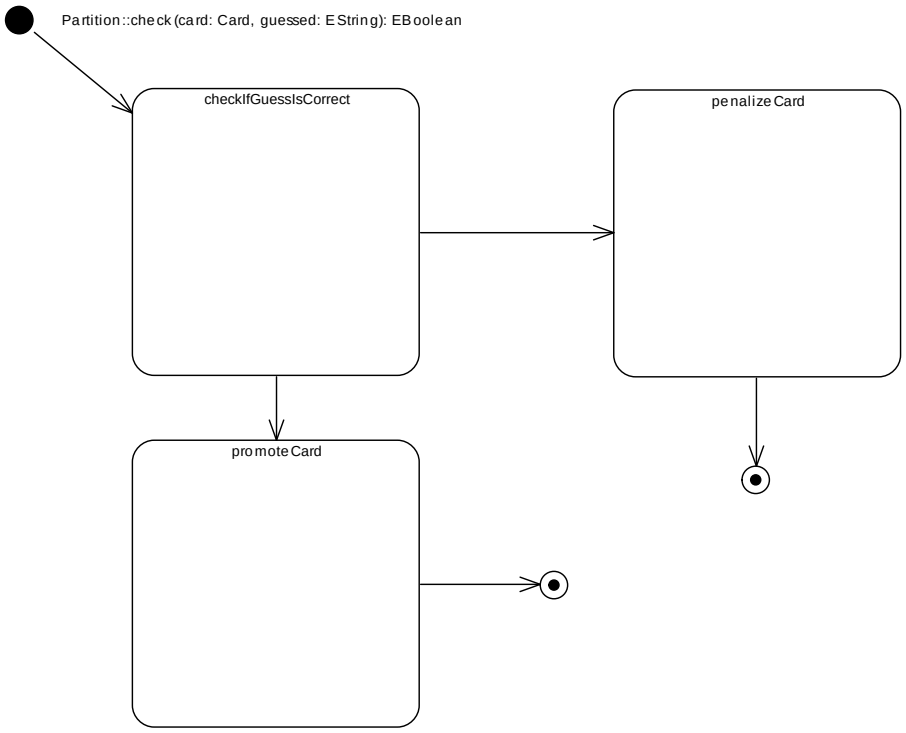


Figure 3.52: Activity diagram for `Partition::check`.

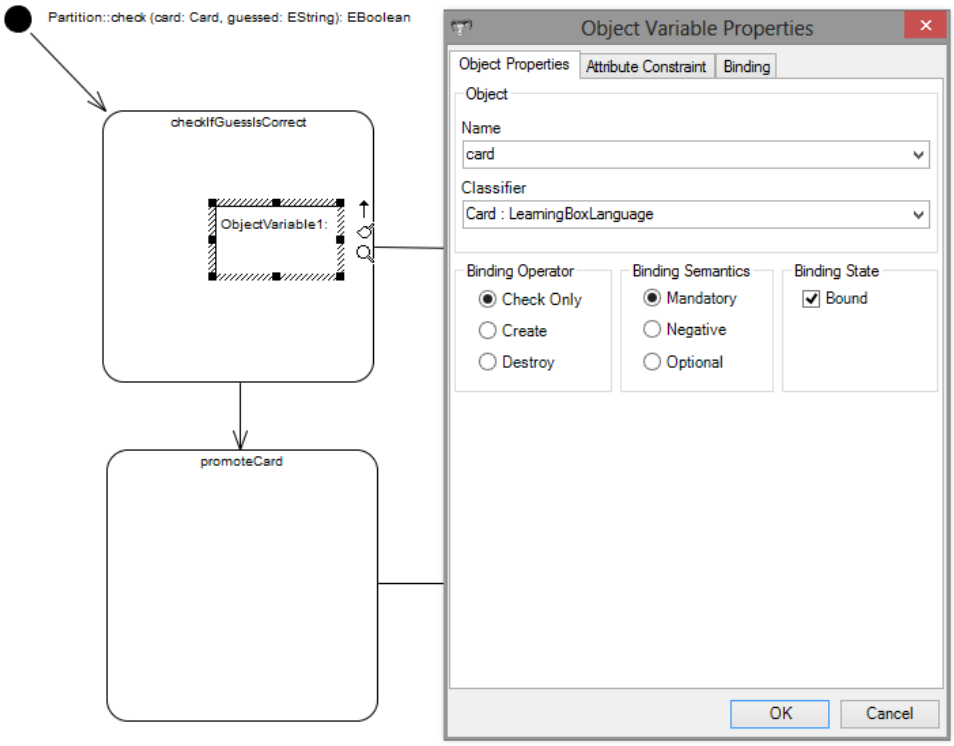


Figure 3.53: Add the card to be checked.

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. 3.54. 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. 3.54)!

Attribute Constraint

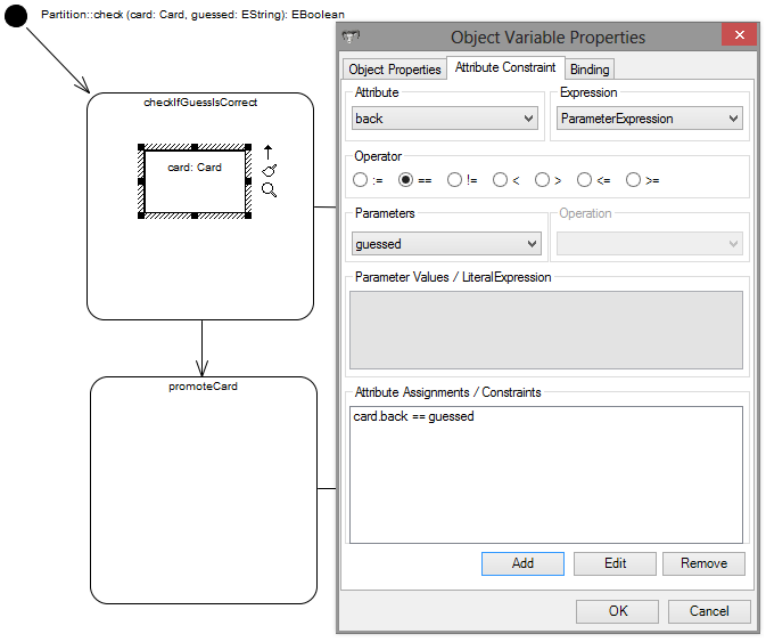


Figure 3.54: Add an attribute constraint with a parameter expression.

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 `CheckIfGuessIs-Correct` 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. 3.55).

Edge Guards

Guard Type

Choose `Failure`, repeat the process for the edge leading to `promoteCard` and choose `Success`.

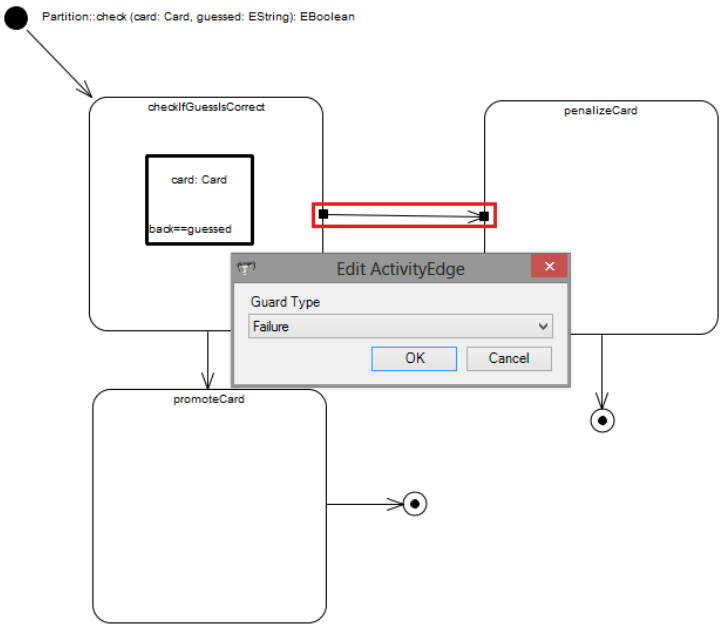


Figure 3.55: Add a transition with a guard.

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.

Extracting Patterns

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).

To extract an empty or already partially modelled story pattern, just double-click the corresponding story node (`promoteCard`) and choose `Extract Story Pattern` (Fig. 3.56). Note the new diagram that is immediately opened and created in the project browser (Fig. 3.57).

Drag & Drop

Yet another EA gesture is good old *Drag and Drop* from the project browser[10], 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. 3.58). A dialogue should

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

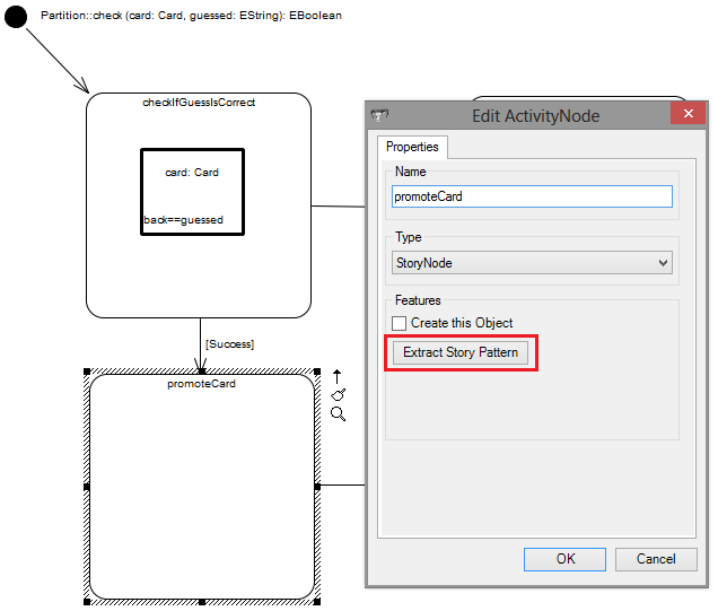


Figure 3.56: Extract a story pattern for more space and a better overview.

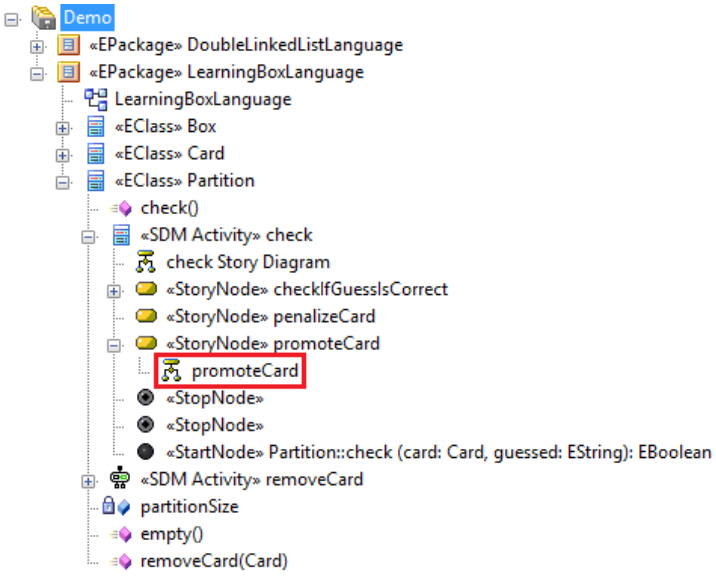


Figure 3.57: A new sub diagram is created automatically.

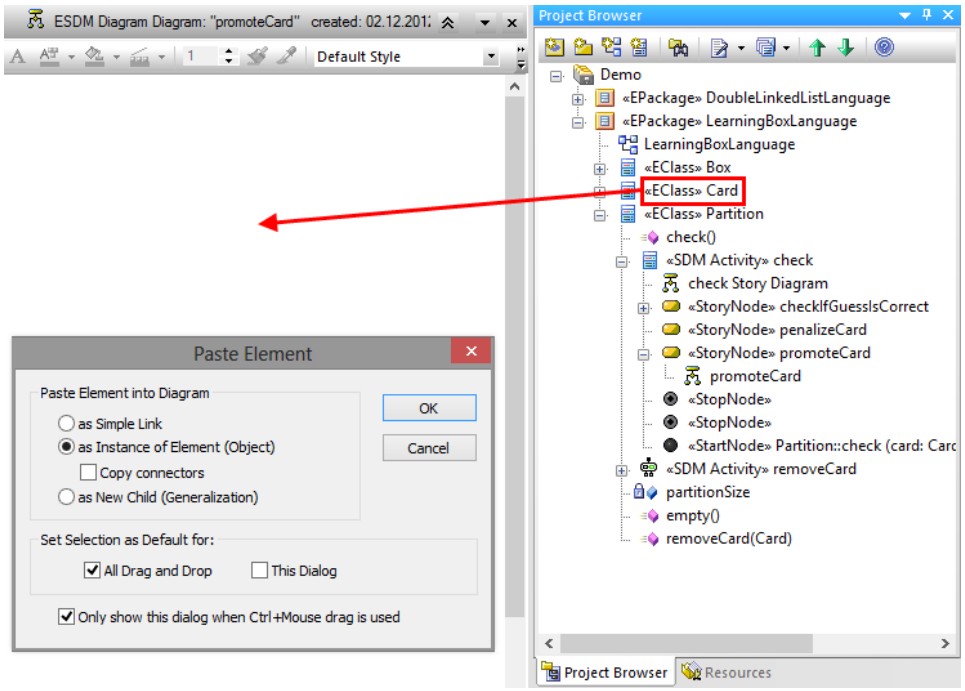


Figure 3.58: Add an object variable per drag and drop.

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.

After creating the object, you will be asked to set the object variable properties. Set its `Name` to "card" and its `Binding State` to "Bound" (Fig. 3.59).

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

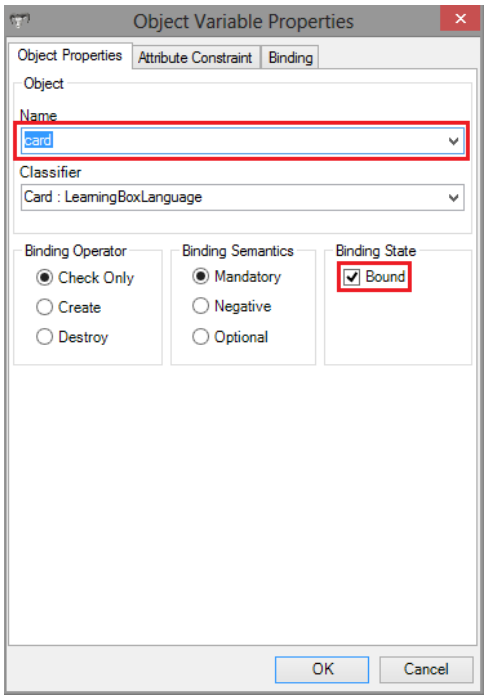


Figure 3.59: Object variable properties of the new card.

drop-down menu and can really be a great time saver for large metamodels[11].

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. 3.60). An important point to note here is that `this` and `card` are visually differentiated from `nextPartition` by their bold border lines. 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

[11] 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

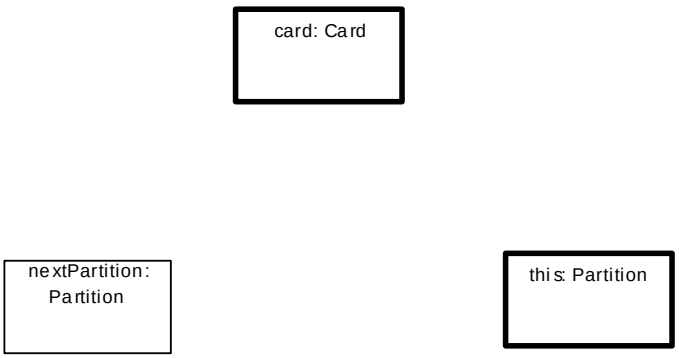


Figure 3.60: All object variables for story pattern `promoteCard`.

`nextPartition` is left empty and quick link from `this` to `nextPartition`, or vice-versa, to create a `next` link variable as indicated in Fig. 3.61.

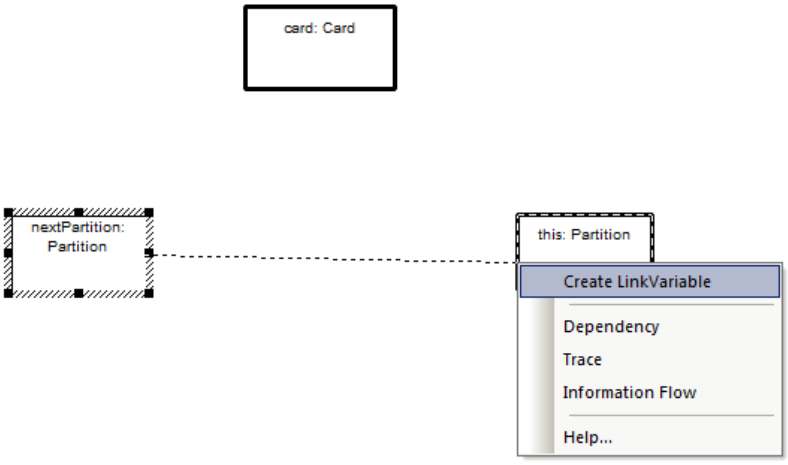


Figure 3.61: Create a link variable.

If you've done everything right, your story pattern should now closely resemble Fig. 3.62. Take a step back and reflect on what the pattern expresses.

Now repeat the process for the story node `penalizeCard`: extract the story pattern, and create all variables as depicted in Fig. 3.63. 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`.

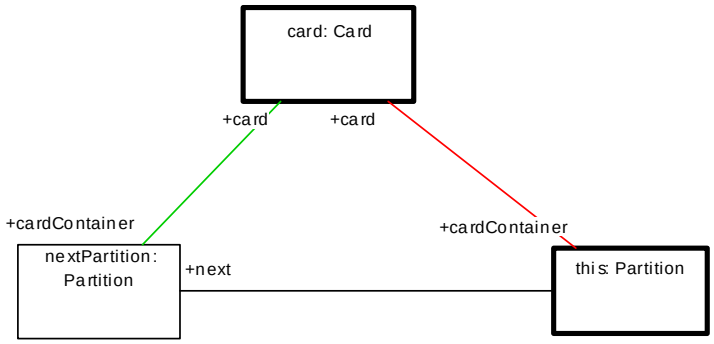


Figure 3.62: Complete story pattern for activity node `promoteCard`.

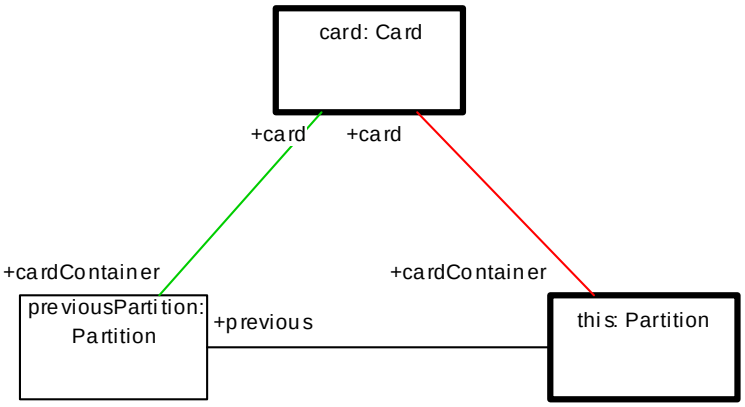


Figure 3.63: Story pattern for activity node `penalizeCard`.

Literal Expression

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. 3.64).

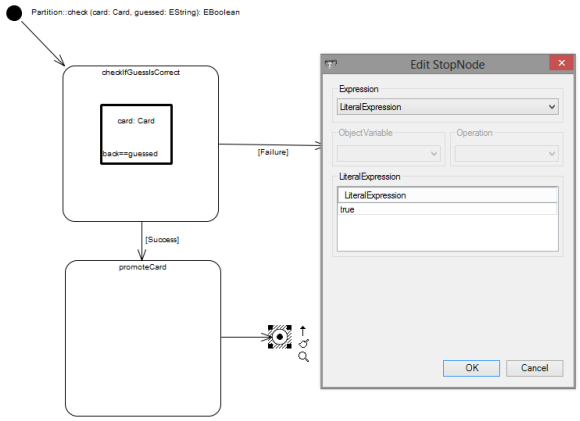


Figure 3.64: Add a return value with a literal expression.

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[12] and should be avoided if possible.

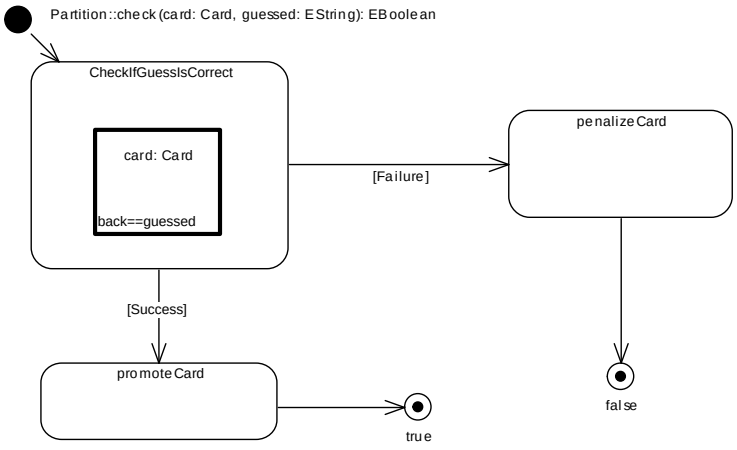


Figure 3.65: Complete SDM for `Partition::check`.

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

Type in `true` as the value of the expression (Fig. 3.64) and complete the SDM by returning `false` after penalising a card. Please ensure that your SDM (the control flow) closely resembles Fig. 3.65. 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.

3.5.3 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. 3.66).

For Each

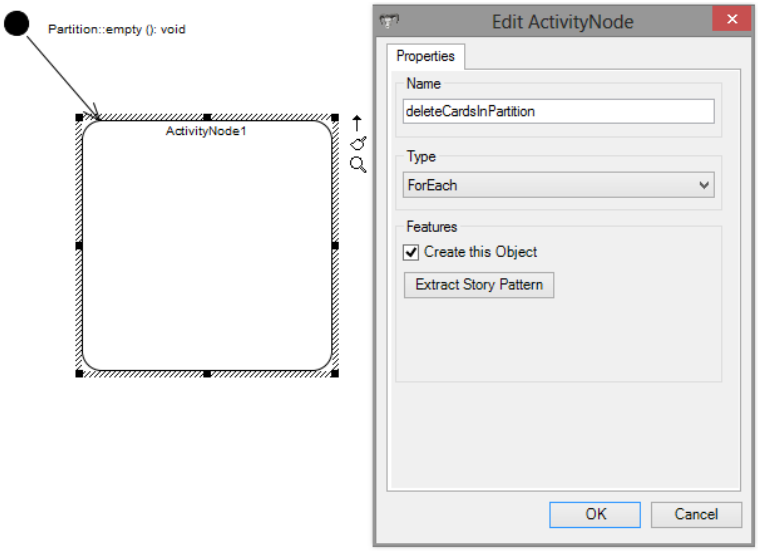


Figure 3.66: A for each loop in SDM.

A for each story node is visualised as a double node to indicate the potential repetition (Fig. 3.67). Complete the story pattern as indicated in Fig. 3.67. 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.

Non-Determinism

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.

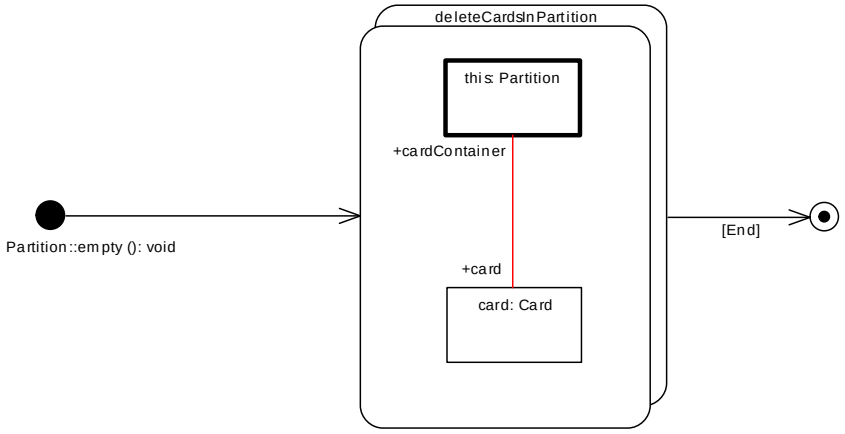


Figure 3.67: Complete story pattern with `[end]` guard.

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[13] 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?

3.5.4 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 learning box, which makes sense

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

when learning, for example, a new language. You're no longer an SDM beginner so try to model the SDM depicted in Fig. 3.68.

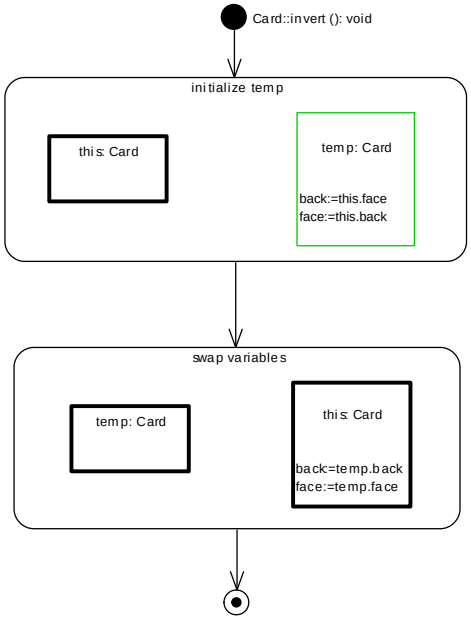


Figure 3.68: Swap back and face of the card.

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 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].

Assignments

Assertion

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.

3.5.5 Growing the box by adding a new partition

In this SDM, we shall specify how our learning 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. 3.69. This matches the box (`this`), and *any* two partitions in the box.

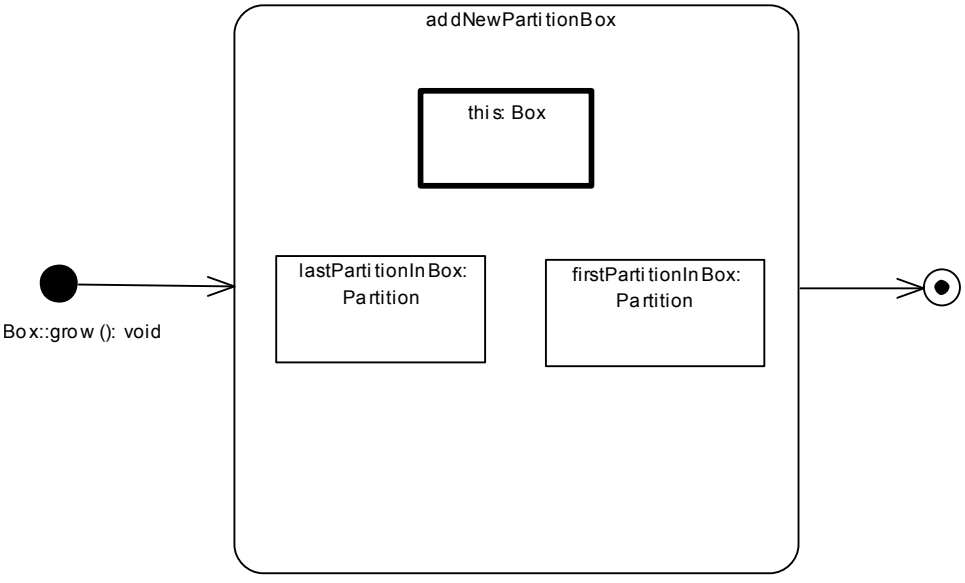


Figure 3.69: Context elements for SDM.

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 3.5.3, 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[14]. A NAC is a negative element that should *not* be present in a valid match. In the theory of algebraic graph transformations [3], NACs can be complex graphs that are much more general and powerful. In our implementation[15], however, we only support single negative elements (object or link variables).

NACs

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. 3.70). The object variable should be visualised as being cancelled or struck out.

Binding Semantics

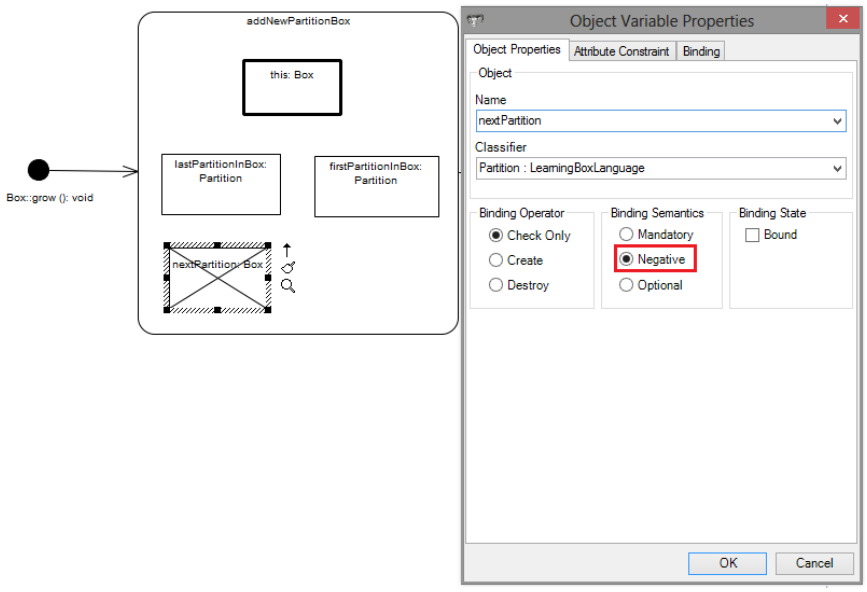


Figure 3.70: Adding a negative element.

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. 3.71. The NACs can be interpreted as follows: The first/last

[14] Pronounced \'*nak*\

[15] To be more precise CodeGen2 from Fujaba.

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.

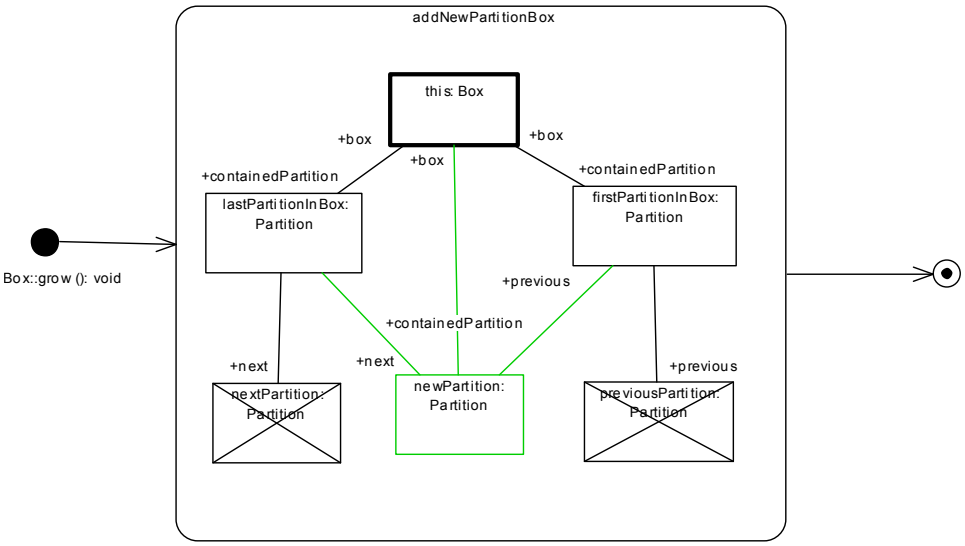


Figure 3.71: Determining the first and last partition with NACs.

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.

MethodCallExpression

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. 3.72 choosing the argument `this` as target and `determineNextSize` as the method to be invoked. Parameters could be specified by just choosing the appropriate parameter declaration between guillemets (e.g. `<Box box>`) 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. Since `determineNextSize` does not require any parameters, you can ignore the `Parameters` field this time.

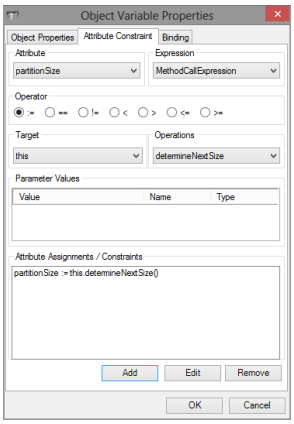


Figure 3.72: Invoking a method via a `MethodCallExpression`.

If you've done everything right, your SDM should now closely resemble Fig. 3.73.

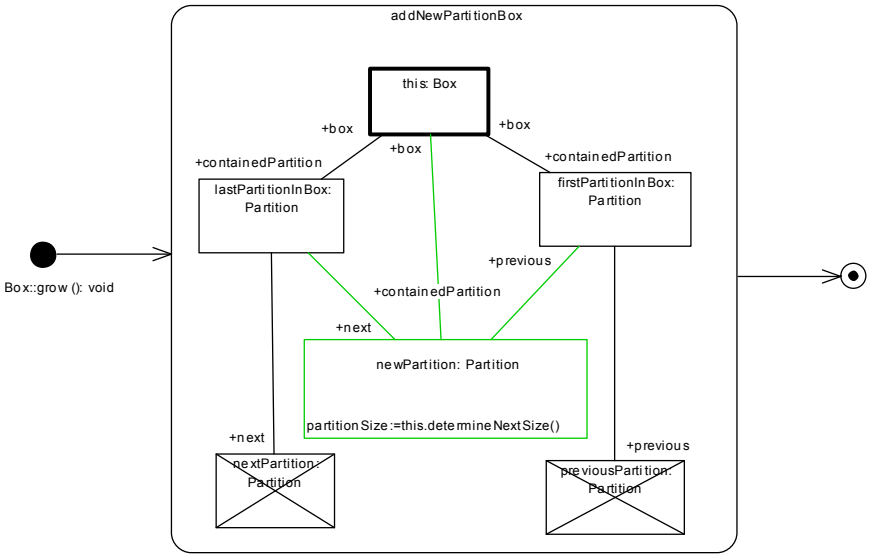


Figure 3.73: Complete SDM for `Box::grow`.

3.5.6 A string representation for our learning box

With the next SDM, we shall create a string representation for a complete learning 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. 3.5.3 we already used the `[end]` edge guard to terminate a loop and, as depicted in Fig. 3.74, 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. 3.74. 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`.

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 `addToStringRep` as the name, and `StatementNode` as the type of the activity node (Fig. 3.75). 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.

Statement Nodes

As we have already used a MethodCallExpression in an attribute constraint (Sec. 3.5.5), go ahead and click the `Method Call Expression` tab and select `MethodCallExpression` as expression, `this` as target, `addToStringRep` as operation and `card` as value of the parameter (Fig. **??**). That way, we pass the object variable `card` to the method `addToStringRep` as parameter.

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.

Recursion

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. 3.5.5).

To complete the SDM, return the final string representation of the box via an `AttributeValueExpression` in the stop node (Fig. 3.76b).

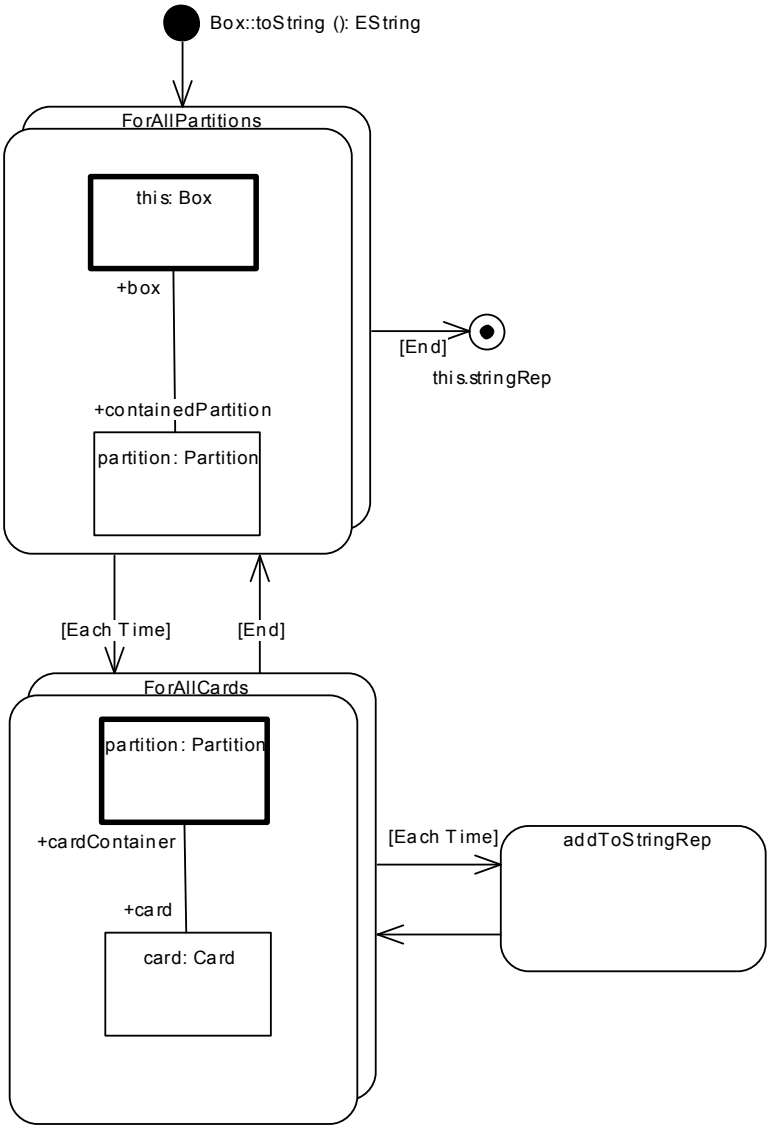


Figure 3.74: Control flow with nested loops.

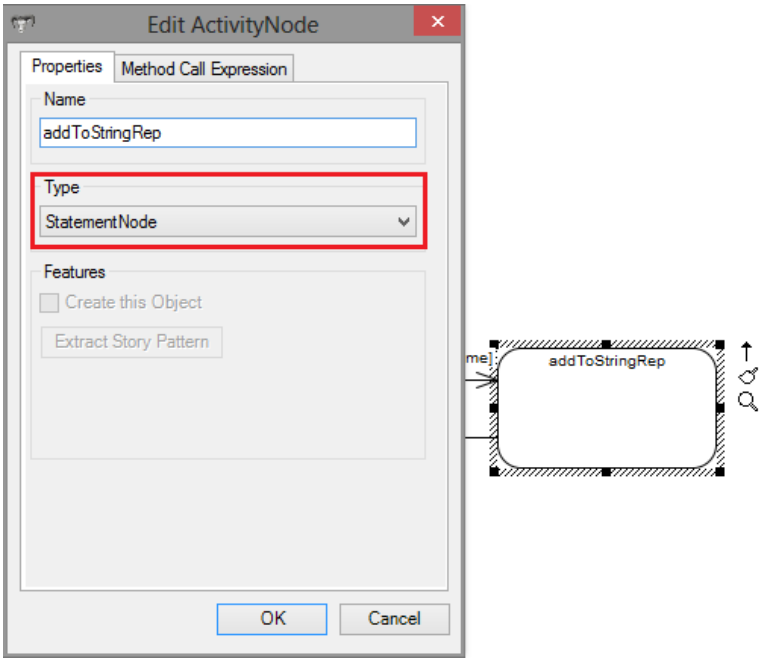
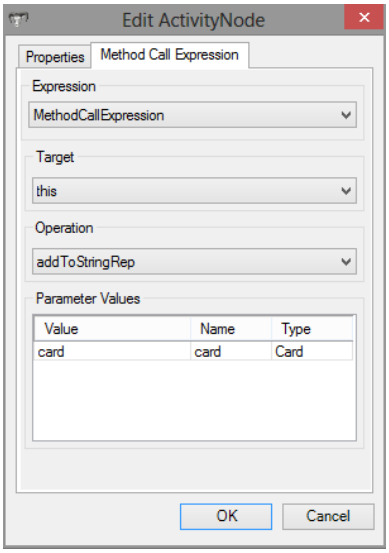
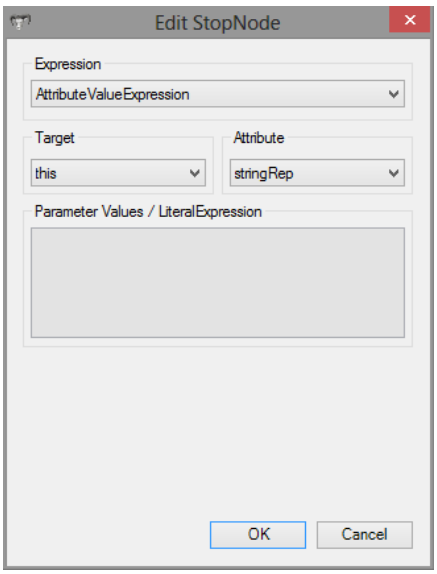


Figure 3.75: Invoking a method in a `StatementNode`.



(a) Specify a `MethodCallExpression` in the `StatementNode`.



(b) Using a `AttributeValueExpression` as a return value.

Figure 3.76

Take some time to compare and reflect on the complete SDM as depicted in Fig. 3.77. 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 `addToStringRep`could, for example, build up a string buffer and update this string representation. 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[16] means of invoking methods from SDMs.

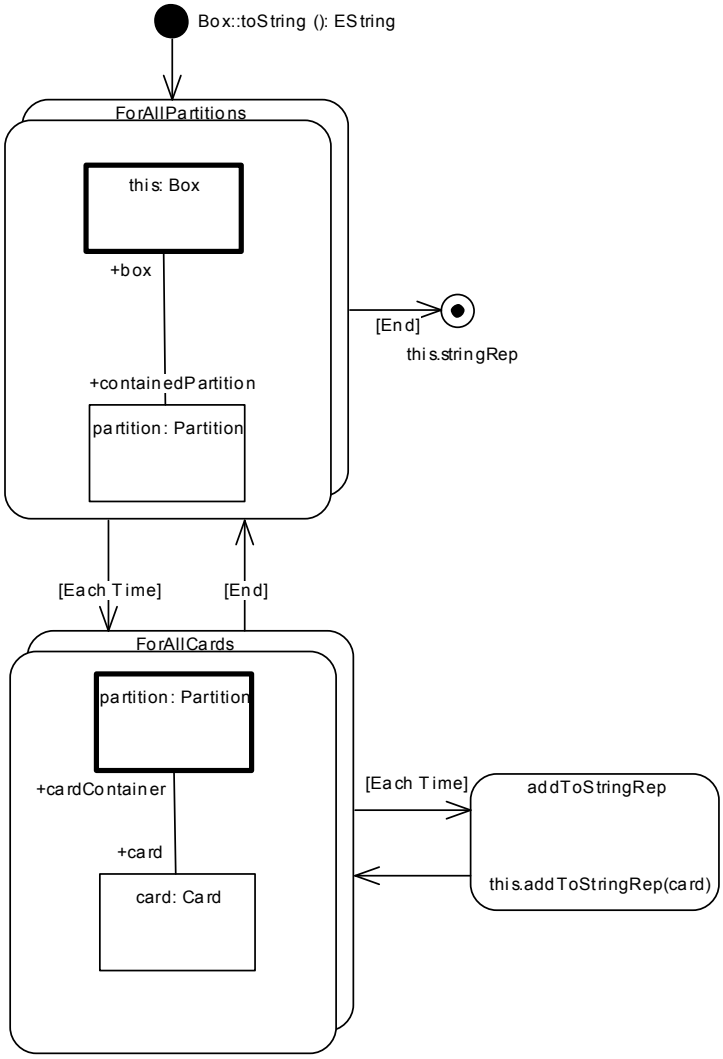


Figure 3.77: The complete SDM for `Box::toString`.

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

3.5.7 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 learning 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. 3.78).

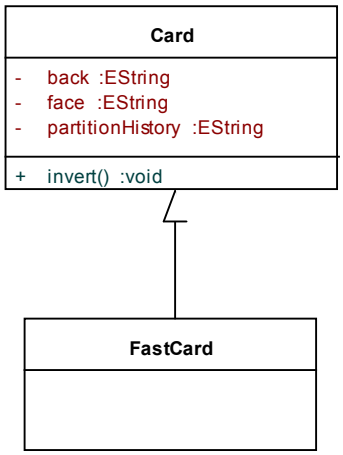


Figure 3.78: Fast cards are a special kind of card.

Now go to the SDM `check` in `Partition` and extend the control flow as depicted in Fig. 3.79.

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[17] 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

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

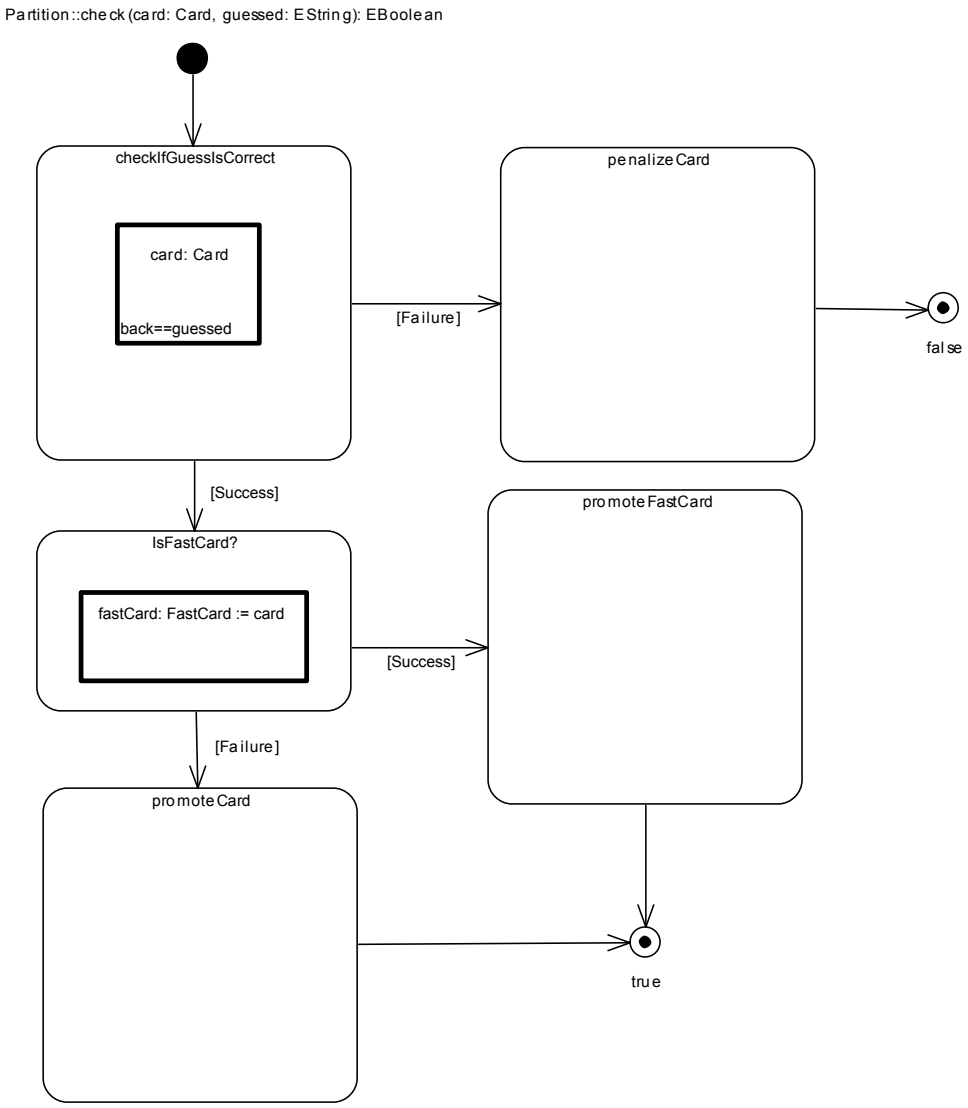


Figure 3.79: Extend check to handle fast cards.

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.

To create a binding for `fastcard`, choose the `Binding` tab in the `Object Variable Properties` dialogue (Fig. 3.80).

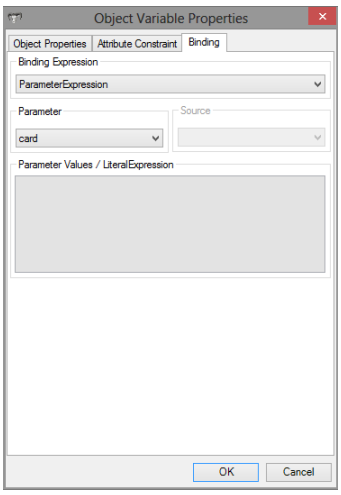


Figure 3.80: Create a binding for `fastcard`.

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 ParameterExpression 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. 3.81.

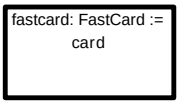


Figure 3.81: Visualisation for binding expression.

To complete the SDM, extract the story pattern of `promote fast card` and specify the pattern according to Fig. 3.82. 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. 3.5.5).

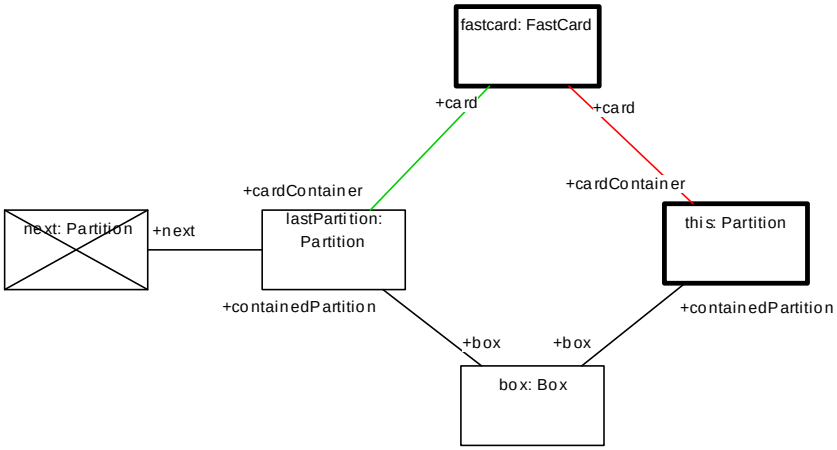


Figure 3.82: Story pattern for handling fast cards.

Export, generate code and inspect the implementation for `check`. Can you find the generated type casts for `fastcard`?

3.6 Injections

In this chapter you will learn how to implement methods that you cannot express as SDMs by adding handwritten code to classes created from your model.

Injections are inspired by partial classes in C#, and are our preferred way of providing a clean separation of generated from handwritten code.

- To implement `Box::addToStringRep`, right-click on the class `Box.java` in Eclipse and choose "eMoflon→Create injection for class" (Fig. 3.83).

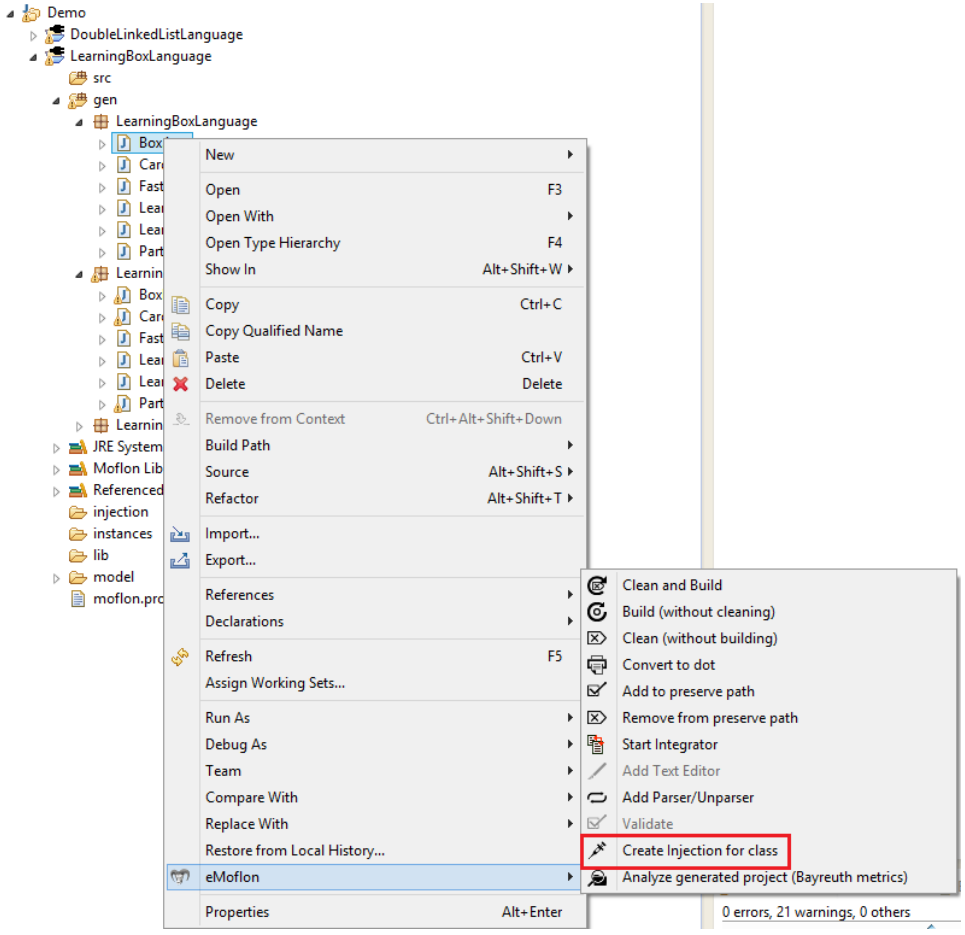


Figure 3.83: Create a new injection

This creates a new file in the `injection` folder of your project with the same packages and name as the Java class but with ".inject" as extension (Fig. 3.84). This file contains the definition of a *partial class* (Fig. 3.85).

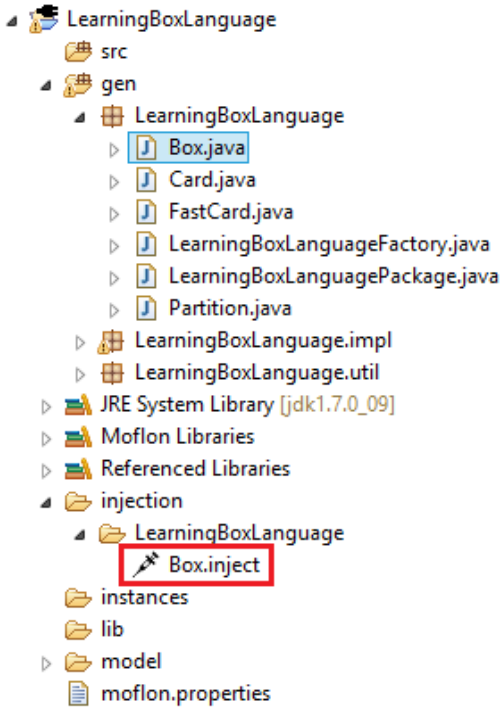


Figure 3.84: Create a new injection

```
partial class Box
{

}
```

Figure 3.85: Empty injection file

▶ Now copy and paste the code in Fig. 3.86 into your injection file.

```
partial class Box
{
    @model addToStringRep(Card card) <--
        StringBuilder sb = new StringBuilder();
        if (stringRep == null)
        {
            sb.append("BoxContent: [");
        }
        else
        {
            sb.append(stringRep);
            sb.append(", [");
        }
        sb.append(card.getFace());
        sb.append(", ");
        sb.append(card.getBack());
        sb.append("]");
        stringRep = sb.toString();
    -->
}
```

Figure 3.86: Implementation of helper method as an injection

▶ Rebuild your project (eMoflon → Clean and build) and this code will be injected in `LearningBoxLanguage.impl.BoxImpl.java` (Fig. 3.87). For more information on injections, read A.5.

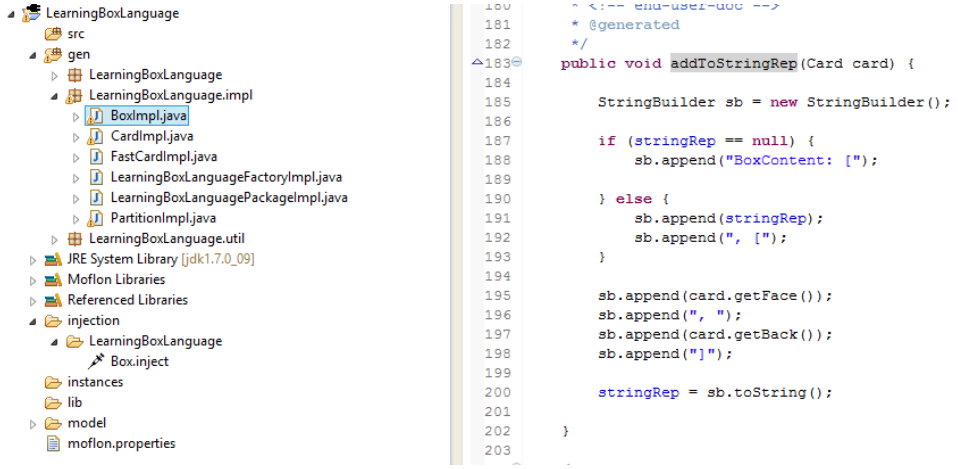


Figure 3.87: Injected code in BoxImpl after code generation

Chapter 4

Grokking EA

In this chapter, we have collected a few of the most important tips and tricks for working productively with EA. You can of course decide to skip over to the next chapter but we really believe that spending the time to grok EA is necessary for a pleasant modelling experience!

4.1 How to lay out elements

Layout is always an important factor when using a visual language. A well laid out diagram is easier to understand and by centralizing important elements or clustering related elements, you can actually impart additional information via a well-chosen layout.

- To lay out a group of elements select them by drawing a selection box around them (or select them one by one by holding down `Ctrl` and clicking on each element).

- At the right side of the element that was selected last, a little symbol appears (Fig. 4.1). Click on the symbol to obtain different options that are applied to all selected elements simultaneously. Experiment a bit to find out what effect each option has. The last symbol opens a further drop-down menu with standard layout algorithms.

- Right-clicking one of the selected elements opens a different menu with a further set of layout options (Fig. 4.2). Especially `Align Centers` can be pretty useful.

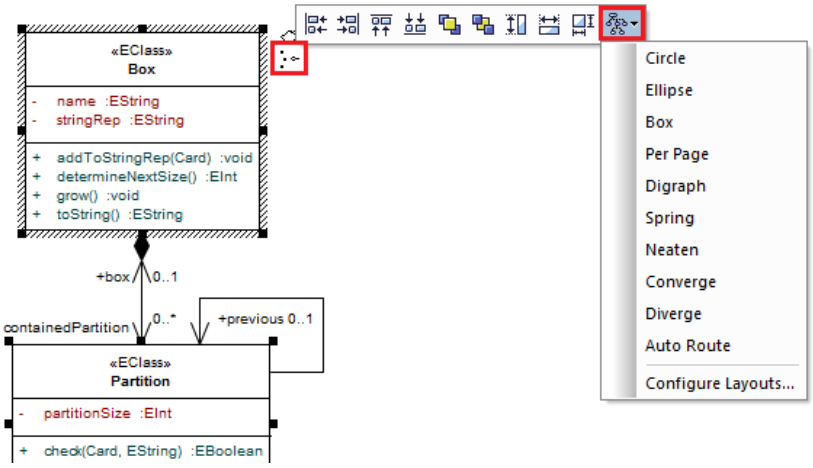


Figure 4.1: How to layout elements

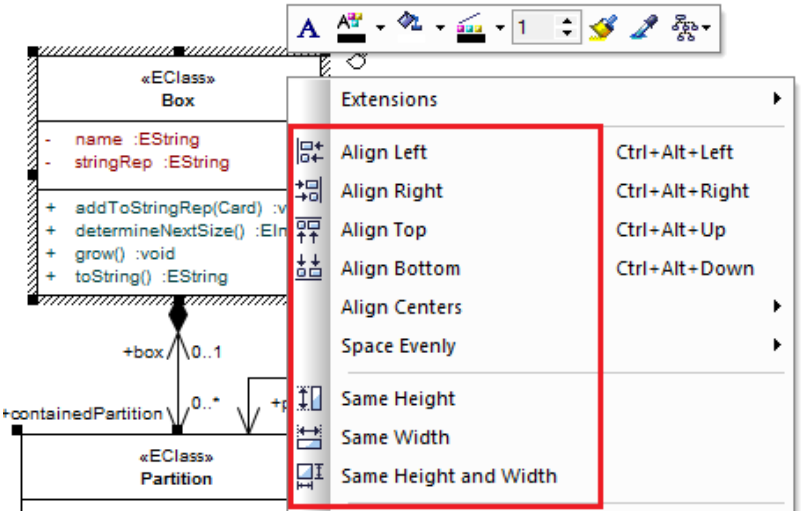


Figure 4.2: Further layout options

4.2 Bending lines to your will

Almost as important as a good layout is getting lines to be just the way you want them to be. In EA you can add and remove bending points which can be used to control the appearance of a line.

- Hold down `Ctrl` and click on a line to create a bending point (Fig. 4.3). You can now pull and place the bending point as you wish.

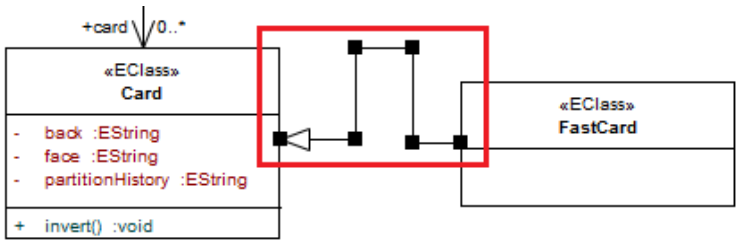


Figure 4.3: How to bend lines

- You can create as many bending points as you wish, or *remove* them also by holding down `Ctrl` and clicking on the point to be deleted.

4.3 Deleting vs. removing elements from diagrams

A central feature that new users should understand as soon as possible is the way EA handles diagrams. A diagram is simply treated as a *view* of the complete "model" in EA. The complete model can always be browed in a tree view (the package browser) and contains all elements that are exported. Diagrams typically do not contain all elements and one usually uses multiple possibly "redundant" diagrams to show different parts of the model. Thinking in this frame is crucial and provides a pragmatic solution to the problem of having huge unmaintainable diagrams. A tricky consequence one must get used to is that *removing* an element from a diagram does *not* delete it from the model.

- One of the most common mistakes of new users is to remove an element from a diagram by pressing `Del` and expecting the element to be deleted from the model. This is however not the case as the element is only removed from the current diagram and is still in the model and thus in the package browser (Fig. 4.4).

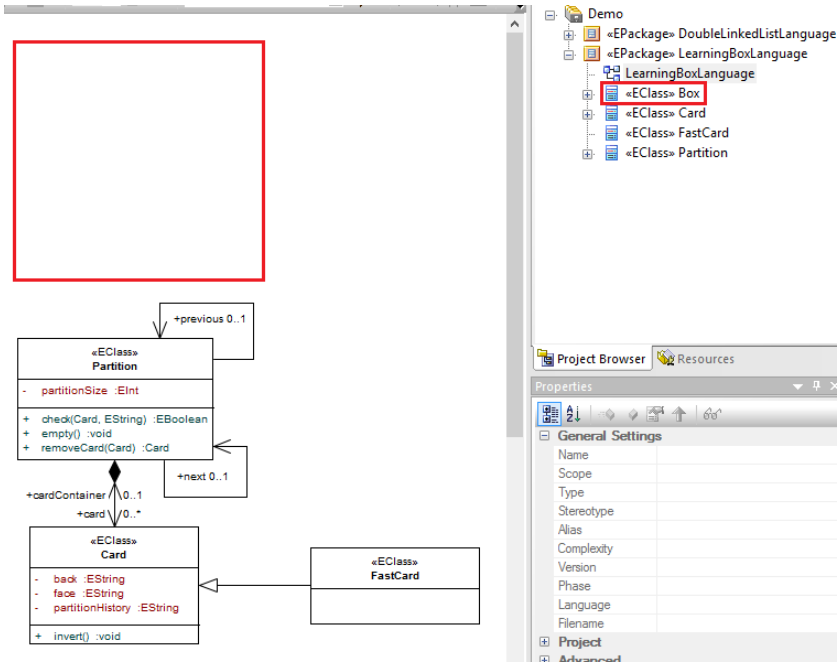


Figure 4.4: Removing an element from a diagram via pressing `Del` does not delete it from the model and it is still present in the package browser

- By pressing `Ctrl+Del` and confirming (Fig. 4.5) you can *delete* an element from a diagram *and* completely from the model as well (the element will no longer be in the `Project Browser`). Elements can also be deleted via the context menu in the `Project Browser` (invoked by right-clicking the element in the `Project Browser`).

4.4 Excluding certain projects from the export

Sometimes it might be necessary to be able to exclude projects from the export (`Export All to Workspace` option). This might be (i) because the project is still work in progress and simply not yet ready to be exported, (ii) because the complete project is present in your Eclipse workspace but has not been modelled completely in EA and you wish to do this gradually on demand (this is currently the recommended strategy as we do not have an import yet), (iii) because the project is not meant to be present in your Eclipse workspace as generated code and is instead provided via a plugin (this is usually the case for standard metamodels like Ecore, UML etc.), or (iv) because the project is rather large and pretty stable, and you do not want to wait each time for an unnecessary export (and you do not wish

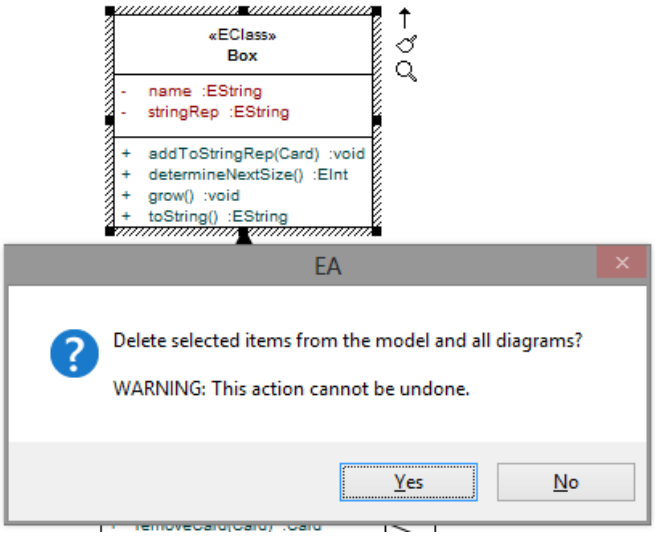


Figure 4.5: Deleting an element from a diagram and from the model

to click and export the other projects individually). Whatever the reason, you can achieve this by setting a certain *tagged value* of the project to be ignored:

- Select "View/Tagged Values" from the menu bar (Fig. 4.6).
- A tagged value *Moflon::Export* should already be present and be set to `true` per default (Fig. 4.7). If this is not the case then create it afresh. If you want the project to be ignored by the export, change the value of Moflon::Export to `false` (and conversely back to `true` to export it again).

4.5 Getting verbose!

Although we use colours in SDMs to indicate when an element is to be matched (black), created (green), or destroyed (red), sometimes, especially for a black and white printout, it makes sense to indicate this via explicit stereotypes.

- Rightclick on a diagram and select "Extensions/extras/Set Moflon::Verbose to true" to enable `<<create>>` and `<<destroy>>` stereotypes (Fig. 4.8).
- You can hide the stereotypes again by selecting "Extensions/extras/Set Moflon:: Verbose to false" (Fig. 4.8).

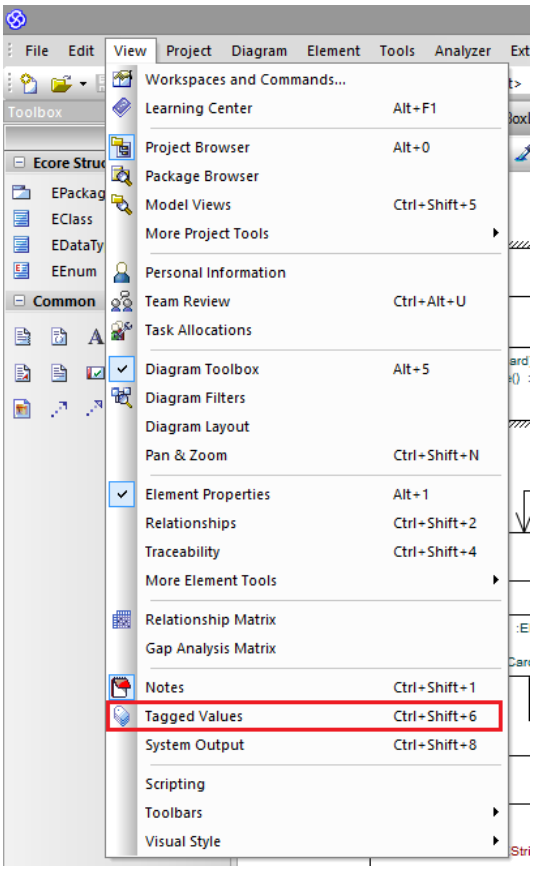


Figure 4.6: view the TaggedValue

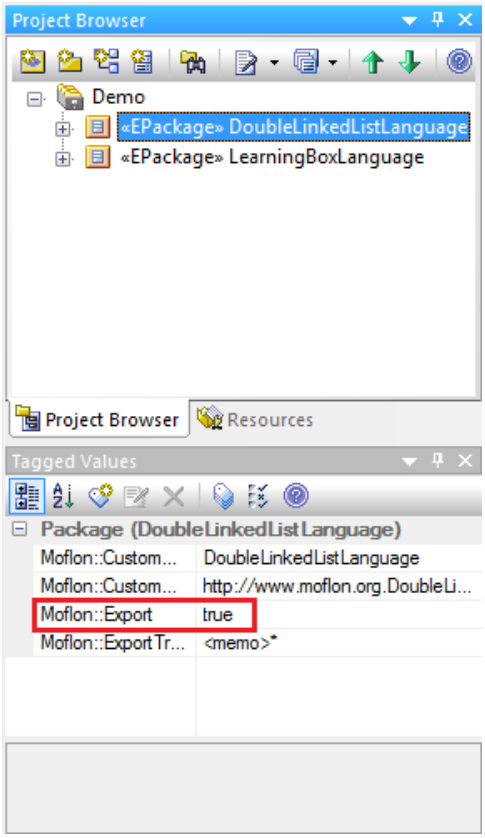


Figure 4.7: Tagged value Moflon::Export is used to ignore projects

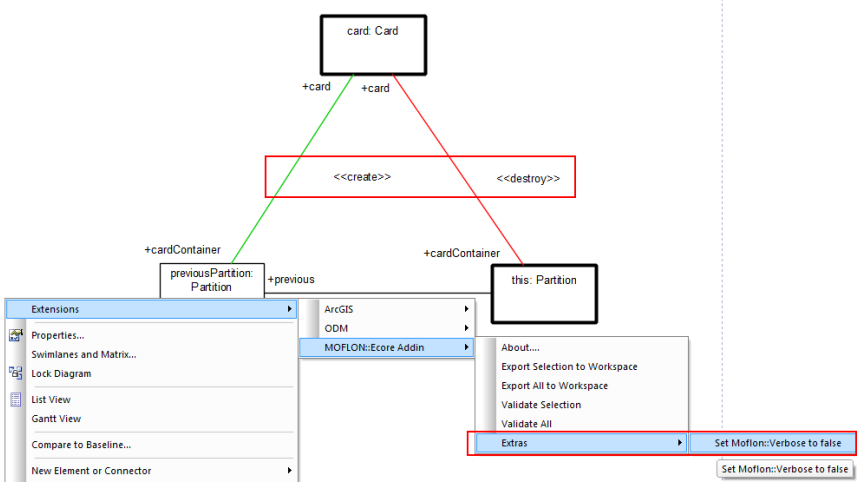


Figure 4.8: Verbal change of links

4.6 Duplicating elements via drag&drop

By holding down `Ctrl` and dragging objects in diagrams or in the project browser, you can copy (duplicate) most elements. Typically, the appropriate dialogue pops up for the new element (Fig. 4.9).

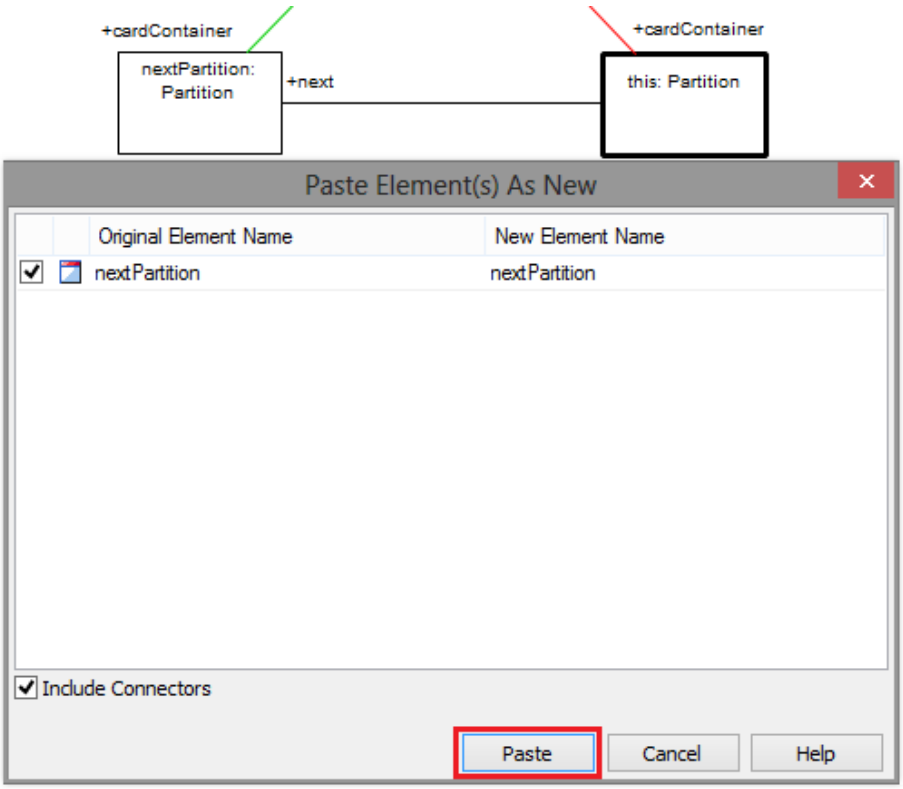


Figure 4.9: Copying objects

4.7 Seek, and ye shall find ...

EA has a model search function that can be quite handy for large models with thousands of elements:

- Select "Model Search Window" and enter the name of an element you wish to search for (Fig. 4.10).
- All elements that meet the search criteria are listed and you can right-click on each of the items and select "Find in Diagrams" or "Find in Project Browser" to locate the elements.

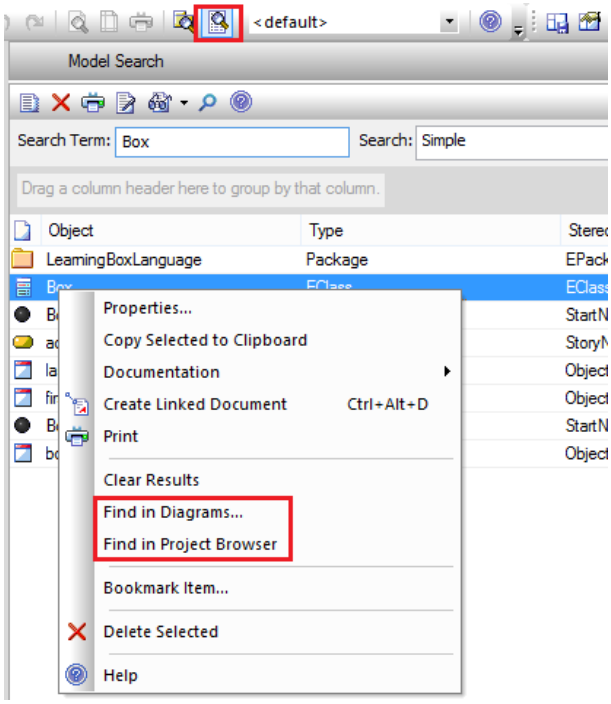


Figure 4.10: Model Search Window

- In a similar way, you can locate the corresponding class of an object by right clicking and selecting "Find/Locate Classifier in Project Browser".

For more flexible search functionality, see Appendix A.1.

Chapter 5

A Dictionary Language

A Taxonomy of Model Transformations

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

Model-to-Text Transformations

There are many *types* of model transformations and [2, 8] 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*.

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 to the target model m_{trg}.

Endogenous Model Transformations

Δ 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 learning box) are examples of endogenous transformations.

Exogenous Model Transformations

Δ is *exogenous*, if m_{src} and m_{trg} are instances of different metamodels. In this chapter, we shall complement our learning 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 and only specific words are looked-up now and then. A learning box, on the other hand, is more geared towards supporting the actual memorization process. Ergo? One could start with a learning 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 learning 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 learning box.

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

In-Place Model Transformations

Δ operates *in-place*, if m_{src} is destructively transformed to m_{trg}. The SDMs for our learning box (e.g. grow or check) are examples for in-place transformations as they perform changes directly to a source model, transforming it destructively into the target model.

Out-Place Model Transformations

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

Although endogenous + in-place is the natural case for SDMs (like for our learning 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?

Δ 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.

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 learning 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.

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

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

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.

What is Moca?

Fig. 5.1 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.

5.1 Setting up your M2T workspace

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 [9] 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.

The first 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. 5.2).
- Choose "Dictionary" as project name and select `Add eMoflon languages` as depicted in Fig. 5.3.
- After the project is created as usual (Fig. 5.4) double-click the EAP file to open it.

[2] Some languages are syntactically quite challenging.
[3] For simple cases, *regular expressions* can also be used.
[4] Extended Backus-Naur Form

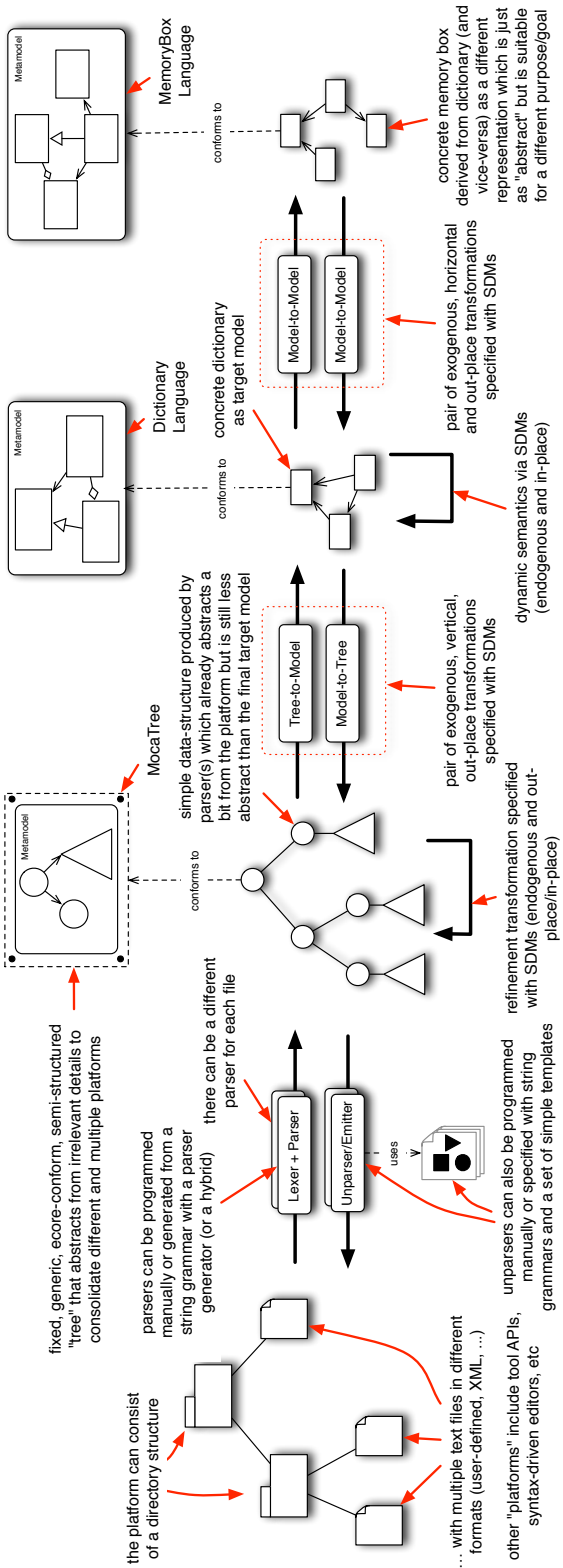


Figure 5.1: Overview of model-to-text with the MOCA framework

Figure 5.2: Invoking the `New Metamodel` wizard.

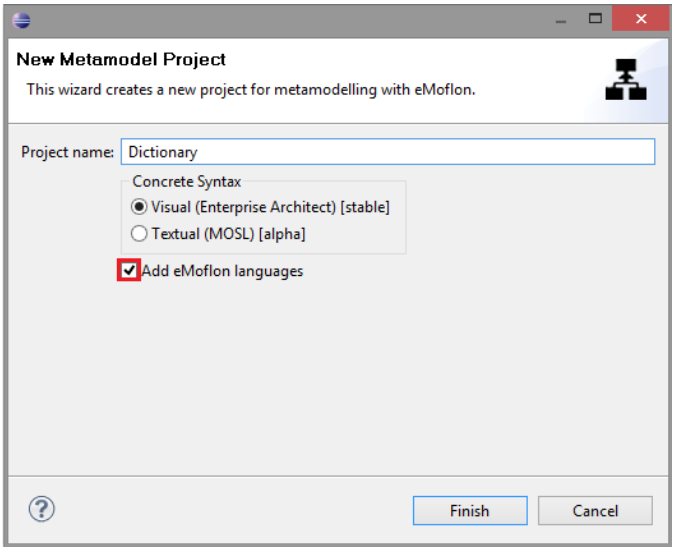


Figure 5.3: Add Metamodel project with MOCA support

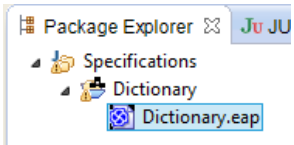


Figure 5.4: Eclipse workspace after creating the `Dictionary` project

- 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. 5.5). Note that the `MocaTree` package has a special tagged value `Moflon::Export` that is set to `false`[5]. 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.

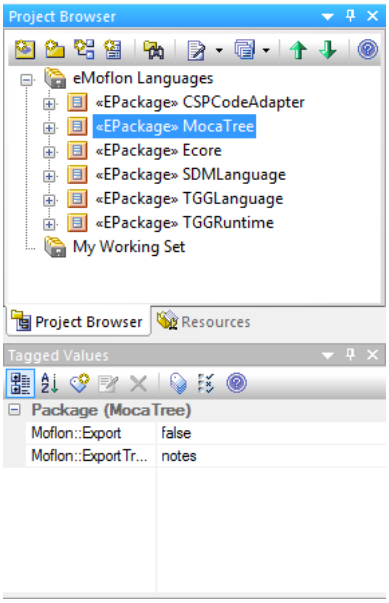


Figure 5.5: `MocaTree` in default EA project

Go ahead and inspect the `MocaTree` metamodel (Fig. 5.6). It basically combines concepts from a filesystem (folders and files), XML concepts (text-only nodes and attributes), and a general indexed[6] containment hierarchy.

- Add a new package `DictionaryLanguage` and model the required classes and relationships for our dictionary language (Fig. 5.7).

[5] Tagged values can be viewed in the `Tagged Values` view in EA (Fig. 5.5).

[6] The index attribute in `TreeElement` can be used to demand a certain *order* of nodes in an SDM, which is otherwise not guaranteed by default (order is in general non-deterministic).

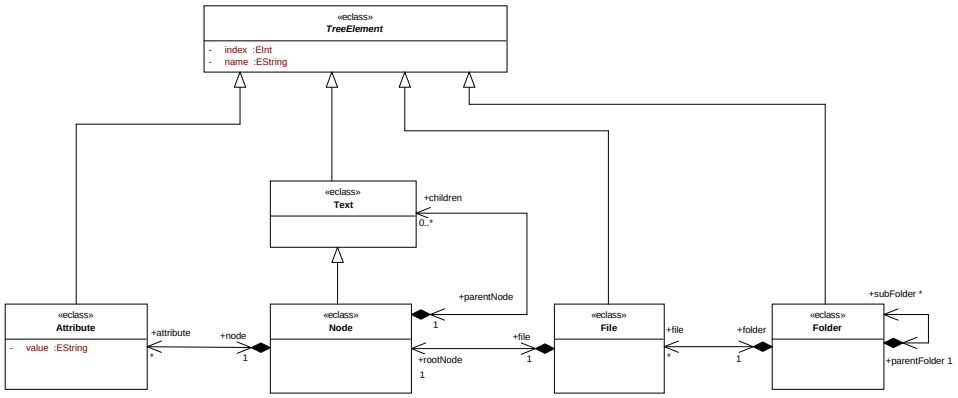


Figure 5.6: `Mocatree` Metamodel

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.

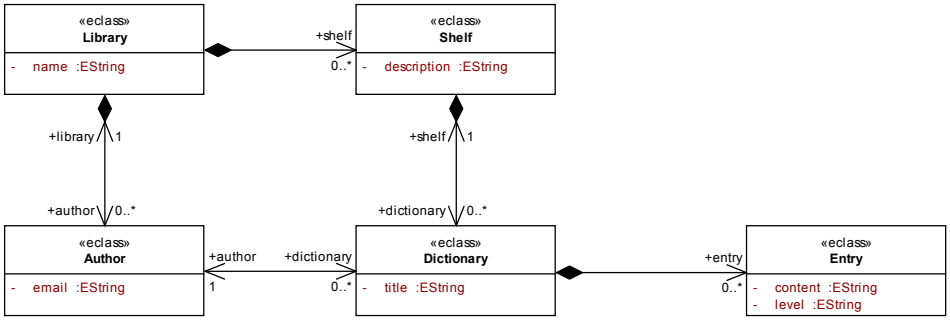


Figure 5.7: Dictionary Metamodel

▶ For the moment, add an empty package in EA named `Dictionary-CodeAdapter` so that your EA workspace closely resembles Fig. 5.8.

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.

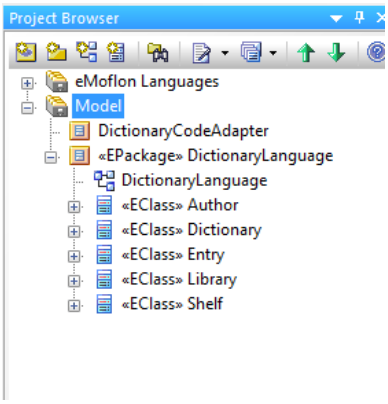


Figure 5.8: EA workspace before exporting

- Export as usual and ensure that your Eclipse workspace closely resembles Fig. 5.9. Note especially the library nodes (`Moflon` and `Moca`) that reference jars for all required dependencies. (If your `DictionaryCodeAdapter` is not exported or you receive an error message while exporting it, you can add an empty diagram to it)
- Right-click on `DictionaryCodeAdapter` one more time and choose "Add Parser/Unparser", this time from the eMoflon context menu (Fig 5.10).
- In the wizard dialogue (Fig 5.11), 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`.

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. 5.12.

5.2 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 5.13 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 step is called *lexing* and it simplifies the actual comprehension of the complete text. Interestingly human beings actually comprehend text in a similar manner, one

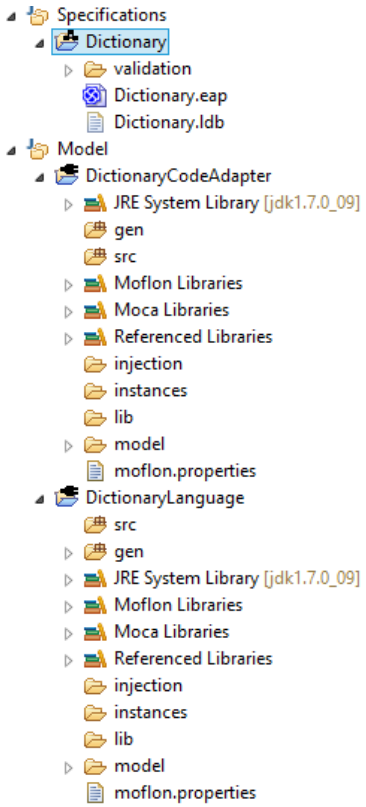


Figure 5.9: Workspace after export to Eclipse

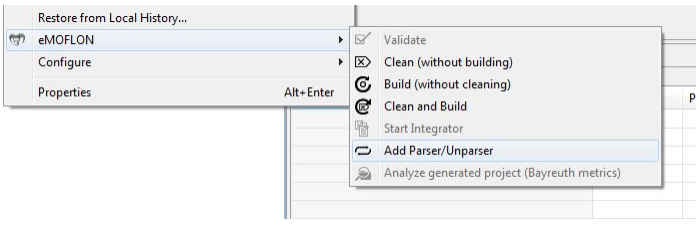


Figure 5.10: Invoking the `Add Parser/Unparser` wizard

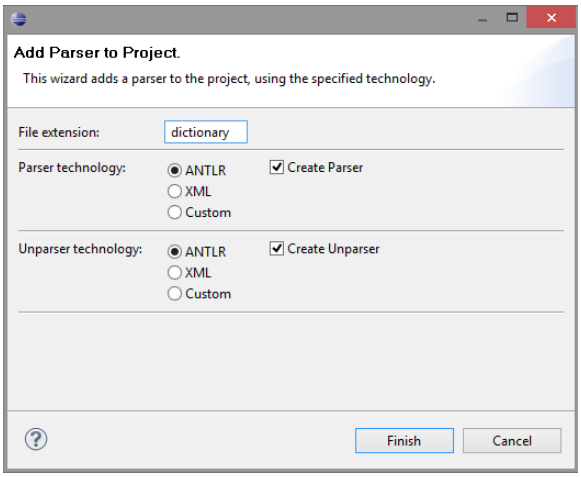


Figure 5.11: Add Parser/Unparser

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. 5.13, `ANTLR` can automatically generate a lexer in Java from a compact specification as depicted in Fig. 5.14. This is actually a DSL for lexing and is explained in detail in [9]. 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 [9].

- Edit `DictionaryLexer.g` so it closely resembles Fig. 5.14. Be careful to avoid any typos and mistakes. Save and make sure it compiles.

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. 5.15. 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.

- Edit `DictionaryParser.g` so it closely resembles Fig. 5.16. As with the lexer, avoid typos and mistakes and make sure it compiles.

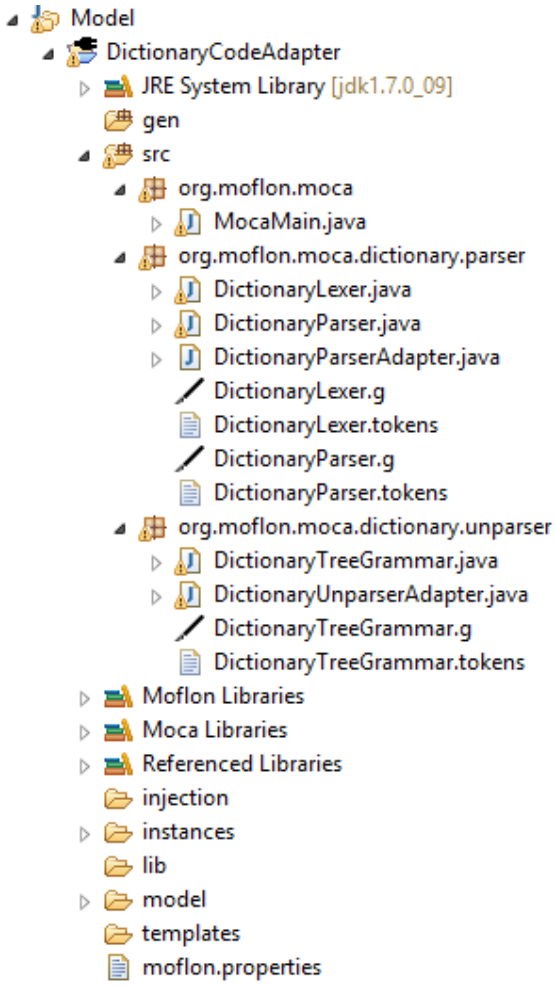


Figure 5.12: Workspace after wizard finishes

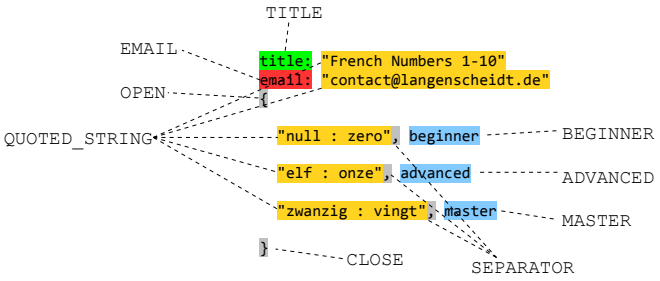


Figure 5.13: Identified tokens in a dictionary file.

```
1  lexer grammar DictionaryLexer;
2
3  @header {
4  package org.moflon.moca.dictionary.parser;
5  import org.moflon.moca.MocaUtil;
6  }
7
8  WS: (' ' | '\t' | '\n' | '\r')+  { skip(); };
9
10 TITLE: 'title:';
11
12 EMAIL: 'email:';
13
14 QUOTED_STRING:   '"' .* '"' { MocaUtil.trim(this, 1, 1); };
15
16 BEGINNER: 'beginner';
17
18 ADVANCED: 'advanced';
19
20 MASTER: 'master';
21
22 OPEN: '{';
23
24 CLOSE: '}';
25
26 SEPARATOR: ',';
27
28 DICTIONARY: 'DICTIONARY';
29
30 ENTRY: 'ENTRY';
```

Figure 5.14: Lexer grammar

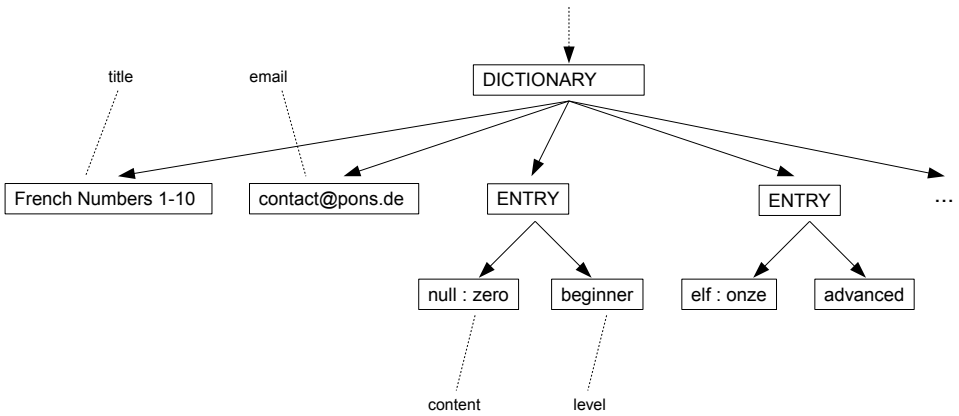


Figure 5.15: MocaTree structure

```
1  parser grammar DictionaryParser;
2
3  options {
4    language   = Java;
5    tokenVocab = DictionaryLexer;
6    output     = AST;
7  }
8
9  @header {
10 package org.moflon.moca.dictionary.parser;
11 }
12
13 main: title email? OPEN entry+ CLOSE  -> ^(DICTIONARY title email? entry+ );
14
15 title: TITLE QUOTED_STRING -> QUOTED_STRING;
16
17 email: EMAIL QUOTED_STRING -> QUOTED_STRING;
18
19 entry: QUOTED_STRING SEPARATOR level  -> ^(ENTRY QUOTED_STRING level);
20
21 level: ( BEGINNER | ADVANCED | MASTER );
```

Figure 5.16: Parser grammar

The parser grammar is quite similar to the lexer grammar, but there are *parser actions* after the `->` symbol, which build up the tree. Using this simple tree language, one can (1) abstract from tokens like { or }, which are just *syntactical noise*[7] and (2) enrich the tree with structural nodes like `ENTRY`, which add explicit structure to the tree. Please refer to [9] and online resources for a detailed explanation of the syntax and semantics of the parser grammar supported by `ANTLR`.

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. 5.17. For the moment, please comment out line 24[8] as we shall define the unparser, i.e., model-to-text a bit later. Do not change anything else and 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, for example, to 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 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)

[7] In this context, content that is irrelevant for our model.
[8] If you forget this the default implementation of the unparser will throw an exception.

```java
package org.moflon.moca;
import java.io.File;

public class MocaMain
{

  private static CodeAdapter codeAdapter;

  public static void main(String[] args)
  {
    BasicConfigurator.configure();

    // Register parsers and unparsers
    codeAdapter = MocaFactory.eINSTANCE.createCodeAdapter();
    codeAdapter.getParser().add(new DictionaryParserAdapter());
    codeAdapter.getUnparser().add(new DictionaryUnparserAdapter());

    // Perform text-to-tree
    Folder tree = codeAdapter.parse(new File("instances/in/"));

    // Save tree to file
    eMoflonEMFUtil.saveModel(MocaTreePackage.eINSTANCE, tree, "instances/tree.xmi");

    // Perform tree-to-model
    //TODO

    // Save model to file
    //TODO

    // Perform model-to-tree
    //TODO

    // Perform tree-to-text (using initial tree)
    codeAdapter.unparse("instances/out", tree);
  }
}
```

Figure 5.17: Generated main method

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. 5.1 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. 5.18 in `Dictionary-CodeAdapter` and enter the contents from Table 5.1 for each of the four `dictionary` files[9].

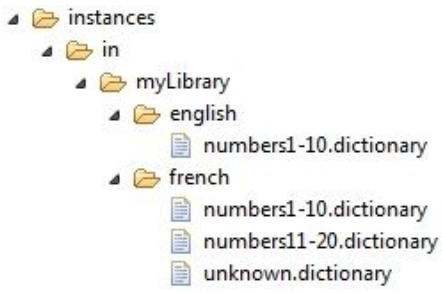


Figure 5.18: Input directory structure.

- After creating all the dictionaries, run `MocaMain.java` as a normal Java application. If everything works out right, a file called `tree.xmi` and an `out` folder should be created in the `instances` directory[10]. Inspect their contents and compare them to Fig. 5.19. 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`.

- Double-click `tree.xmi`[11] and compare the contents to Fig. 5.20. 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.

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

[10] You probably have to refresh the `instances` folder to see the newly created files.

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

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 5.1: Input files containing dictionaries.

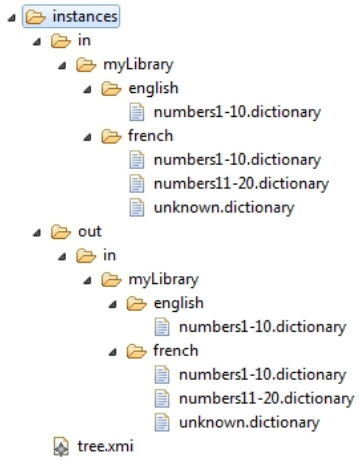


Figure 5.19: Directory `instances` after parsing

This is important 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`.

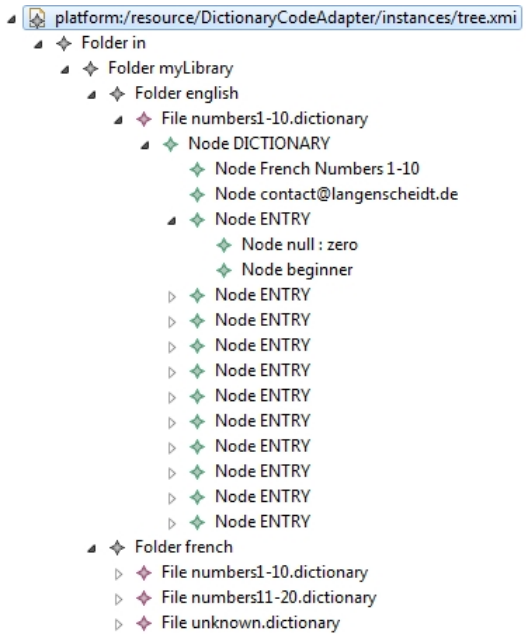


Figure 5.20: MocaTree created by the framework using our parser

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.

5.3 Tree-to-Model transformation with SDMs

The next step is to specify a set of SDMs to transform our tree to an instance of our dictionary metamodel. Take a look at the overview (Fig. 5.1) 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 references to the `Library` class as depicted in Fig. 5.21.[12] Some of the methods are for the model-to-text transformation and the corresponding SDMs shall be handled in Chapter 5.4.

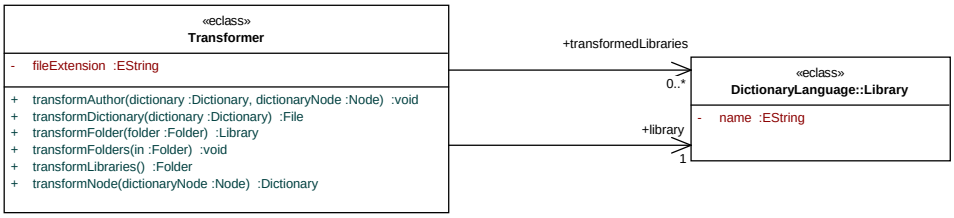


Figure 5.21: Transformer class with methods for SDMs

- The first SDM `transformFolders(Folder) :void` (Fig. 5.22) simply iterates through all the library folders and stores the created libraries in the collection `transformedLibraries`.
- `transformFolder(Folder) :Library` (Fig. 5.23) does the main work and creates a corresponding library with shelves for the given folder, delegating the actual creation of dictionaries to `transformNode`. The SDM uses features that we have treated in previous chapters.

[12] Do not forget to hold `Ctrl` when dragging the class in and to choose "as Simple Link" so that the class itself is added to the diagram and not an instance of it.

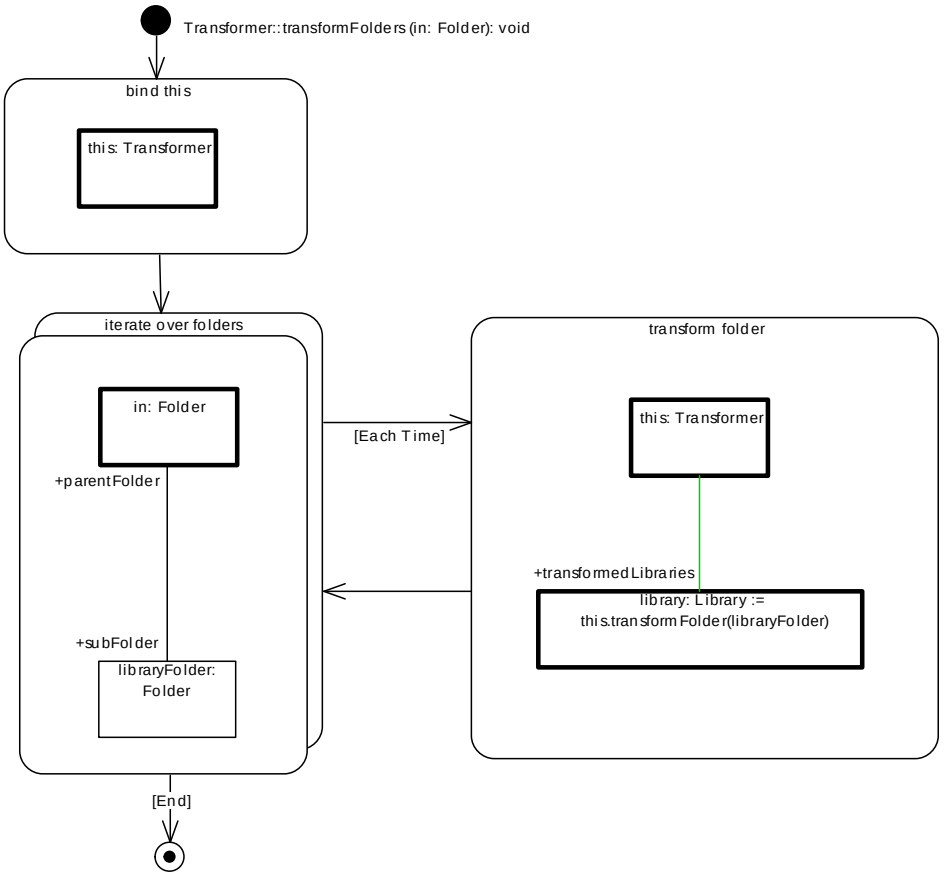


Figure 5.22: Iterate over all subfolders of the given folder

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?

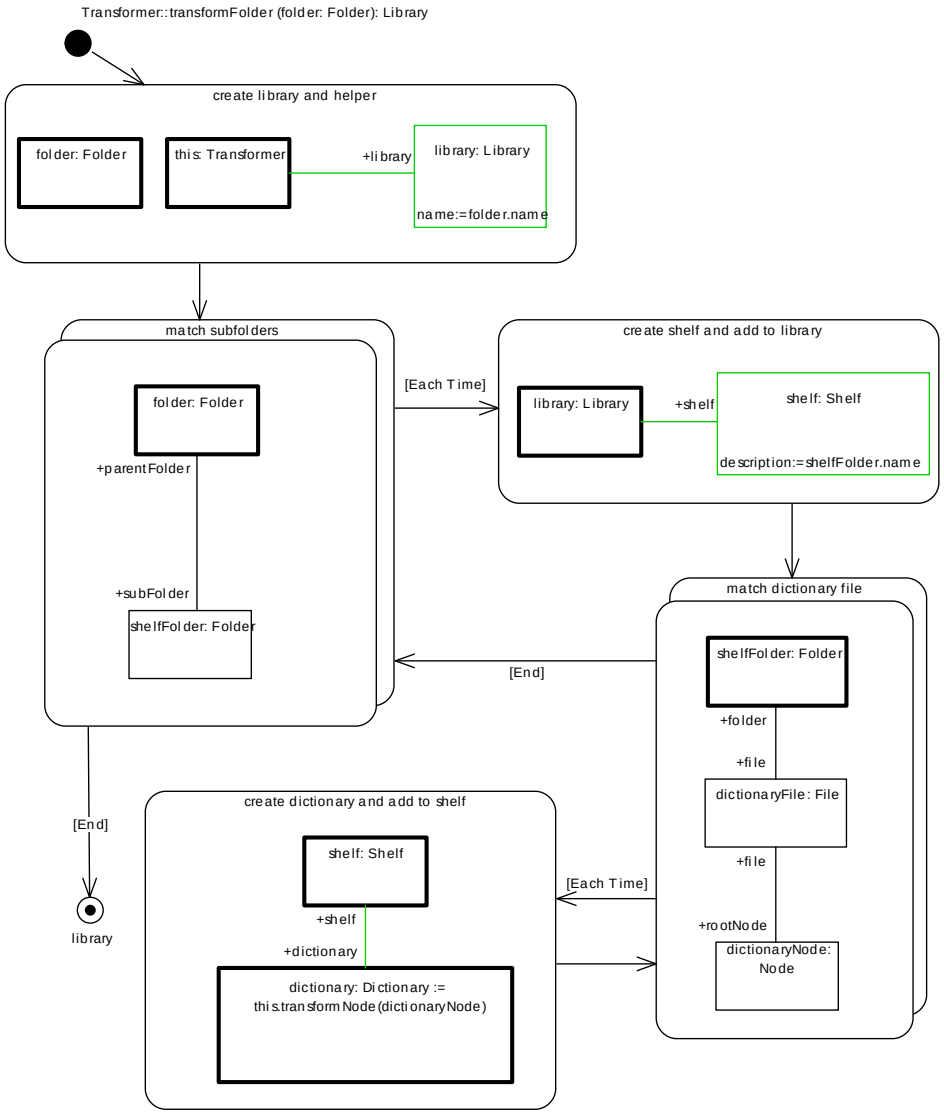


Figure 5.23: Transforming the outermost folder into a library

- The next SDM, `transformNode(Node) :Dictionary` (Fig. 5.24) takes a node, representing a dictionary, and builds up a dictionary object, adding entries appropriately. It further delegates creation of authors to `transformAuthor`.[13] 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.

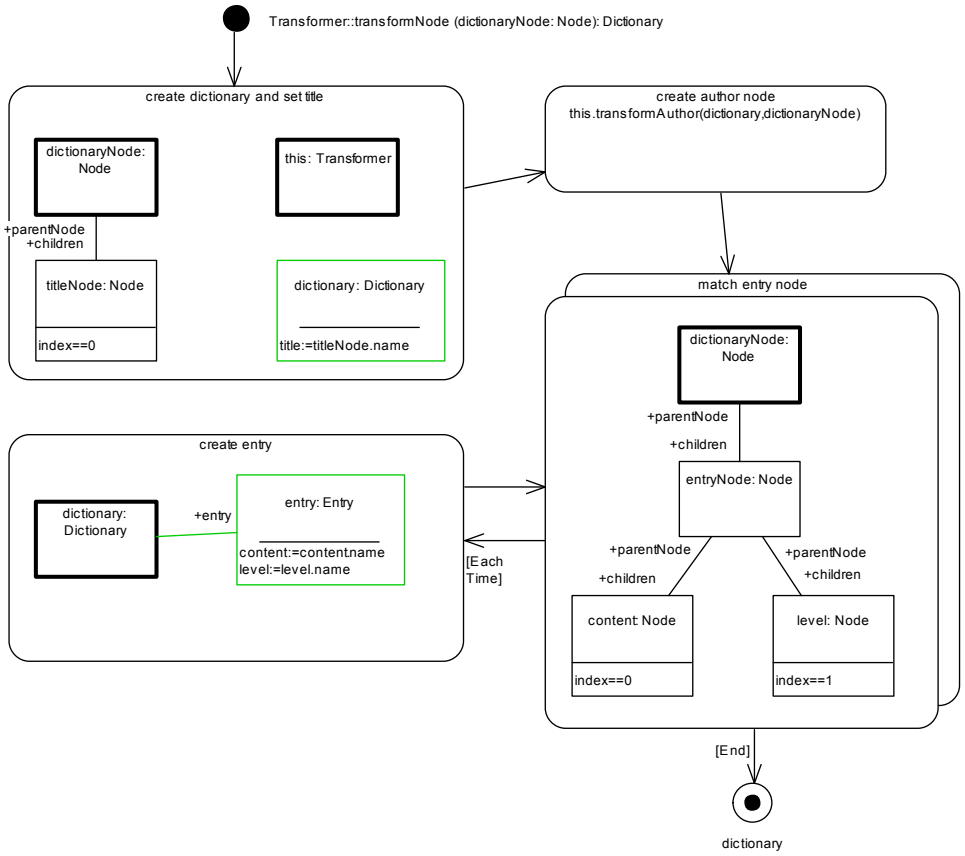


Figure 5.24: Creating dictionaries from dictionary nodes

- To wrap things up, create the last SDM, `transformAuthor(Node)` as depicted in Fig. 5.25. 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,

[13] Note that multiple arguments for a MethodCallExpression *cannot* be entered on a single line and separated, e.g. with a "," but rather have to be chosen in the drop-down menu and entered separately, pressing `Save` each time.

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?

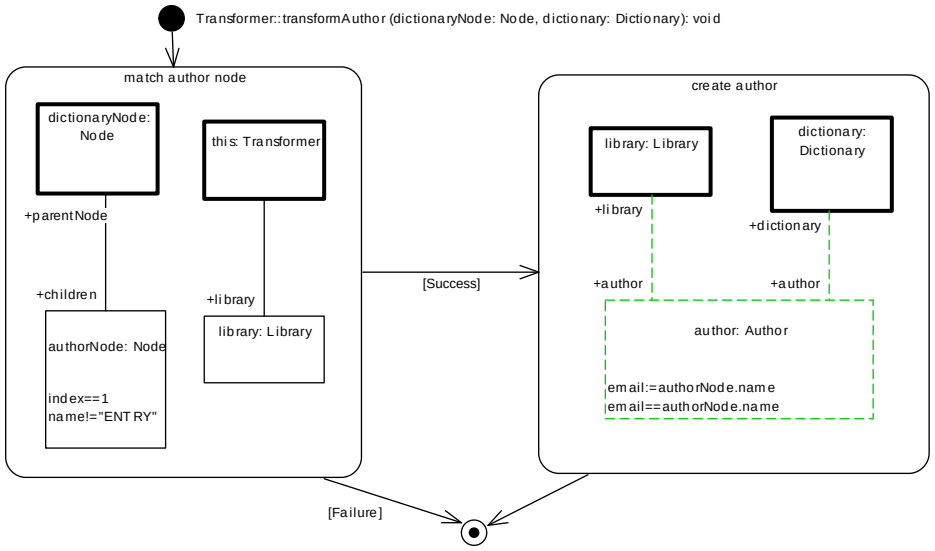


Figure 5.25: Handling Authors

- As a final step, open `MocaMain.java` (Fig. 5.17) and edit lines 32 – 36 as follows:

```
// Perform tree-to-model
Transformer transformer = DictionaryCodeAdapterFactory.
  eINSTANCE.createTransformer();
transformer.transformFolders(tree);

// Save library models
for (Library library : transformer.getTransformedLibraries())
  eMoflonEMFUtil.saveModel(library,
    "instances/" + library.getName() + ".xmi");
```

To see the effects of running `MocaMain.java`, refresh (`F5`) the Eclipse project to see the newly created `myLibrary.xmi` (Fig. 5.26). Open

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

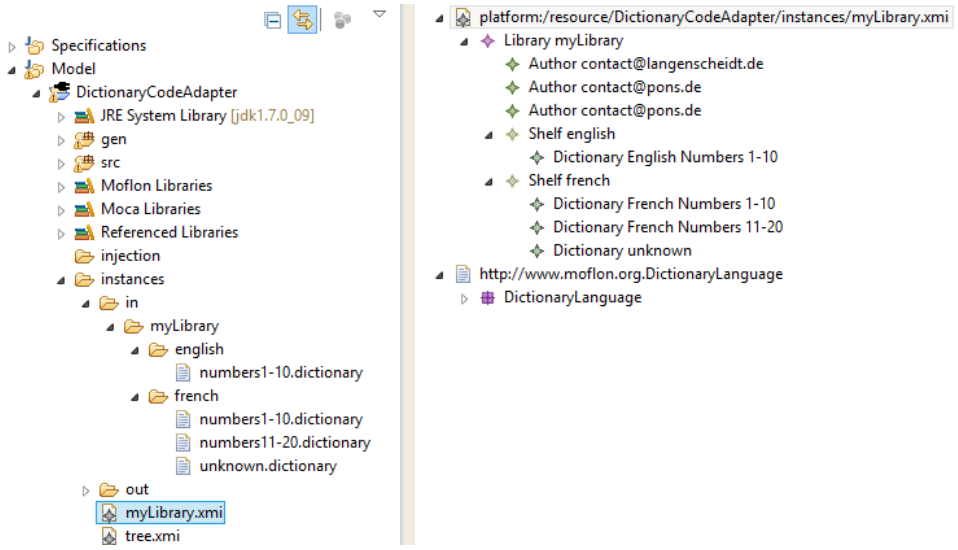


Figure 5.26: Created library model

5.4 Model-to-Tree transformation with SDMs

Do you remember that our unparsed files are still all empty (Fig. 5.19)? Well it's time to fix that - we need a *model-to-text* transformation to convert library instances to a folder structure with dictionary files in our DSL. Just like the *text-to-model* transformation, we shall take the same approach of breaking the job into two simpler steps: a *model-to-tree* transformation with SDMs (discussed in this chapter) and a *tree-to-text* transformations using ANTLR and a set of templates (discussed in Chapter 5.5). To remind you of the big picture, take a look at Fig. 5.1 and try to figure out where we are right now.

The following SDMs implement the model-to-tree transformation and transform a set of libraries to a single folder with a subfolder per library:

- `transformLibraries() :Folder` (Fig. 5.27) iterates through all libraries, shelves and dictionaries, creating the appropriate folder structure in the process. The actual transformation of each dictionary to a file is delegated using a binding expression (a method-call expression).

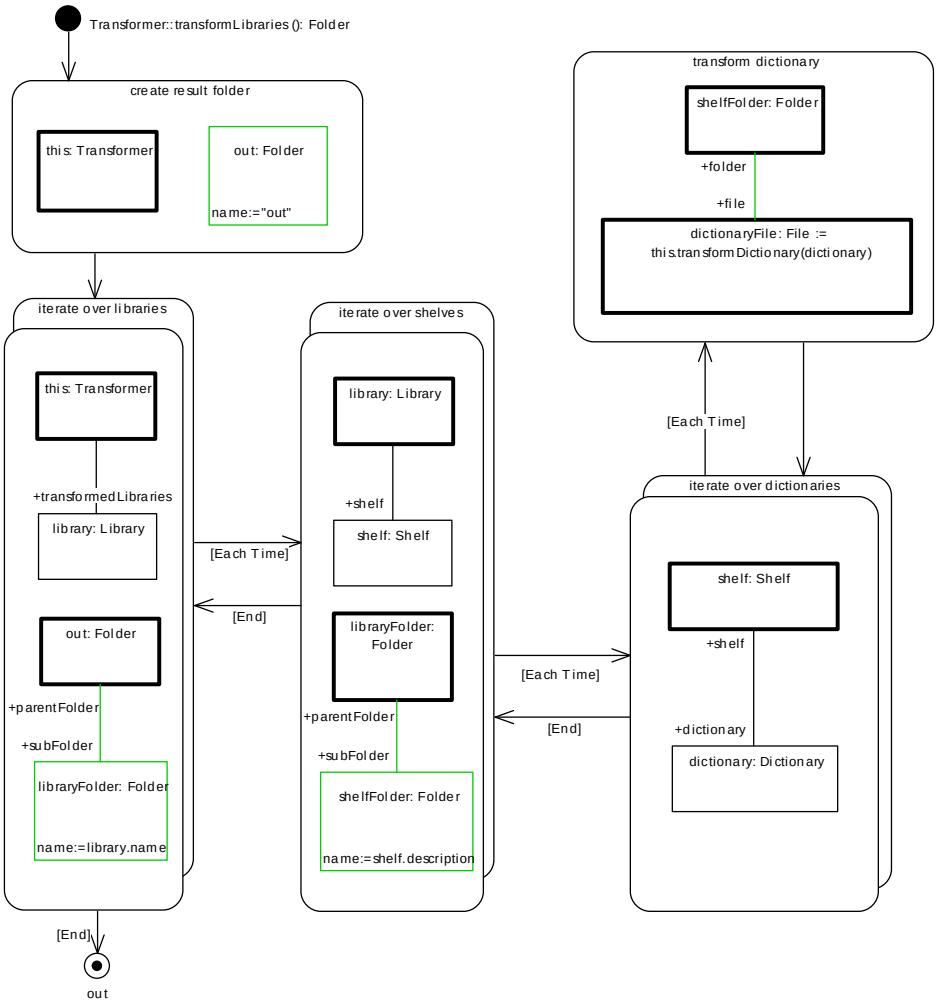


Figure 5.27: Iterate over all libraries, shelves and dictionaries

- `transformDictionary(Dictionary) :File` (Fig. 5.28)[14] does the actual work of transforming a dictionary to a file, handling authors as an optional node in the file subtree.

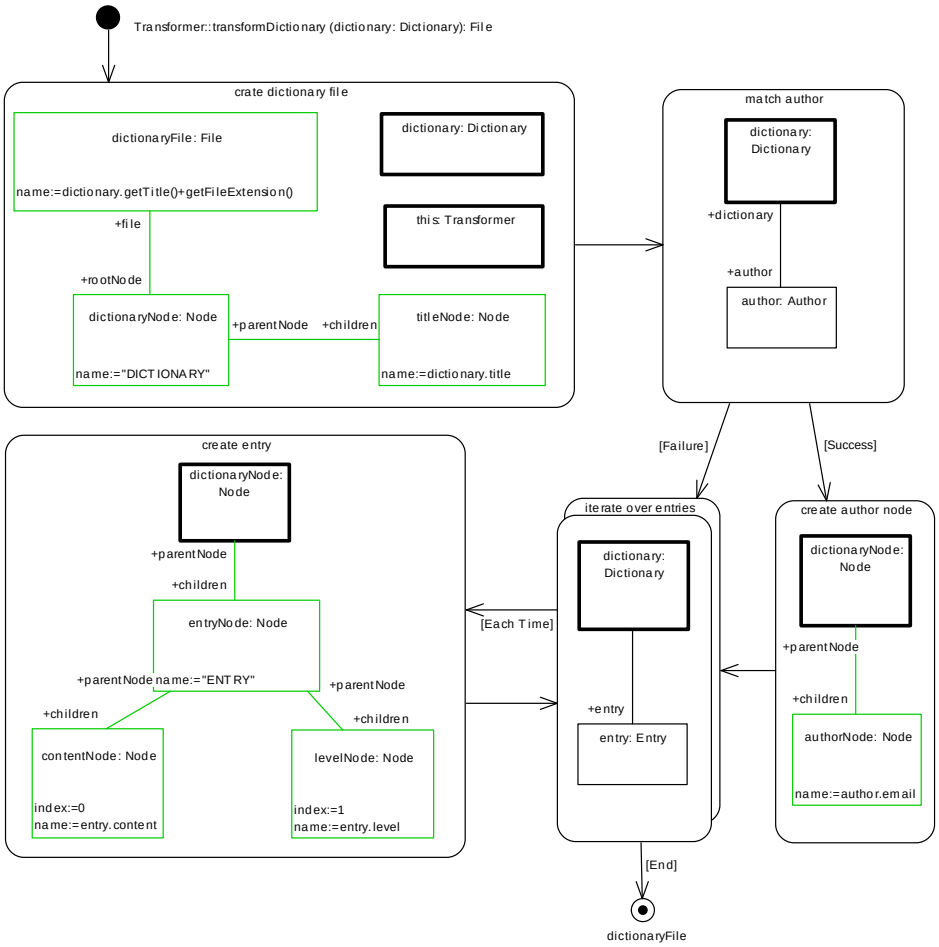


Figure 5.28: Create a file from a dictionary with an optional author

- To invoke the model-to-tree transformation, open `MocaMain.java` (Fig. 5.17) and edit lines 38-39 as follows.

```
// Perform model-to-tree transformation
transformer.setFileExtension(".dictionary");
Folder out = transformer.transformLibraries();
```

[14] Please note that the right-hand side of the attribute assignment in `dictionaryFile` is a literal expression.

The tree `out` can of course be persisted to file using `eMoflonUtil.saveModel` if you wish to inspect it before continuing. If you did everything right, it should resemble Fig. 5.20 very closely.

5.5 Tree-to-Text transformation with SDMs

In this last step, we are going to transform our tree to an actual folder structure with files containing text. One of the coolest things about ANTLR is that the same parser technology can be used to *unparse* a tree. Analogously to parsing text with a lexer and parser grammar to produce a tree, a tree can be unparsed to text using a *tree grammar* and templates.

- Open `src/org.moflon.moca.dictionary.unparser/DictionaryTreeGrammar.g` (Fig. 5.12), and edit the contents of the file as depicted in Fig. 5.29. A tree grammar is actually very similar to standard EBNF and consists of a set of rules (`main`, `entry`) that match a tree fragment (instead of a text fragment) and evaluate a template (instead of building up a tree). For further details concerning tree grammars we refer to [9] and the ANTLR website `www.antlr.org`.

```
1  tree grammar DictionaryTreeGrammar;
2
3  options {
7
8  // Tokens used internally by Moca
9  // ID: ('a'..'z' | 'A'..'Z')+;
10 // STRING: ( ID | ('0'..'9') )+;
11 // ATRIBUTE: Used as an imaginary token for coding attributes in XML files (ATTRIBUTE name=ID value=STRING)
12
13 tokens {
14   ID;
15   STRING;
16   ATTRIBUTE;
17 }
18
19 @header {
20 package org.moflon.moca;
21 }
22 // tree grammar rules:
23
24 main:  ^('DICTIONARY' name=STRING author=STRING? entries+=entry+)
25         -> dictionary(name={$name}, author={$author}, entries={$entries});
26
27 entry: ^('ENTRY' entryLabel=STRING level=STRING)
28         -> entry(entry={$entryLabel}, level={$level});
```

Figure 5.29: Tree Grammar for the dictionary DSL

- In the generated `src/org.moflon.moca.dictionary.unparser/DictionaryUnparserAdapter.java` (Fig. 5.12) a method for retrieving a group of templates has to be implemented. Figure 5.30 depicts the generated version showing how to use either a folder containing different template files, or a single file that contains all templates. The

latter is better for numerous smaller templates, while the former makes sense when the templates contain a lot of static text.

```
@Override
protected StringTemplateGroup getStringTemplateGroup() throws FileNotFoundException
{
  //TODO provide StringTemplateGroup ...
  // ... from folder "dictionary" containing .st files
  // return new StringTemplateGroup("dictionary", "templates/dictionary");
  // ... from group file Dictionary.stg
  // return new StringTemplateGroup(new FileReader(new File("./templates/Dictionary.stg")));
  throw new UnsupportedOperationException("Creation of StringTemplateGroup not implemented yet ...");
}
```

Figure 5.30: Unimplemented method getStringTemplateGroup

For our small example a single file with all the templates is ideal, so uncomment the option for a *group file*. Your unparser adapter should now closely resemble Fig. 5.31.

```
13 public class DictionaryUnparserAdapter extends TemplateUnparserImpl
14 {
15
16   @Override
17   public boolean canUnparseFile(String fileName)
18   {
19     return fileName.endsWith(".dictionary");
20   }
21
22   @Override
23   protected String callMainRule(CommonTreeNodeStream tree, StringTemplateGroup templates) throws RecognitionException
24   {
25     DictionaryTreeGrammar dictionaryTreeGrammar = new DictionaryTreeGrammar(tree);
26     dictionaryTreeGrammar.setTemplateLib(templates);
27     StringTemplate st = dictionaryTreeGrammar.main().st;
28     if (st==null) {
29       return "";
30     }
31     else {
32       return st.toString();
33     }
34   }
35
36   @Override
37   protected StringTemplateGroup getStringTemplateGroup() throws FileNotFoundException
38   {
39     return new StringTemplateGroup(new FileReader(new File("./templates/Dictionary.stg")));
40   }
41
42   @Override
43   protected String[] getTokenNames()
44   {
45     return DictionaryTreeGrammar.tokenNames;
46   }
47 }
```

Figure 5.31: Unparser Adapter

- The next step is to create the referenced template group file `Dictionary.stg` in `./templates`. In this file, enter the contents depicted in Fig. 5.32. ANTLR uses a template language called *StringTemplate*. StringTemplate is a very simple template language with few constructs and almost no constructs for complicated logic. This might sound like a disadvantage but it actually prevents you from *programming* in the templates and keeps them simple, readable and easily replaceable. For further details about StringTemplate we refer to the website

`www.stringtemplate.org`, a paper that argues for its simplicity [11], and books on usage and features in combination with ANTLR [9, 10]. StringTemplate is pretty intuitive, take a good look at Fig. 5.32 and pay especially attention to how the templates are invoked in the tree grammar (Fig. 5.29). You can easily make changes in the templates and see how it affects the generated files.

```
1  group dictionary;
2
3  dictionary(name, author, entries) ::= <<
4  title: "<name>"
5  <if(author)>email: "<author>"<endif>
6  {
7    <entries; separator="\n">
8  }
9  >>
10
11 entry(entry, level) ::= <<
12 "<entry>", <level>
13 >>
```

Figure 5.32: Templates for the Dictionary DSL

- To complete the model-to-text transformation, open `MocaMain.java` and make sure it closely resembles Fig. 5.33 which shows the complete final version. Note especially line 30 that is now uncommented to register our unparser adapter and lines 52-53 that call the framework method *unparse*, which creates a corresponding folder structure for a MocaTree instance and delegates code generation for each file using the registered unparsers.

  ```
  // Perform tree-to-text
  codeAdapter.unparse("instances", out);
  ```

  This way, different unparsers can be registered for different file types (defined by the unparser adapter's `canUnparseFile` method) and a whole folder structure containing different files can be generated. Each parser can either use a single template group file like in our case, or a folder containing separated template files.

  If you have done everything correctly, you should be able to run `MocaMain` and obtain similar results as depicted in Fig. 5.34. Note the created model `myLibrary.xmi` and the `.dictionary` files in `/out` that should now contain appropriate text in our DSL. As a final test you can manipulate the model directly (e.g., create a new dictionary or shelf, add/delete an author) and adjust `MocaMain` appropriately so only the model-to-text transformation is performed. Now compare the generated dictionaries to see if your changes are reflected correctly.

```java
package org.moflon.moca;

import java.io.File;

public class MocaMain
{

   private static CodeAdapter codeAdapter;

   public static void main(String[] args)
   {
      BasicConfigurator.configure();

      // Register parsers and unparsers
      codeAdapter = MocaFactory.eINSTANCE.createCodeAdapter();
      codeAdapter.getParser().add(new DictionaryParserAdapter());
      codeAdapter.getUnparser().add(new DictionaryUnparserAdapter());

      // Perform text-to-tree
      Folder tree = codeAdapter.parse(new File("instances/in/"));

      // Save tree to file
      eMoflonEMFUtil.saveModel(MocaTreePackage.eINSTANCE, tree, "instances/tree.xmi");

      // Perform tree-to-model
      Transformer transformer = DictionaryCodeAdapterFactory.eINSTANCE.createTransformer();
      transformer.transformFolders(tree);

      // save library models
      for (Library library : transformer.getTransformedLibraries())
      {
         eMoflonEMFUtil.saveModel(MocaTreePackage.eINSTANCE, library, "instances/" + library.getName() + ".xmi");
      }

      // Perform model-to-tree
      transformer.setFileExtension(".dictionary");
      Folder out = transformer.transformLibraries();

      // Perform tree-to-text (using initial tree)
      codeAdapter.unparse("instances", out);
   }
}
```

Figure 5.33: Completed main method for Text-to-Model and Model-to-Text

Figure 5.34: Directory instances after parsing and unparsing.

Chapter 6

Learning Box to Dictionary *and* Back Again with TGGs

If you are joining us directly in this chapter and are only interested in bidirectional model transformation with *Triple Graph Grammars* then welcome!

To get eMoflon up and running, however, you should at least work through Chapter 2 for the required installation and set-up. We try to assume as little as possible in the rest of this chapter and give appropriate references when we use terminology introduced in previous chapters.

Up until now in the tutorial, we have gotten to know what a learning box is, we have specified its *abstract syntax* and *static semantics* as a *metamodel* and its *dynamic semantics* with graph transformations (SDMs). If the previous sentence could just as well have been in Chinese[1] for you then please work through Chapter 3.

We have also established a simple textual *concrete syntax* for a dictionary and implemented *model-to-text* and *text-to-model* transformations with a merry mix of string grammars, graph transformations and templates (Chapter 5). As this was already in both directions (model⇔text) we have practically seen how to implement a *bidirectional* transformation as two unidirectional transformations using a unidirectional transformation language (SDMs). Although SDMs are crazily cool (don't you forget that!), this is rather unsatisfactory. If you take a careful and critical look at the forward (text-to-model) and the backward (model-to-text) transformations, you should be able to notice the following problems.

[1] Replace with Greek if you are chinese. If you are chinese but speak fluent Greek then we give up - you get the point anyway right?

Productivity: We have to implement two transformations that are actually quite similar... somehow. This does not really feel productive. Wouldn't it be nice to implement maybe the forward transformation and get the backward transformation for free? Or vice-versa? Or how about deriving forward *and* backward transformations from a *common* joint specification?

Maintainability: Another, maybe even more important point, is that two separate transformations become quite a pain to maintain and keep *consistent*. If the forward transformation is adjusted to produce a different target structure, the backward transformation must be updated appropriately and vice-versa. Again it would be great to get this for free or at least have some support.

Traceability Finally, one often needs to identify the reason why a certain object has been created during a transformation process. This increases the trust in the specified transformation and is essential for working with systems that may actually do some harm (e.g., automotive or medical systems). With two separate transformations, *traceability* would have to be supported manually and, once again, it would be nice to have this for free!

As we shall see in this chapter, Triple Graph Grammars (TGGs) are a *bidirectional* transformation language that solves these problems.

After a brief introduction to TGGs, we shall specify a bidirectional transformation between learning boxes and dictionaries. This transformation has already been motivated and explained in Chapter 3, so lets move on.

6.1 Triple Graph Grammars in a nutshell

Triple graph grammars [12, 13, 6] are a declarative, rule-based technique of specifying the simultaneous evolution of three connected graphs. If you have read the previous sentence at least three times and are still scratching your head wondering if you need a cup of coffee – welcome to the club[2] ;)

A TGG is basically just a bunch of rules. Each rule is actually quite similar to a *Story Pattern* (see Page 48) and describes with a precondition (LHS) and postcondition (RHS) how a graph structure is to be built-up. So what's the difference to SDMs? Well, a TGG rule always describes how a *graph triple* evolves and triples can be interpreted as consisting of a source component, a correspondence component and a target component. This means

[2] If not then congrats, stop reading, and go grab some real (academic) papers on TGGs.

that executing a sequence of TGG rules results in a source graph and a target graph, which are connected *only* via nodes in a correspondence graph. Note that the names "source" and "target" are arbitrarily chosen and do not imply a certain transformation direction. Naming the graphs "left" and "right", or "foo" and "bar" would also be fine, but some smart guys decided years ago to name them source and target and the convention has somehow stuck. Just remember, TGGs are *symmetric* in nature.

So far so good, but you should now be asking yourself the following question: What the heck does this have to do with bidirectional model transformation?! "*That*, Detective, is the right question. Program terminated."[3]

We believe there are two main keys to understanding TGGs:

(1) A TGG defines a consistency relation: Given a TGG (remember – just a bunch of rules), you can take a look at a source graph S and a target graph T and say if they are *consistent* with respect to the TGG. How? You simply check if a triple $S \leftarrow C \rightarrow T$ can be created using the rules of the TGG, which has the source graph S as its source component and the target graph T as its target component, and some correspondence graph C as its correspondence component [5]. If such a triple can be created then the given source and target graphs are consistent. Let's denote this as $S \Leftrightarrow_{TGG} T$. This consistency relation can already be used to check if a bidirectional transformation, i.e., a pair of forward (f) and backward (b) transformations are consistent (according to the TGG which can be viewed as a specification of how the transformations are to behave): $S \Leftrightarrow_{TGG} f(S)$ and $b(T) \Leftrightarrow_{TGG} T$.

(2) The consistency relation can be operationalized: Now comes the surprising (and extremely cool) part – a forward *and* backward rule can be derived automatically from every TGG rule [4, 5]! Just to make sure you got the point: The description of the simultaneous evolution of the source, correspondence and target graphs is *sufficient* to derive a forward transformation, i.e., given source, determine consistent correspondence and target graphs, and a backward transformation, i.e., given target, determine consistent correspondence and source graphs. If you can't imagine why then just accept it as magic for now. As these derived rules are actually executed to perform forward and backward transformation, they are called *operational* rules as opposed to the TGG rules, which are called *declarative* rules. The derivation process, therefore, is referred to as the *operationalization* of a TGG.

[3] Memorable quote from *I, Robot*.

Before we sum up and get our hands dirty on our concrete example, just a few extra points for the interested reader:

- A lot more can be automatically derived from the consistency relation including inverse rules to *undo* a step in a forward/backward transformation [7] and rules that simply check consistency of an existing triple of source, correspondence and target graphs.

- Why do we need the correspondence graph? Hmm... Thats a hard one ;) Well, first of all it can be viewed as explicit traceability links which are quite nice to have, i.e., you *see* which element relates to which after a forward/backward transformation. No guessing, no heuristics, no interpretation or ambivalence – the information *is* just there! This clearly fulfils the traceability wishes from the motivation at the beginning of the chapter.

  The second reason is a bit more subtle and difficult to explain without a concrete TGG: The point is mainly that the forward[4] transformation is *not* injective and cannot be inverted! So how can we provide a backward transformation?? If you paid attention in school, alarm bells should be going off in your head now – ding ding ding, a function can only be inverted if it is bijective which means injective and surjective[5], ding ding... Good point! When executing the forward transformation we sort of "cheat" and remember what target element was chosen if there was a choice (exactly this choice makes the transformation non-injective right?). In this way we *bidirectionalize* the transformation on-the-fly by creating correspondence links. Got it? Great! No? Then just remember: the correspondence model is cute and for some reasons also important. And if you loose it somehow – no worries, because the *same* TGG specification that has been used to derive your forward and backward transformations can also be used to reconstruct the correspondence model between two existing source and target models (at least to a certain degree).[6]

That was quite a lot at once so it might make sense to re-read these sections *after* working on the example. Enough theory! Grab your computer (if you're not hugging it already) and lets churn out some TGGs!

[4] Note that the TGGs are symmetric and forward/backward can be interchanged freely. As it is becoming a pain to always write forward/backward we shall just say forward as from now on.

[5] If you don't have an idea what injective/surjective/bijective means then ask your good friend Google.

[6] We refer to this type of operational rule as *link creation*.

6.2 Specifying a TGG schema

The first step on the way to a TGG is to model the correspondence component of your triples. The correspondence or link metamodel consists of *correspondence types* which connect classes from the source and target metamodels. In our case, our source metamodel is `LearningBoxLanguage` and target `DictionaryLanguage`. Remember that a *correspondence type* can be considered as a traceability link which relates the elements from the source and target components. Although the correspondence metamodel is actually just a normal metamodel, we use a slightly different concrete syntax to present it and adhere to certain naming conventions. The metamodel triple consisting of (relevant parts of) the source metamodel, the correspondence metamodel and (relevant parts of) the target metamodel is called a *TGG schema*.

TGG Schema

- If you've worked through the tutorial until now, you can import `DictionaryLanguage` into the EAP file containing `LearningBoxLanguage` or vice-versa. Important is that *both* metamodels are in the *same* EAP file so we can specify an integration with TGGs. How to transfer projects between EAP files is explained in Chapter 4.
- If you're just joining us for TGGs, or just don't want to go through the trouble of transferring the EA projects, simply import the prepared metamodel project "ForChapterOnTGG/LearningBox2Dictionary" (in the `eMoflonTutorial.zip` you downloaded with this tutorial) into your Eclipse workspace .
- Open `LearningBox2Dictionary.eap` in EA, and add a new package to your model root with `LearningBoxToDictionaryIntegration` as its name. Create a diagram for the new package and select `TGGSchema` as diagram type (Fig. 6.1). The diagram type indicates that the new package is a TGG Project.
- After choosing `TGGSchema` as diagram type, a new dialogue should pop up asking you for the source and target projects of your TGG project. Choose `LearningBoxLanguage` as source and `DictionaryLanguage` as target project and affirm with `OK` (Fig. 6.2).

The structure of your TGG project should now resemble Fig. 6.3. Please note that a subpackage `Rules` and an underlying diagram with the same name are also generated.

Now it's time to insert classes from our source and target projects into our TGG project and declare our first correspondence type between them. The

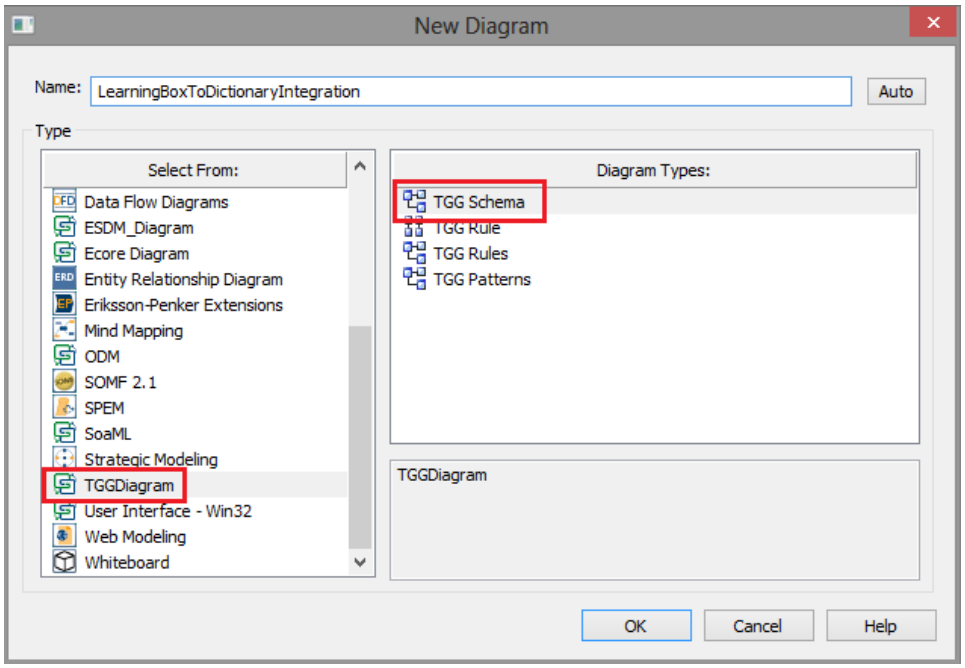


Figure 6.1: Choose `TGGSchema` as your diagram type

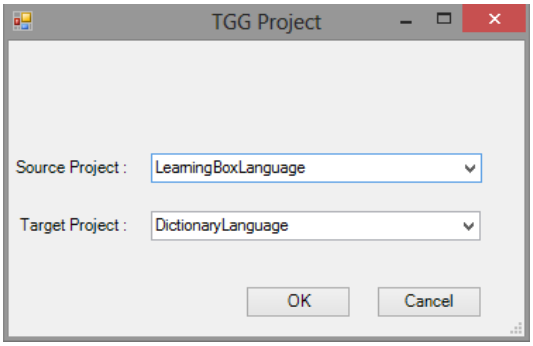


Figure 6.2: Select source and target projects for the TGG project

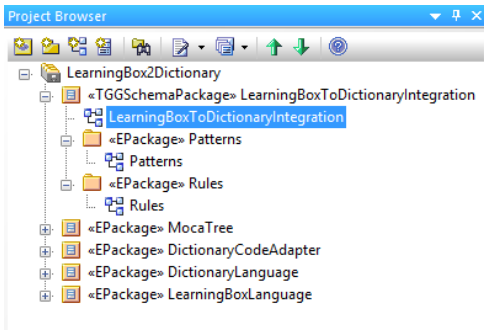


Figure 6.3: Initial structure of a new TGG project

classes `Box` and `Dictionary`, both at the top of the composite hierarchy of their respecitve languages, are to be related to each other:

- Drag & drop the class `Box` in `LearningBoxLanguage` from the project browser to the newly created TGG schema diagram `LearningBoxToDictionaryIntegration`. Ensure that the class is pasted `as a simple link` into the diagram as depicted in Fig. 6.4. If the dialogue in Fig. 6.4 doesn't show up and the element is pasted instead as an instance of the class, hold down `Ctrl` when dropping to invoke it.

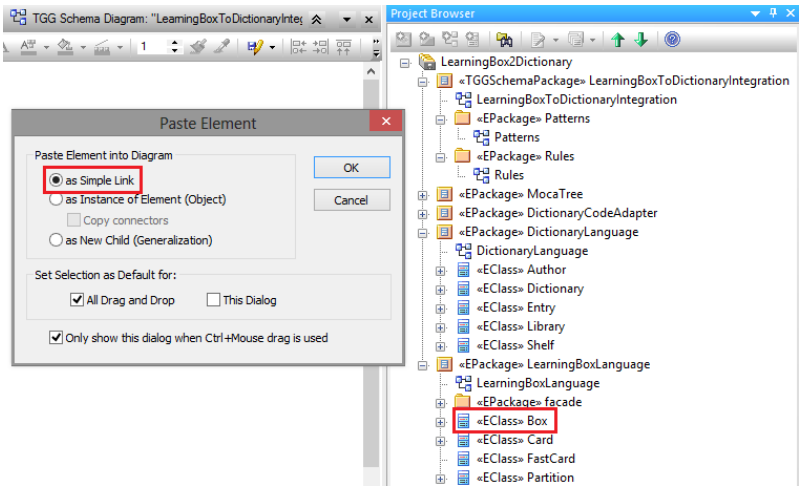


Figure 6.4: Drag & Drop `Box` as a simple link into the TGG schema

- In the same way, drag & drop the class `Dictionary` from `DictionaryLanguage` into the TGG schema.

With a class from both source and target projects, we can now create a correspondence type between them.

- Quick link from `Box` to `Dictionary` and select `Create TGG Correspondence Type` as depicted in Fig. 6.5.

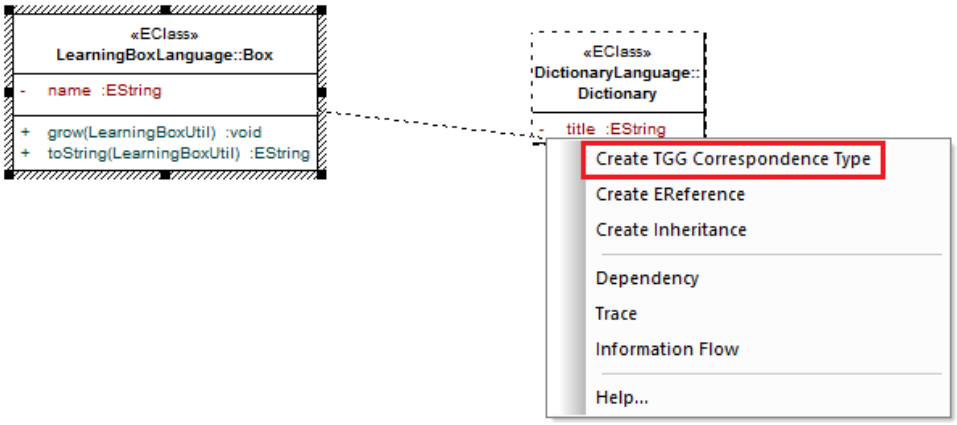


Figure 6.5: Creating a TGG correspondence type

A hexagon-shaped correspondence type with the name `BoxToDictionary` and references to the source and target elements should be generated. You can rename the correspondence type as you wish but please leave the references as they are (multiplicity and naming conventions are satisfied automatically).

To complete our TGG schema, declare a further correspondence type between `Card` and `Entry`. The complete TGG Schema is depicted in Fig. 6.6.

A TGG schema can be viewed as the metamodel triple to which all created triples must conform. In less technical lingo, it gives an abstract view on the relationship or *correspondence* between two metamodels or domains. Just by looking at the TGG schema, a domain expert should understand *why* the connected elements are related, irrespective of *how* the relationship is actually established by TGG rules (coming up next). Note that all other information in the source and target metamodels such as attributes, methods and references between classes are also visualized per default in the TGG schema – just in case you were wondering where all the attributes and references suddenly came from.

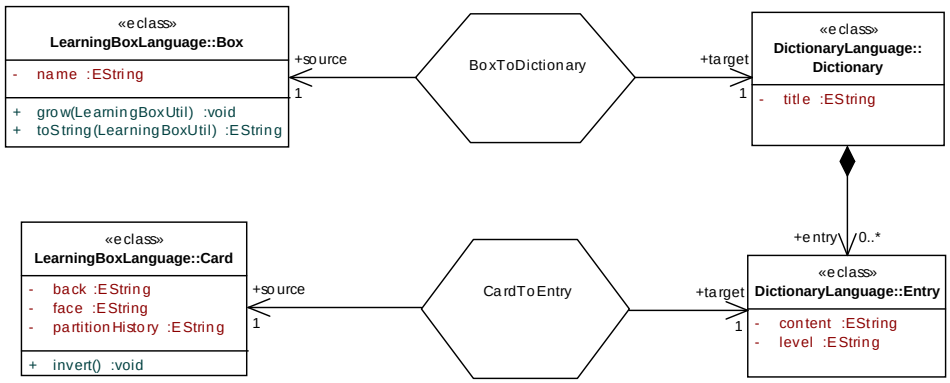


Figure 6.6: Complete TGG schema for our example

6.3 Specifying TGG rules

After declaring our correspondence types in the TGG schema, we can now specify a set of *TGG rules* to describe the simultaneous evolution of both source, correspondence and target models.

A TGG rule is quite similar to an SDM storypattern[7] and is also of the form (*precondition, postcondition*). In other words, we have to state:

- What pattern must be matched, i.e, under which conditions can the rule be applied (this is the precondition).
- The objects and links to be created when the rule is applied to a match (this is deduced from the postcondition).

Note that the rules of a TGG only describe the simultaneous *build-up* of the models and do not delete or modify any existing elements, i.e., TGG rules are *monotonic*. This might seem surprising at first and you might think this is a terrible restriction. The point is that the TGG should only specify a consistency relation and not directly the forward and backward transformations, which are derived automatically. It turns out that deletion is not necessary on this level to do this, but will of course be used at the right places in the generated transformations.

▶ In EA, open the diagram of the `Rules` package in our TGG project. This package was generated automatically when we first created our TGG project and contains all TGG rules.

[7] If you are not familiar with modelling SDMs in EA see Chapter 3.5

- Create a `Rule` via drag & drop from the TGG toolbox and set its name to `BoxToDictionaryRule` (Fig. 6.7).

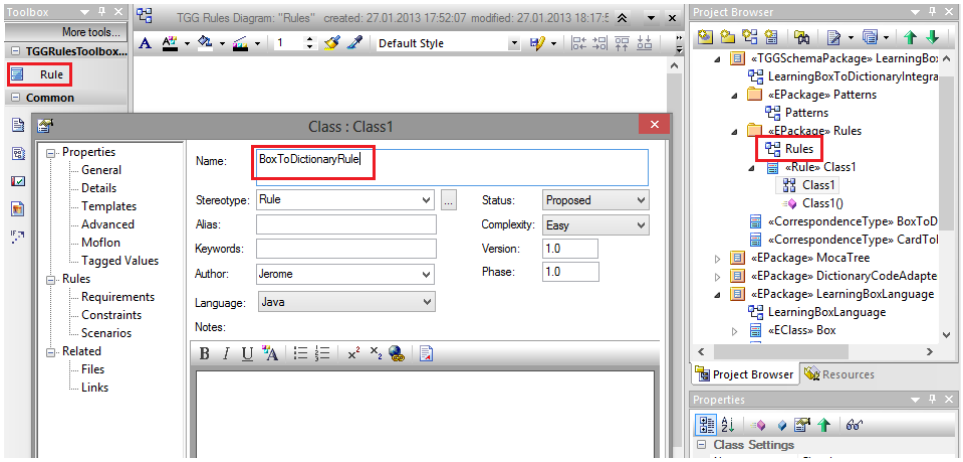


Figure 6.7: Creating a TGG rule

As depicted in Fig. 6.8, the newly created rule has a method and contains a diagram (indicated by two small linked circles at the bottom right).

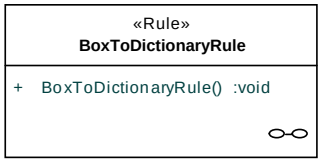


Figure 6.8: Our first TGG rule

- Double click `BoxToDictionaryRule` to open its diagram.
- Drag & drop a `Box` from the project browser, this time to create an instance of `Box`, and enter `box` as its name. Choose `Create` as binding operator (Fig. 6.9).
- In the same way, create `dictionary` as an instance of `Dictionary`.
- Also create `boxToDictionary` as an instance of the correspondence type `BoxToDictionary`, by using Quick-Link between `box` and `dictionary` and choosing "Create TGG Correspondence Link" (Fig. 6.10).

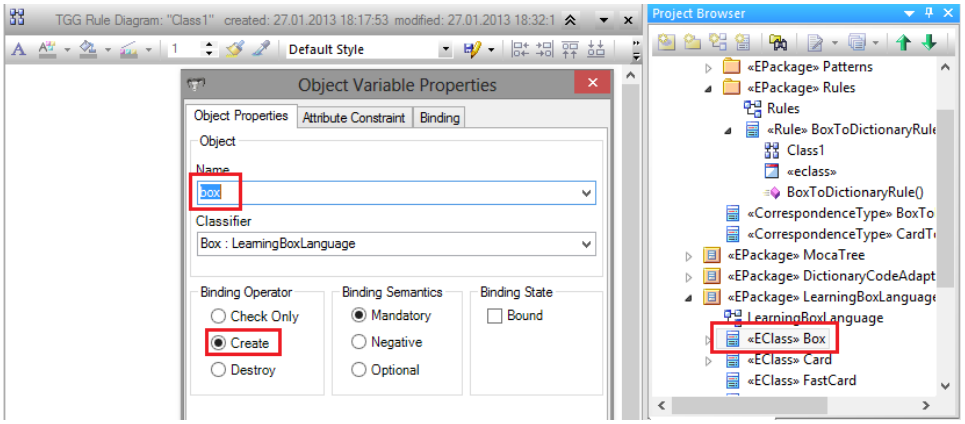


Figure 6.9: Creating an object variable in a TGG rule

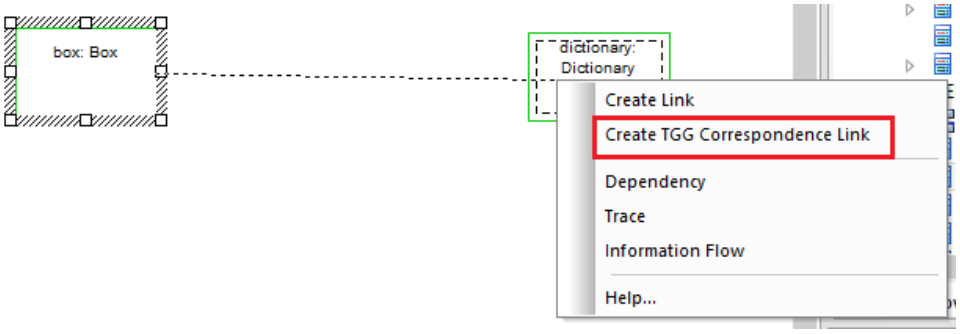


Figure 6.10: Creating a TGG correspondence link via Quick-Link

Our current rule creates a box, a dictionary, and an appropriate correspondence all at the same time.

Attribute Constraints

But how about the `name` of the box and the `title` of the dictionary? We use *attribute constraints* in TGG rules to provide a bidirectional and high-level solution for attribute manipulation. In this case we need a constraint which ensures that `box.name` and `dictionary.title` are set to the same value.

- To create such a constraint, drag & drop a *TGG Constraint* from the `TGGRuleTolboxPage` (Fig. 6.11).

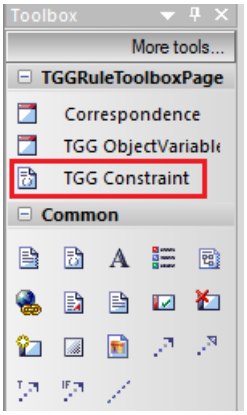


Figure 6.11: Constraint from the Toolbox in EA

- Double click the constraint element to open it, choose "eq" under "Constraints", enter the values depicted in Fig. 6.12 and add the constraint. Affirm with `OK`.
- Quick-link a `Dependency` from the constraint to the involved object variables, `box` and `dictionary`. The rule should now resemble Fig. 6.13.

To complete our first TGG rule, we still need to create the initial structure of a learning box. In contrast to the rather simple dictionary, where `Dictionary` is a direct container for `Entry` objects, we have to create a number of connected `Partitions` that hold the `Cards` in the learning box.

- Create three `Partition` object variables with all appropriate links, so that your TGG rule diagram closely resembles Fig. 6.14.

If you are in hurry, you may already proceed to Sect. 6.4 and transform a box to a dictionary and vice-versa. But please be aware that your specified

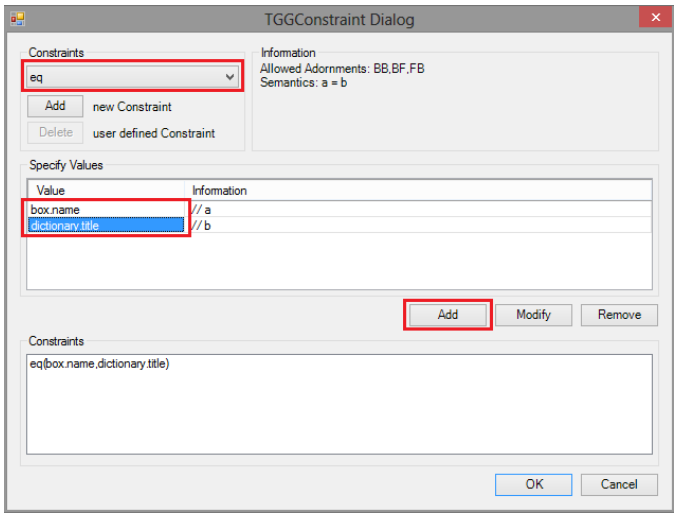


Figure 6.12: Creating a Constraint in EA

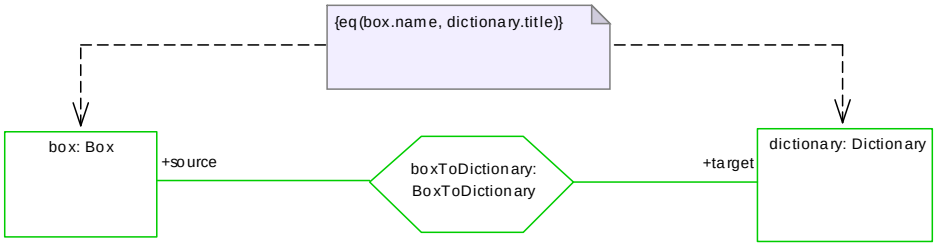


Figure 6.13: A TGG Rule with a Constraint

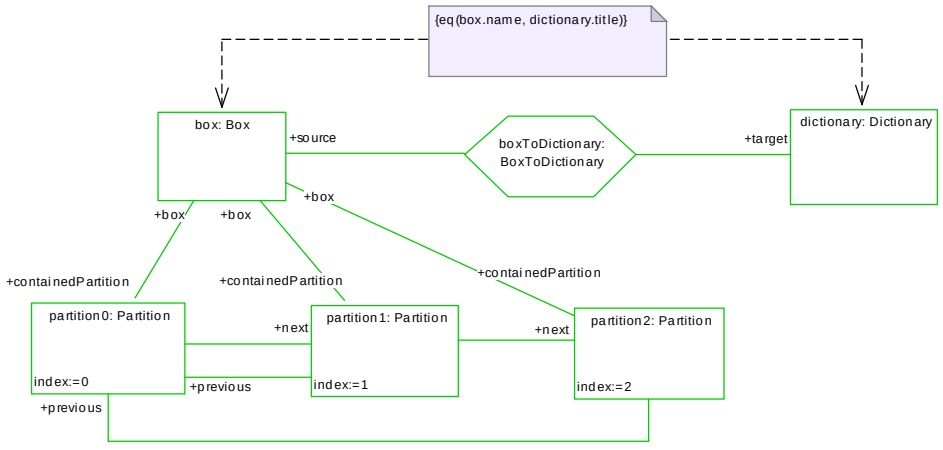


Figure 6.14: Complete TGG rule diagram for `BoxToDictionaryRule`

TGG (with just one rule) will only be able to cope with empty boxes and dictionaries. Handling additional elements (cards in the learning box and entries in the dictionary) requires a second rule.

To create the second rule, we can use a cool feature provided by the editor to derive new rules from existing rules:

- Select `box`, `boxToDictionary`, `dictionary` and `partition0` in `BoxToDictionaryRule`.
- Choose "Derive" in the "eMoflon TGG Functions" section of the add-in window as depicted in Fig. 6.15 (if you cannot see this window, make sure "Extensions → Add-In Window" is activated) and enter "CardToEntryRule" as the name in the dialog that pops up. This will derive a new TGG rule that already contains the chosen elements as context.

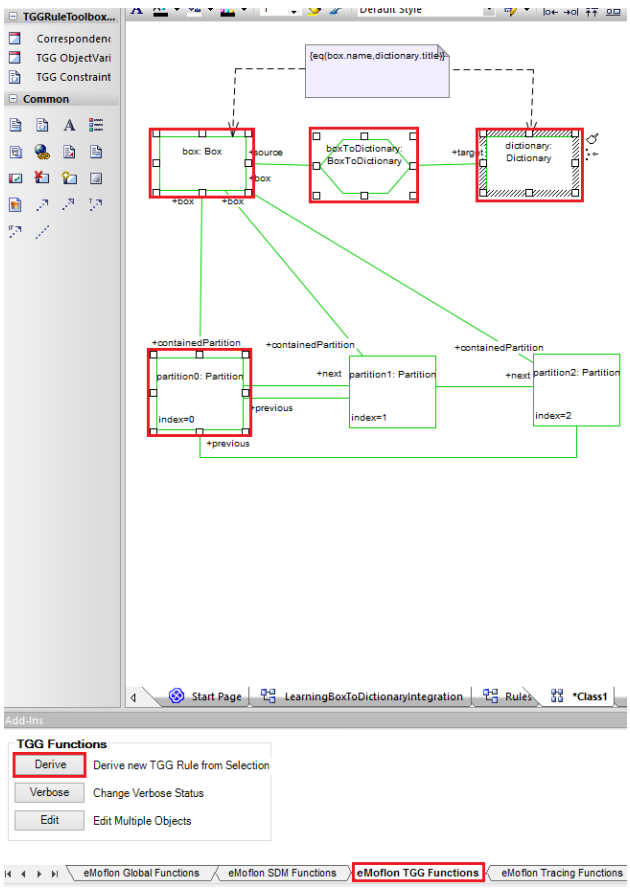


Figure 6.15: Derive from an existing TGG rule

- Add instances of `Card` and `Entry` to the rule and required links until the diagram closely resembles Fig. 6.16.

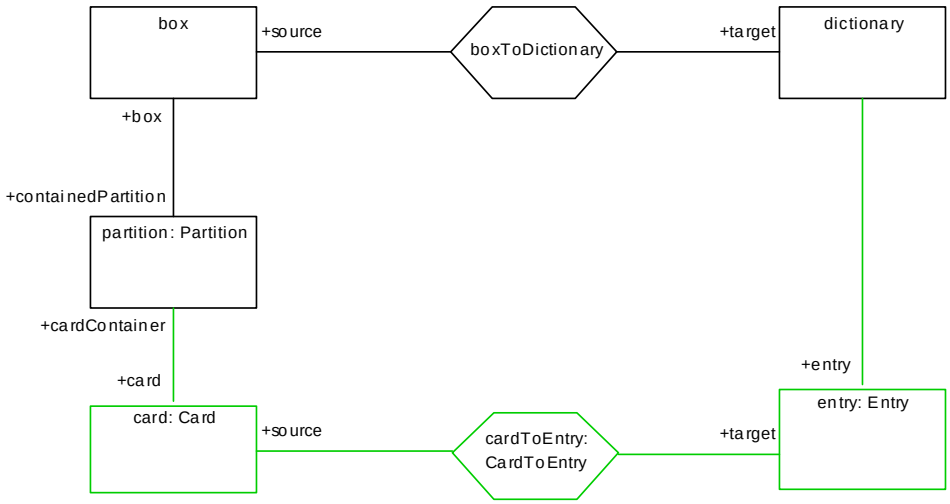


Figure 6.16: `CardToEntryRule` without attribute manipulation

As a final step, we now have to specify how attributes are to be handled via appropriate attribute constraints.

The `content` of an `Entry` in a `Dictionary` is to be of the form `<word> : <meaning>`, while the `face` of a `Card` must read `Question: <word>`, and the `back` of the Card `Answer: <meaning>`.

Using two predefined attribute constraint `addPrefix` and `concat`, we can specify this as a set of constraints:

- Add a new constraint to your diagram with the following three lines:

```
addPrefix("Question: ", word, card.face)
addPrefix("Answer: ", meaning, card.back)
concat(" : ", word, meaning, entry.content)
```

After connecting appropriately, your rule should now resemble Fig. 6.17.

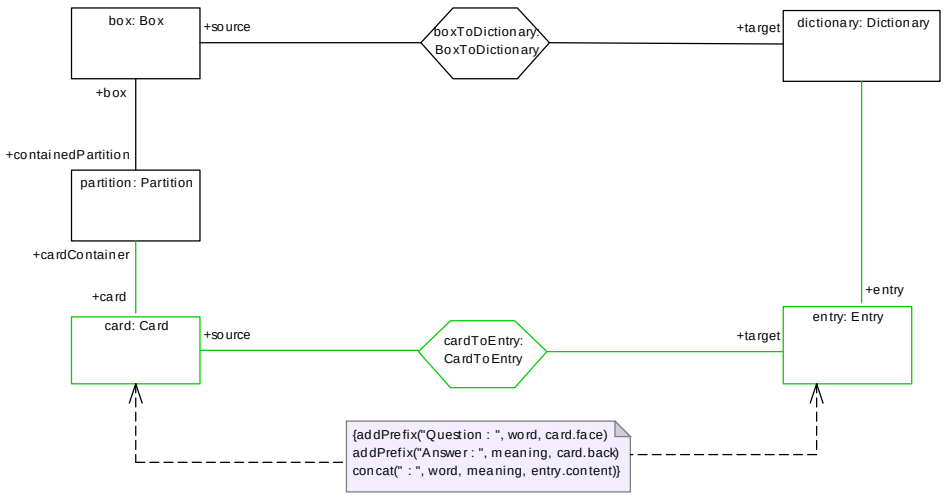


Figure 6.17: Attribute manipulation for `card` and `entry`

Finally, we have to specify how the partition, into which the newly created card is to be placed, must be chosen. We shall implement the following simple rule: a card in a partition with index 0/1/2 corresponds to an `Entry` of level beginner/advanced/master. This time, we must define our very own attribute constraint to handle this mapping. For the moment, we are just going to declare and use the attribute constraint, which will be implemented later in Java.

- Add a further constraint to your diagram, but this time do not choose a predefined constraint but click "Add" below the dropdown menu and enter the values given in Fig. 6.18.

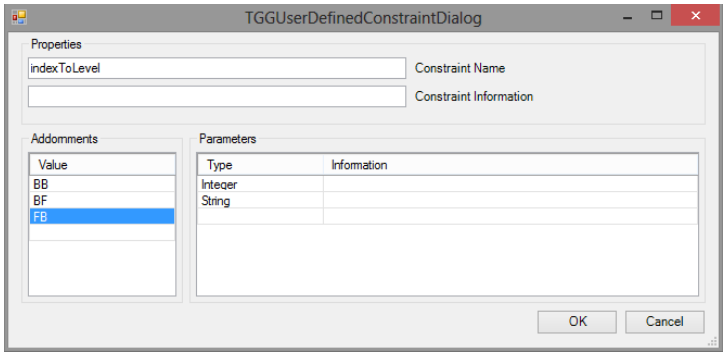


Figure 6.18: Create a user defined constraint.

▶ After saving this new constraint, choose it and enter the given values:

```
indexToLevel(partition.index, entry.level)
```

After defining the dependencies of the constraint, your complete TGG rule should resemble Fig. 6.19.

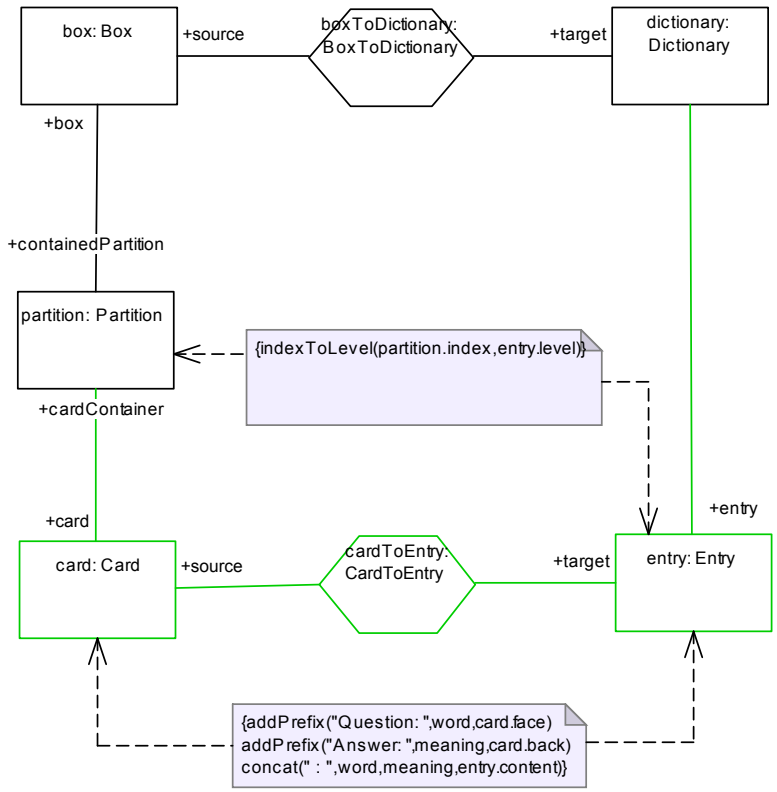


Figure 6.19: `CardToEntryRule` with complete attribute manipulation

Just like the patterns describing *structural* correspondence, attribute constraints can be automatically *operationalized* as required for the concrete transformations (forward, backward). Even more interesting, a set of constraints might have to be ordered a bit differently depending on the direction of the transformation, and some constraints might have to be checked for already set attributes, while others must set values appropriately to fulfill the constraint.

For built-in or *library* constraints such as *eq*, *addPrefix* and *concat*, you do not need to worry about these details and can just express what should hold – everything else is handled automatically.

In many cases, however, a constraint might be very problem-specific, such as our *indexToLevel* constraint, and there might not be any fitting combination of library constraints to express the consistency condition.

In such a case, the new attribute constraint must be declared before its use.

The list of *adornments* in the declaration specifies the cases for which the constraint can be operationalized. Each adornment consists of a `B` for bound or an `F` for free, for each argument of the constraint. This is much simpler than its sounds so lets take a look at our example:

BB means that the `partition.index` and `entry.level` can both be *bound*, i.e., already have assigned values. In this case, the *operation* (the operationalized constraint) must check if the assigned values are correct.

BF means that `partition.index` is *bound* and `entry.level` is *free*, i.e., the operation must determine and assign the correct value to `entry.level` using `partition.index`.

FB means that `partition.index` is *free* and `entry.level` is *bound*, i.e., the operation must determine and assign the correct value to `partition.index` using `entry.level`.

Note that we decide not to support **FF** as we would have to generate a consistent pair of index and level. Although this is possible and might even make sense for some applications, in our case it does not (the pair is not unique... which pair should we take?).

At compile time, the set of constraints (also called *Constraint Satisfaction Problem* (CSP)) for every TGG rule is "solved" for each case (forward, backward) by operationalizing all constraints and determining a feasible (compatible to the declared adornments of each constraint) sequence in which the operations can be executed. An exception is thrown if this is not possible.

6.4 TGGs in action

In this section, we shall export our TGG, implement our new constraint, and get our integration running!

- Export your metamodels and TGG by choosing "`Extensions/MOFLON::-Ecore Addin/Export all to Workspace`" as usual in EA and refres your Eclipse Metamodel project to trigger code generation.

If you have done everything right, code generation and compilation should terminate without any error and the structure of the `gen` folder in `LearningBoxToDictionaryIntegration` should resemble Fig. 6.20.

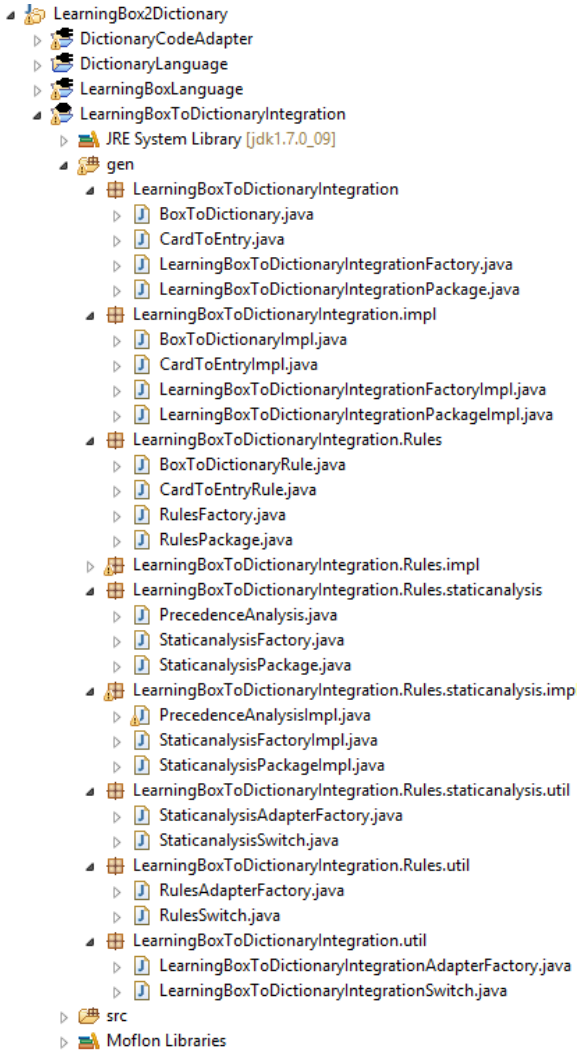


Figure 6.20: Integration project after code generation

- To implement our constraint `IndexToLevel`, locate and open the file `csp/IndexToLevel.java` in the `src` folder of `LearningBoxToDictionaryIntegration`.

- Insert the code provided in Fig. 6.21

```
package csp.constraints;

import java.util.Arrays;
import java.util.List;

import TGGLanguage.csp.Variable;
import TGGLanguage.csp.impl.ConstraintImpl;

public class IndexToLevel extends ConstraintImpl {

  private List<String> levels = Arrays.asList(new String[] { "beginner",
      "advanced", "master" });


  public void solve(Variable<Integer> var_0, Variable<String> var_1){
    String bindingStates = getBindingStates(var_0, var_1);

    switch(bindingStates){
    case "BB":
      int indexBB = var_0.getValue().intValue();
      String level = var_1.getValue();
      setSatisfied(levels.get(indexBB).equals(level));
      break;
    case "BF":
     int indexFB = var_0.getValue().intValue();
      if (indexFB < 0 || indexFB > 2) {
        setSatisfied(false);
      } else {
        var_1.setValue(levels.get(indexFB));
        var_1.setBound(true);
        setSatisfied(true);
      }
      break;
    case "FB":
      String levelBF = var_1.getValue();
      int indexBF = levels.indexOf(levelBF);
      if (indexBF == -1) {
        setSatisfied(false);
      } else {
        var_0.setValue(indexBF);
        var_0.setBound(true);
        setSatisfied(true);
      }
      break;
    }

  }
}
```

Figure 6.21: Implementation of our attribute constraint

In the next few steps, we shall create an instance model[8] of one of our languages and transform it to an instance model of the other language according to our TGG, i.e., perform a forward and backward transformation. As dictionaries are of a much simple structure, let's start with a backward transformation, i.e., dictionary to learning box:

- Open `DictionaryLanguage/model/DictionaryLanguage.ecore` and create a dynamic instance of `Dictionary`, and save it under `LearningBoxToDictionaryIntegration/instances/target.xmi` (Fig. 6.22).

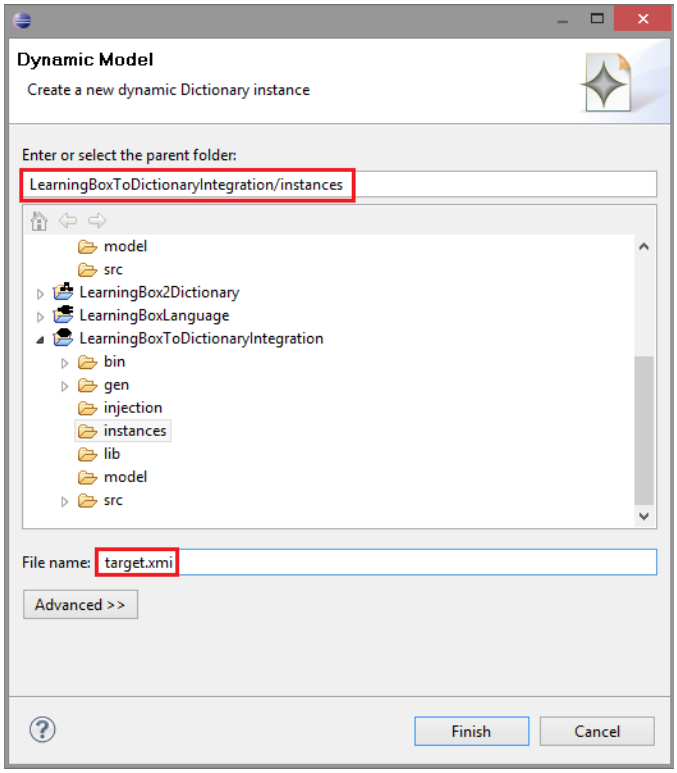


Figure 6.22: Create a dynamic instance of `Dictionary`

- Open `target.xmi` and set `Dictionary.title` to `English Numbers`.
- Create two `Entry` objects as children of `Dictionary`: the first entry with `one : eins` as content and `beginner` as level, the second with `eleven : elf` as content and `advanced` as level (Fig. 6.23).
- Run the automatically created main class `TGGMain` in `LearningBoxToDictionaryIntegration/src` and refresh the folder `LearningBoxToDictionaryIntegration/instances`.

[8]Refer to Chapter 3.4 if you do not know how to do this.

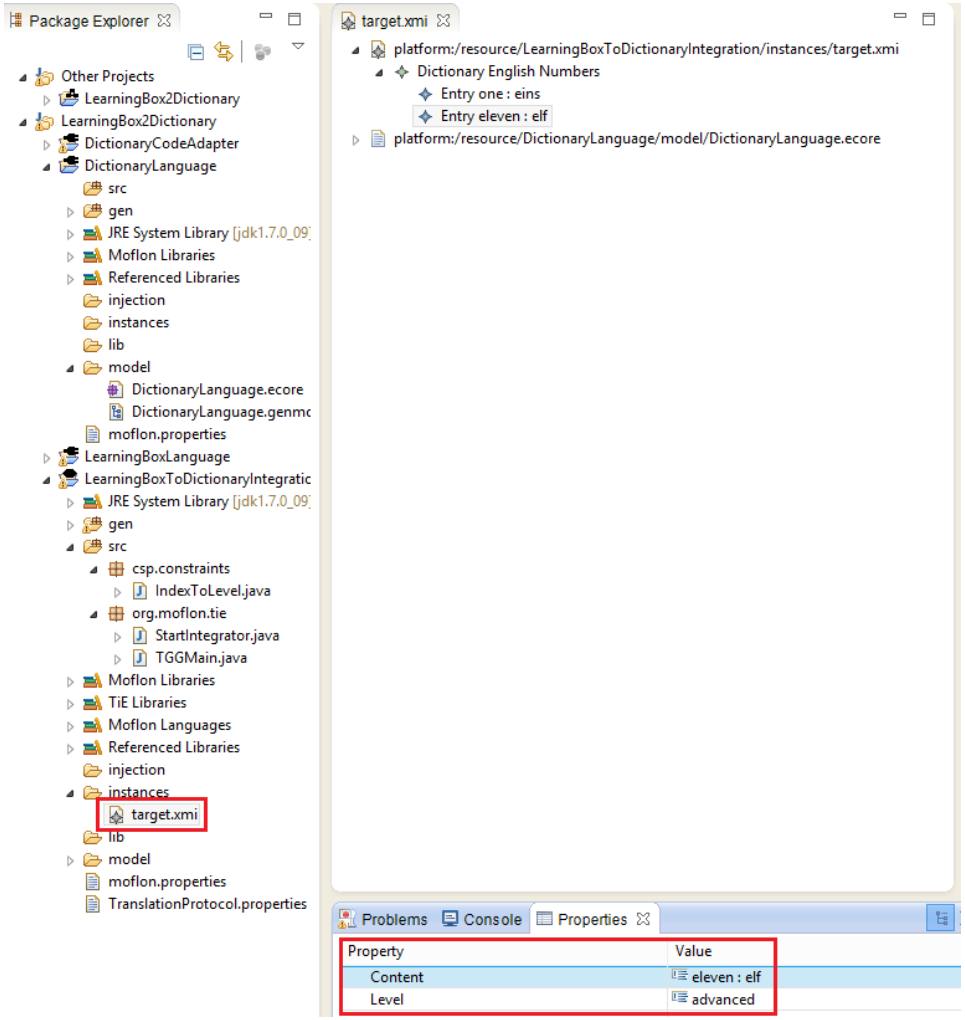


Figure 6.23: Contents of the dictionary

Note the error message "Unable to load instances/source.xmi, instances/-source.xmi does not exist." in the console. This is because no source file was found for the forward transformation. We'll fix that in a moment.

As you can see, our `Dictionary` has been backward translated to a `Box` with the same name (`English Numbers`), containing three `Partitions`, as specified in `BoxToDictionaryRule`. The two `Entry` objects have also been translated to `Card` objects as specified in `CartToEntryRule`. Also note that the `face` and the `back` of the `Card`s are consistent with the `content` of the corresponding `Entry`, e.g. `card.face = ''Question : one''` and `card.back = ''Answer : eins''`. The indices of the partitions containing the cards are also consistent with the level of the entry, i.e., `0` for `beginner`, and `1` for `advanced` (do not let the names of the partitions in the view confuse you: they are labelled using their size (all 0), and not their index, double-click the partitions to see the values of *all* their attributes in the property window).

Congratulations! You have successfully performed your first *backward* transformation from `DictionaryLanguage` (target model) to `LearningBoxLanguage` (source model) using TGGs!

To show that the transformation is actually bidirectional, lets edit the created source model and transform it *forward* to a new target model:

- Make a copy of `target.xmi_BWD.xmi` (the result of the backward transformation) and rename it to `source.xmi`.
- Open `source.xmi` and create some new `Card` objects in the `Partitions` (e.g., create a new `Card` with `Card.face = ''Question : two''`, `Card.back = ''Answer : zwei''` in `Partition 0`).
- Run the `TGGMain.java` again and inspect the created target model `source.xmi_FWD.xmi` (result of the forward transformation).

To end this chapter on TGGs, lets take a look at a *visualization* of the created triple of models:

- Right-click on `corr_BWD.xmi` and choose "eMoflon → Start Integrator", which opens the window depicted in Fig. 6.24.
- Drag and drop `protocol_BWD.xmi` into the integrator window. You will now see the controls explained in the lower part of the window (Fig. 6.25). Note that this chapter will only cover the very basic commands so please refer to Appendix A.7 for further information (e.g. about setting breakpoints).

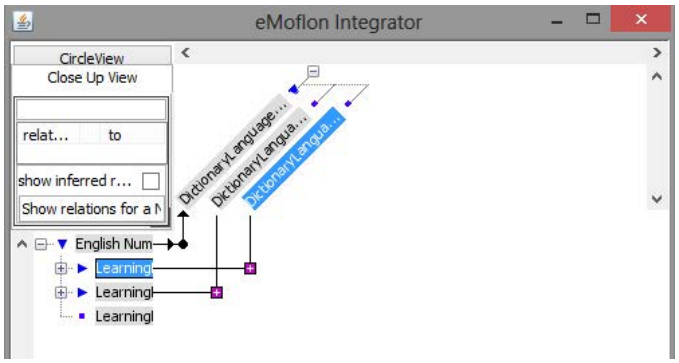


Figure 6.24: Default view of the integrator

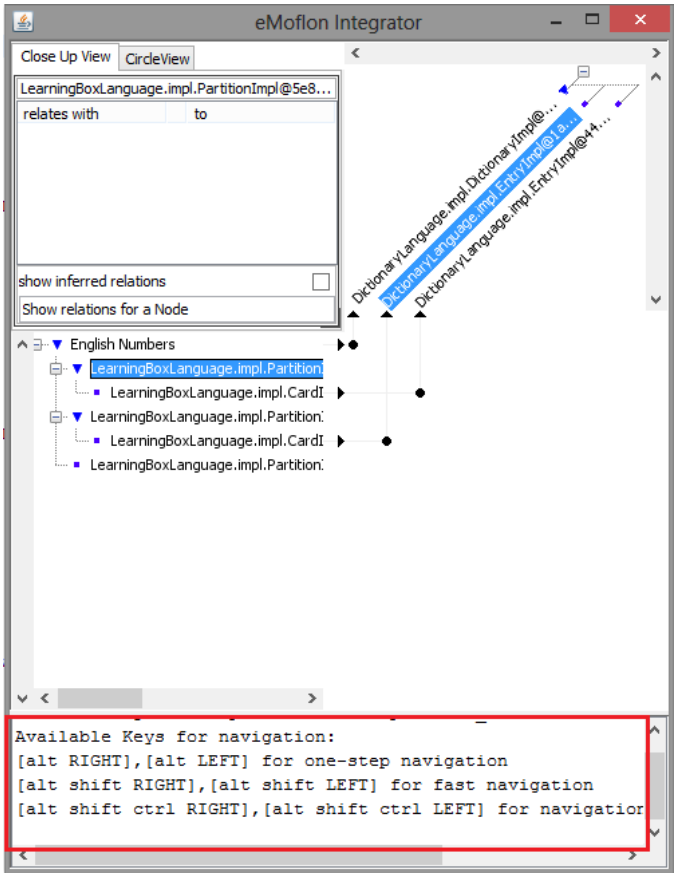


Figure 6.25: Integrator after protocol insertion.

- The integrator works as an "offline" debugger which works on the protocol (trace) of the transformation. You can use `Alt+Right` to navigate forwards through the transformation process and `Alt+Left` to go backwards. When you step through the transformation, you will notice that some elements are highlighted with colours. These are the elements that are currently being processed and the colours have the following meaning:

  Blue The element is now about to be processed (is being "looked at").

  Yellow The element cannot be transformed right now and has been queued for later transformation (e.g., when transforming an Entry to a Card, the Box with partitions to put the Card into must be translated first).

  Green The object has just been created.

Chapter 7

Conclusion

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

Advanced Topics

In this chapter, we have tried to collect and document a series of advanced topics related to eMoflon and EA. It is kept rather compact and is meant to be used mainly as a reference, to be consulted on demand when you need it.

A.1 Advanced search

EA offers an even more advanced search capability using SQL[1]. To make use of this option, go to the model search dialog (Ctrl+Alt+A). Click on the "Builder" button and switch to the SQL-tab. Here you can formulate any query on the underlying database. The SQL-editor helps you with syntax-highlighting and auto-completion. Here are some basic examples:

- To find all eClasses

```
SELECT * FROM t_object
WHERE Object_Type='Class' AND Stereotype='eclass';
```

- To find all associations

```
SELECT * FROM t_connector
WHERE Connector_Type='Association';
```

- To find all inheritance relations

```
SELECT * FROM t_connector
WHERE Connector_Type='Generalization';
```

[1]For some detailed insights to the general database schema used by EA cf. http://www.sparxsystems.com.au/downloads/corp/scripts/SQLServer_EASchema.sql

- To find all connectors attaching a note to an element

```
SELECT * FROM t_connector
WHERE Connector_Type='NoteLink';
```

- To find all control flow edges (used in SDMs)

```
SELECT * FROM t_connector
WHERE Connector_Type='ControlFlow';
```

- To find all associations connected to a class named "EClass"

```
SELECT t_object.Name, t_connector.* FROM t_connector,t_object
WHERE t_connector.Connector_Type='Association'
  AND (t_connector.Start_Object_ID=t_object.Object_ID
    OR t_connector.End_Object_ID=t_object.Object_ID)
  AND t_object.Name='EClass';
```

- To determine all subtypes of "EClassifier"

```
SELECT a.Name FROM t_connector,t_object a,t_object b
WHERE t_connector.Connector_Type='Generalization'
  AND t_connector.Start_Object_ID=a.Object_ID
  AND t_connector.End_Object_ID=b.Object_ID
  AND b.Name = 'EClassifier';
```

- To determine all supertypes of "EClassifier" (cf. above)

```
...
  AND t_connector.Start_Object_ID=b.Object_ID
  AND t_connector.End_Object_ID=a.Object_ID
...
```

To run the search, either hit the run button in the upper left corner of the editor (it shows a triangular shaped "play" pictogram) or punch `F5` on your keyboard.

A.2 Working with multiple EAPs

Although you can simply copy & paste single packages between multiple EAPs, packages with dependencies to other packages cannot be copied so easily. If you do this via copy & paste all links will be destroyed!

- To migrate multiple packages, you have to first export a *complete* root node (a package on the top-most level in the Project Browser) from the source EAP to XMI. In preparation for a transfer of projects, it might, therefore, make sense to prepare a suitable root node with a relevant set of projects to be transferred or copied to another (target) EAP file. Right-click the root node you wish to export and select "Export Model to XMI..." (Fig. A.1). In the dialogue that pops up, enter the name and path of the exported file and choose `XMI 2.1` as "Export type".

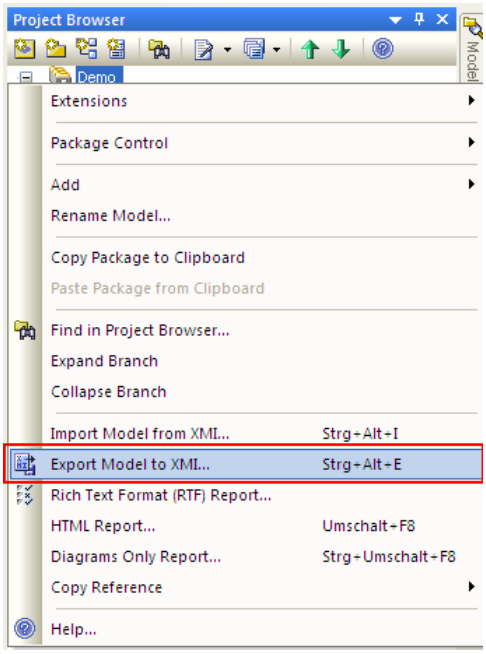


Figure A.1: Export the root node from the source EAP

- Open the target EAP, right-click on the Project Browser and select "Import Model from XMI..." (Fig. A.2). In the dialogue that pops up, enter the name and path of the file to be imported and click `Import`.
- After confirming and starting the import, you have to specify how the model should be imported. As our root models are at the root level,

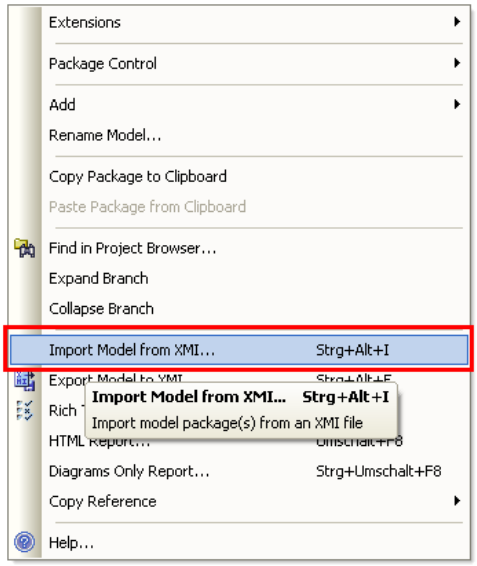


Figure A.2: Import the root node into the target EAP

choose “Yes” in the dialogue depicted in Fig. A.3. After the import, you can now delete packages in the root node, which you do not need.

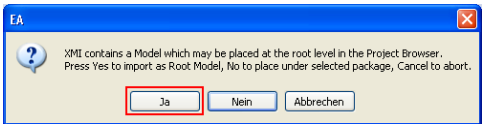


Figure A.3: Confirm the level of the root node

A.3 Using Enterprise Architect with Subversion

The following steps are required to setup EA for use with subversion. This is highly recommended when working in a group and sharing a single EA Project (EAP) file, which is otherwise a huge binary blob. We assume you wish to use (i) Subversion and (ii) Windows. For other SCM and operating systems please consult the official documentation from EA.

A.3.1 Initial preparation and set-up

Download and install Slik SVN (mandatory):

- x32: `http://www.sliksvn.com/pub/Slik-Subversion-1.7.6-win32.msi`

  x64: `http://www.sliksvn.com/pub/Slik-Subversion-1.7.6-x64.msi`

For public/private key authentication, you also need Tortoise SVN:

- x32: `http://sourceforge.net/projects/tortoisesvn/files/1.7.9/Application/TortoiseSVN-1.7.9.23248-win32-svn-1.7.6.msi/download`

  x64: `http://downloads.sourceforge.net/project/tortoisesvn/1.7.9/Application/TortoiseSVN-1.7.9.23248-x64-svn-1.7.6.msi/download`

If you do not want to have your private key password in plain text in an SVN configuration file then also download Pageant:

- `http://the.earth.li/~sgtatham/putty/latest/x86/pageant.exe`

After installing all the tools we need, we now have to setup the SSH tunnel:

- Locate the file `%APPDATA%\Subversion\config` and open it with your favourite editor. Locate the `[tunnels]` section.
- If you do not want to install Pageant and do not mind entering your password in plain text enter the following command:
  `ssh = "<path to Tortoise SVN>/bin/TortoisePlink.exe" -l <username> -pw <password for your private key> -i "<path to your private key>"`

- If you wish to use Pageant then the command can be simplified to: `ssh = "<path to Tortoise SVN>/bin/TortoisePlink.exe" -l <username>` as you can add your private key to Pageant.
- If you just use direct passwords for authentication then you can leave out the `-i` option in both cases.

A.3.2 How to set-up a version controlled EAP file

In the following we assume an EAP file has already been placed under version control *according to our tutorial* and that you wish to check-out this file and work with it. If our instructions do not work, the EAP file might have been placed under version control in a different manner. If this is the case then please contact whoever checked-in the file and set it up for working with EA and SVN for further instructions.

- Check-out the project with the EAP file from the server using TortoiseSVN or Eclipse/Subclipse (or any SVN client of your choice). You should now have a *.svn*-folder in the directory where you saved the revision.
- Open the EAP file. If the EAP file is already under version control *and has been set-up correctly*, a dialogue similar to Fig. A.4 should immediately pop-up.
- Click "Yes" to open the "Version Control Settings" dialogue (Fig. A.5).

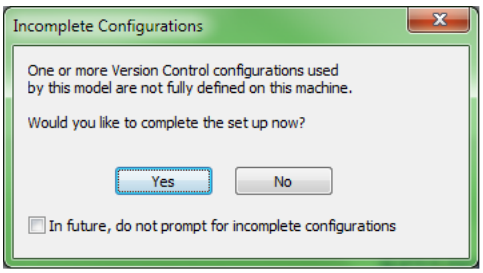


Figure A.4: Configure an EAP file which is already under version control

- To work with the EAP file, you now have to *redefine* the SVN variable for the file in your EA workspace. To accomplish this, choose the local path to the folder which contains the EAP file in the "Working Copy Path" text-box, and correct the value in "Subversion Exe Path" if necessary (to fit your Slik installation location).

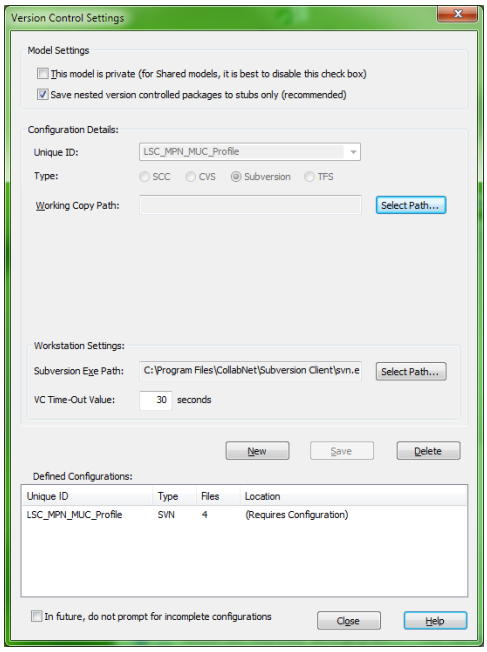


Figure A.5: Update settings as required

A.3.3 Working with a version controlled EAP file

- A `Check Out` retrieves the lock for a certain package and gives you exclusive access, i.e., no one else can change the package. Very important: if subpackages are also under version control, they are not affected by checking out the "super"-package and remain locked. A `Check Out` also updates the package to the latest version.
- A `Check In` commits your work to the server and gives up the lock on the package so others can work on it. If you do not want to commit your changes, you can just use `Undo Check Out...` to revert all local changes.
- The corresponding `..Branch` options perform the actions for the current package and all subpackages. Please note, this has nothing to do with "branching" in normal SVN lingo.
- `Get Latest/Get All Latest` retrieves the latest version of the selected package / all packages. This is basically an update but does not retrieve the lock for any package.
- Conversely, `Put Latest` saves all your changes without giving up any locks.

- `Compare with controlled version` can be used to review incoming changes. Green elements will be added, red will be deleted.
- `File History` gives you a summary of all commits made while you were lying on the beach. For a useful file history, always use meaningful commit statements when checking in! A date stamp is created automatically.

A.3.4 Placing an EAP file under version control

If you already have an EAP file and would like to place it under version control, you first have to check it in as usual on the server using your favourite SVN client. Once the project is checked in, the required .svn folder should be in the folder containing the EAP file. The next step is to register an SVN-variable in EA:

- Open the EAP file, right click on a root folder and select "Package Control" and then "Version Control Settings..." (Fig. A.6).

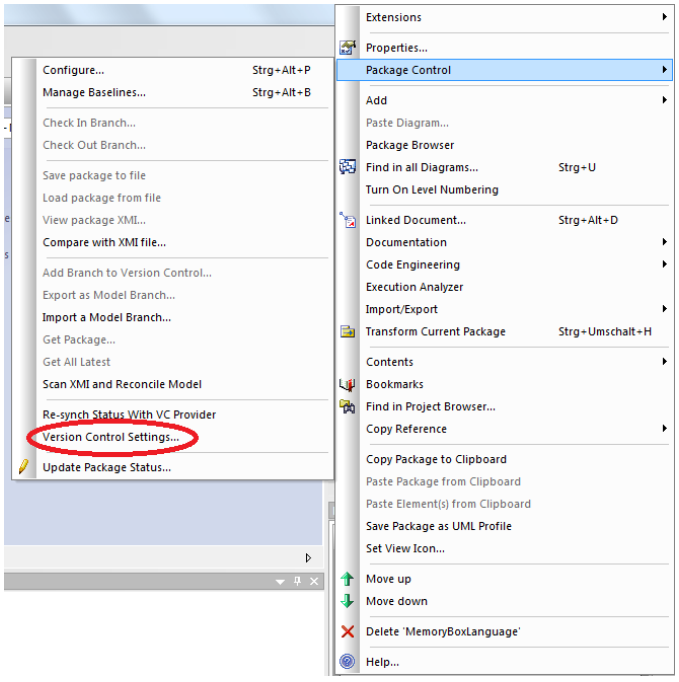


Figure A.6: Select version control settings

- In the dialogue, choose a unique ID of your choice (we suggest you use

the name of the EAP file) for the settings and activate the "Subversion" radio button below.

- Choose the local path to the folder which contains the EAP file in the "Working Copy Path" text-box.

- The field "Working Station" must point to where you installed Sliksvn, i.e., `<path to SlikSVN>\bin\svn.exe"`). Press "Save" and close the dialogue (Fig. A.7). If the dialogue closes without an error message, then you can be sure to have configured everything correctly.

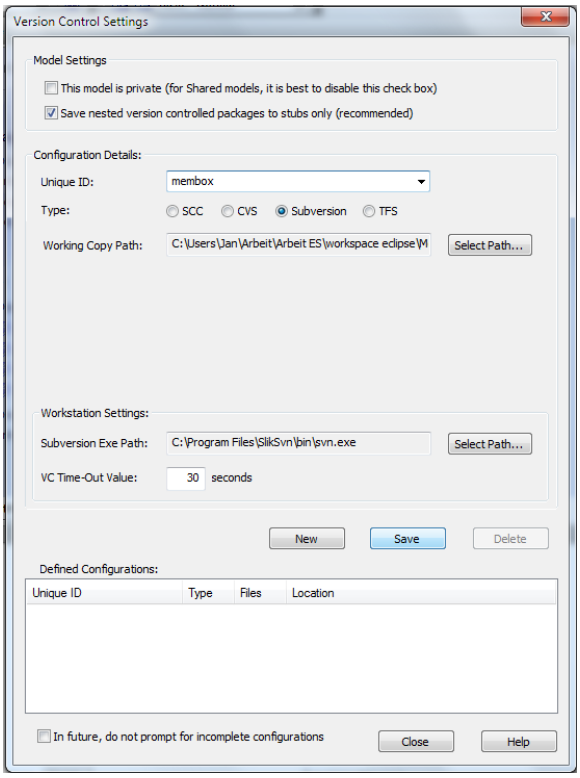


Figure A.7: Register an SVN variable in EA

- In the EAP file, choose "Package Control\Configure..." for *each package* you wish to place under version control.

- In the ensuing dialogue, activate "Control Package" and select your previously defined SVN variable from the drop-down menu. Enter the path where the XML file for the project should be placed. Although this is not enforced in any way, we recommend you create a folder structure that mirrors the package structure in EA (Fig. A.8). This

process has to be repeated *for all sub-packages* as soon as their super-package has been placed under version control.

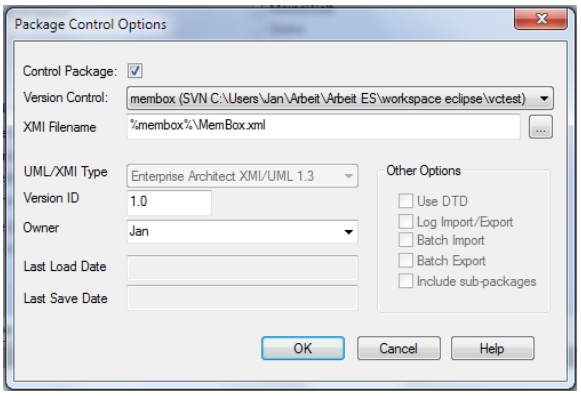


Figure A.8: Placing a package under version control

- As a final step, check-in the current state of the EAP file directly with your SVN client. As from this point, the EAP file should not be checked-in anymore, and all versioning actions should be performed via EA (and not directly with your SVN client).

A.4 Conditional branching with StatementNodes

When working with SDMs, you often need to choose between two different patterns based on the return value of an arbitrary (black box) operation. This is like the normal `Success`/`Failure` construction, but instead of a pattern being matched or failing, the decision can be implemented with a another SDM or a standard Java method. This feature is a further means (besides `MethodCallExpressions` for attribute values and `Bindings`) of integrating hand-written Java code in SDMs (and can lead to spaghetti SDMs so please use with caution!).

As an example, consider a class `A` with an operation:

`doSomeCheck(`p_1`,...,`p_n` :EClass) :EBoolean`

This method could be implemented in hand-written Java code or be specified by another SDM specification.

Although dummy boolean attributes *could* be used to achieve the same effect, it is much simpler to branch the control flow based on the result of a `StatementNode`. If the method returns an `EBoolean`, `Success` and `Failure` correspond to `true` and `false`, respectively. If the method returns anything else, then `Failure` corresponds to `null`. Void methods *cannot* be used to branch and an exception is thrown during code generation.

Fig. A.9 depicts the class `A` and shows how `doSomeCheck` is used to branch in an SDM. Fig. A.10 depicts the corresponding generated if/else branch in Java.

Perform the following steps to branch with a `StatementNode`:

- Add a new statement node (cf. Fig. A.11) at the appropriate location in your SDM.
- Invoke a non-void method (an operation in the metamodel) via a `MethodCallExpression` (cf. Fig. A.12).
- Add `Success` *and* `Failure` edges to the `StatementNode` to branch appropriately.

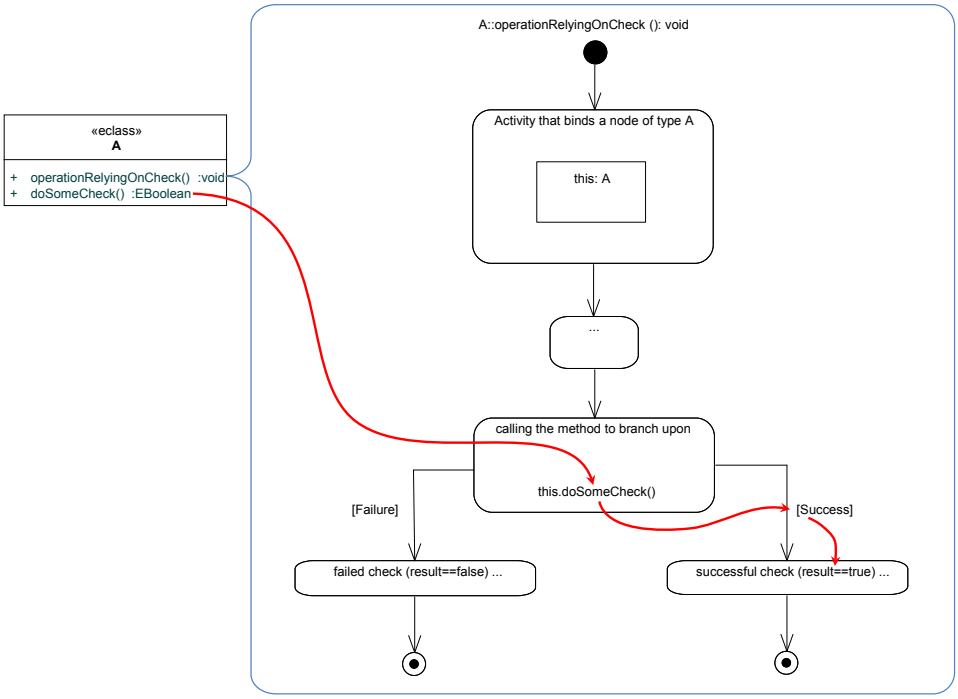


Figure A.9: Conditional branching based on the result of an operation

```
public void operationRelyingOnCheck() {
	boolean fujaba__Success = false;
	// some other code
	// ...
	// calling the method to branch upon
	fujaba__Success = this.doSomeCheck();
	if (fujaba__Success) {
		// successful check (result==true) ...
		return;
	} else {
		// failed check (result==false) ...
		return;
	}
}
```

Figure A.10: Generated code for branch

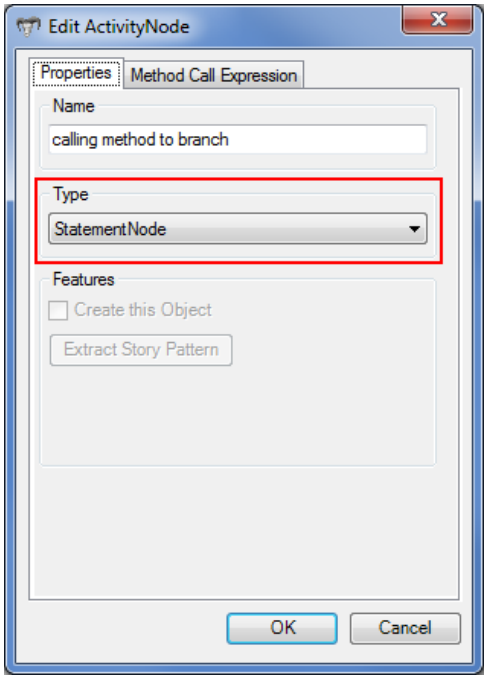


Figure A.11: Switch from activity node to statement node

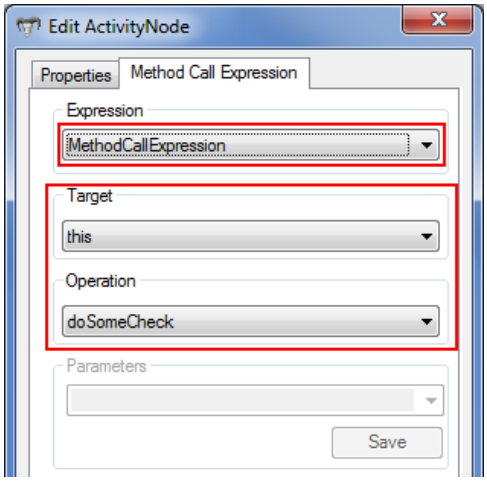


Figure A.12: Specify the method call expression

A.5 Injections

This chapter describes some extra options when using our injections to integrate handwritten code with generated code. This complements the short introduction in Chapter 3.6.

As an example, we shall provide functionality for saving our DoubleLinkedList to a file.

- Begin with an empty workspace, create a new metamodel Demo and replace the Demo.eap with the file from our .zip file (in the folder containing this tutorial).
- Now open Demo.eap and change the "Default Language for Code Generation" (Tools → Options → Source Code Engineering) to Ecore.
- Add the method void toFile(EString path) to the class List (Fig. A.13) and export the project to your Eclipse workspace.

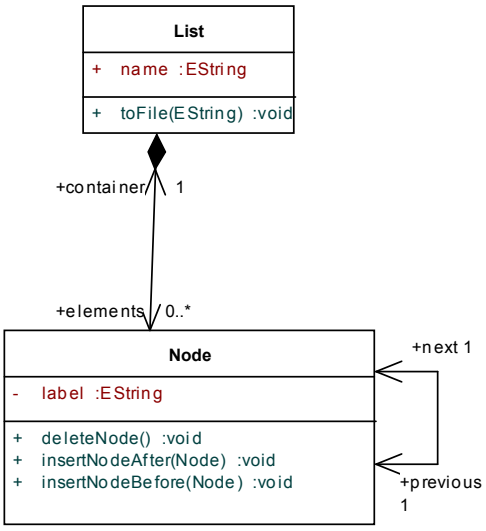


Figure A.13: Add a new method toFile to the class List

- To implement the toFile method, right-click gen/DoubleLinkedListLanguage/-List.java in the generated Eclipse project and choose "eMoflon/create Injection for class", which will generate the file injection/DoubleLinkedListLanguage/List.inject. Insert the code depicted in Fig. A.14 into this file.

- To make use of some basic templates for code completion, type "@", press "Ctrl+Shift" and choose (in this case) "Inject method in model". Functions that are annotated with `@model` implement methods that are defined in the interface of the model, i.e., are explicitly modelled as operations in EA, and have the structure:
`@model [signature] <-- [code] -->`.

```
partial class List
{
@model toFile(String path) <--

        try{
        FileWriter fstream = new FileWriter(path);
        BufferedWriter out = new BufferedWriter(fstream);
        out.write(this.toText());

        out.close();
        }catch (Exception e){
        System.err.println(e.toString);
        }

-->
}
```

Figure A.14: Injection for List::toFile(String)

You may have noticed that we use an undefined method List::toText() in the implementation of the toFile method. We will now inject this as a private method in the implementation class ListImpl.java.

- Right-click gen/DoubleLinkedListLanguage/impl/ListImpl.java and choose "eMoflon/create Injection for class" to create injection/DoubleLinkedListLanguage/impl/ListImpl.inject. Implement the private method toText() with the code depicted in Fig. A.15.
- When you now rebuild the project (right-click on DoubleLinkedListLanguage and choose "eMoflon/Build and Clean"), this code will be injected into ListImpl.java.

In this way, member functions and fields that were *not* modelled in EA can be injected using the `@members` keyword. Everything between `<-- -->` is copied into the end of the generated ListImpl.java file without any restrictions at all. Note how imports can also be injected (as done here for java.io.*).

Although it is possible to inject members in the interface file (i.e., List.java in this example), this is considered dirty and should be used very sparingly (if possible never). You can have several `@model` injections in your .inject file, but only a single `@member` scope.

```
import java.io.*;

partial class ListImpl
{
    @members <--

    private String toText(){
        StringBuilder sb = new StringBuilder();

        for(Node element : elements){
            sb.append(element.getLabel());
            sb.append("\n");
        }

        return sb.toString();
    }

    -->
}
```

Figure A.15: Injection for toText() in ListImpl.java

A.6 Using Existing EMF Projects in eMoflon

This chapter contains stepwise instructions on how to use existing EMF/-Ecore projects with an eMoflon project. As an example, we shall implement a small subset of the `Ecore -> GenModel` transformation. The *GenModel* for a given Ecore model can be viewed as a *wrapper* that contains additional Java code generation specific details. These details are separated from the Ecore model to keep it free of such "low-level" information and settings.

As the metamodel for GenModels is part of the EMF/Ecore standard, this is an example of an existing metamodel which must be integrated in eMoflon before you can, for example, specify the transformation using SDMs. Based on this example, the basic workflow for using an existing EMF project in eMoflon is described in the following.

A.6.1 Modelling relevant aspects in EA

The first step is to get the existing metamodel into EA. A complete and automatic import of existing Ecore files in EA is currently not possible and, therefore, *relevant parts* of the existing metamodel (`GenModel`) have to be modelled manually in EA. Although this might sound frightening (especially for large complex metamodels), the emphasis here on *relevant* indicates that only elements that are used for the transformation have to be present in EA and can be added iteratively as the transformation grows.

- To specify our example transformation, create a new metamodel project `EcoreToGenModel` in Eclipse, check the option `Add eMoflon languages` in the wizard, and switch to EA by double-clicking the created `EcoreToGenModel.eap` file.
- Note the packages already present in EA (eMoflon Languages), especially Ecore, which we shall use for the transformation.
- Create a new package `GenModel` and model the elements as depicted in Fig. A.16. The actual `GenModel` metamodel contains lots more elements, but this subset is sufficient for our task.

- Create another package `Ecore2GenModel` to contain the `Transformer` class with the methods as depicted in Fig. A.17.
- Implement the SDMs depicted in Figs. A.18 and A.19.

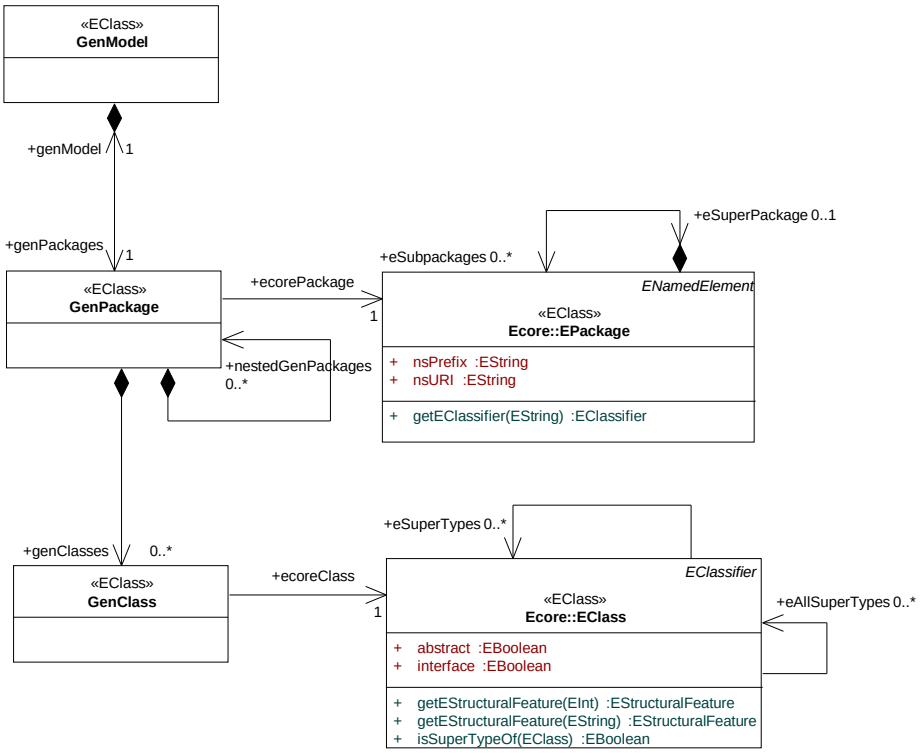


Figure A.16: MetaModel of GenModel

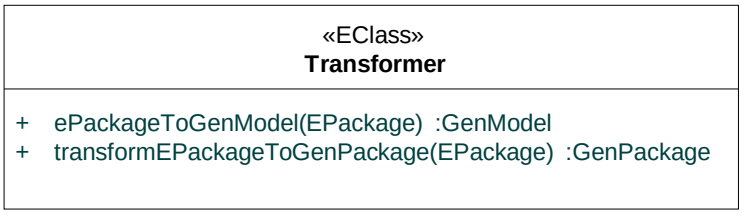


Figure A.17: Methods in Transformer

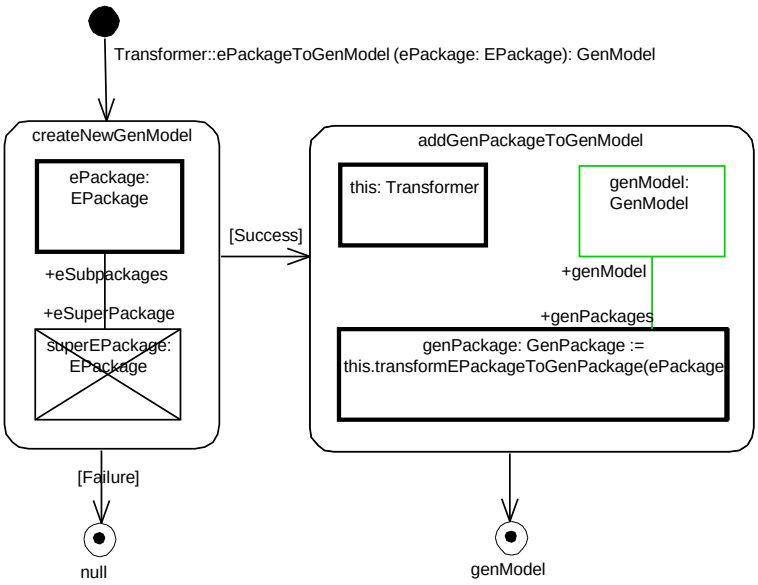


Figure A.18: Main method for EPackage to GenModel transformation

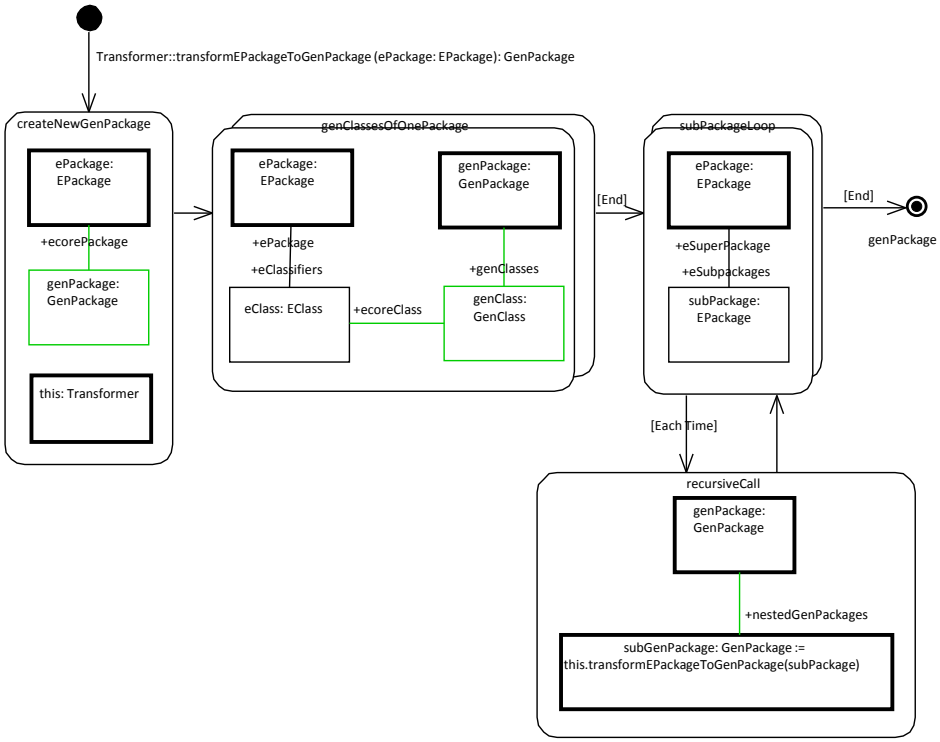


Figure A.19: Helper function to transform all EPackages to GenPackages

A.6.2 Configuration for code generation in Eclipse

As there is already generated code (provided via a plugin in Eclipse) for the existing GenModel metamodel, we do *not* want to export our incomplete subset of GenModel in EA.

- To prevent this, right-click the GenModel package in EA and select "Properties/Moflon" and change the tagged value `Moflon::Export` to `false` (Fig. A.20).

Furthermore, we have to set the "real" name and URI of the project to be used in Eclipse so that references are exported properly.

- In the "Properties/Moflon" dialogue for GenModel, create the new tagged values `Moflon::CustomNsPrefix` and `Moflon::CustomNsUri` and set them according to Fig. A.20. These values can be determined by inspecting the corresponding values in the existing .ecore file (i.e., the existing metamodel).

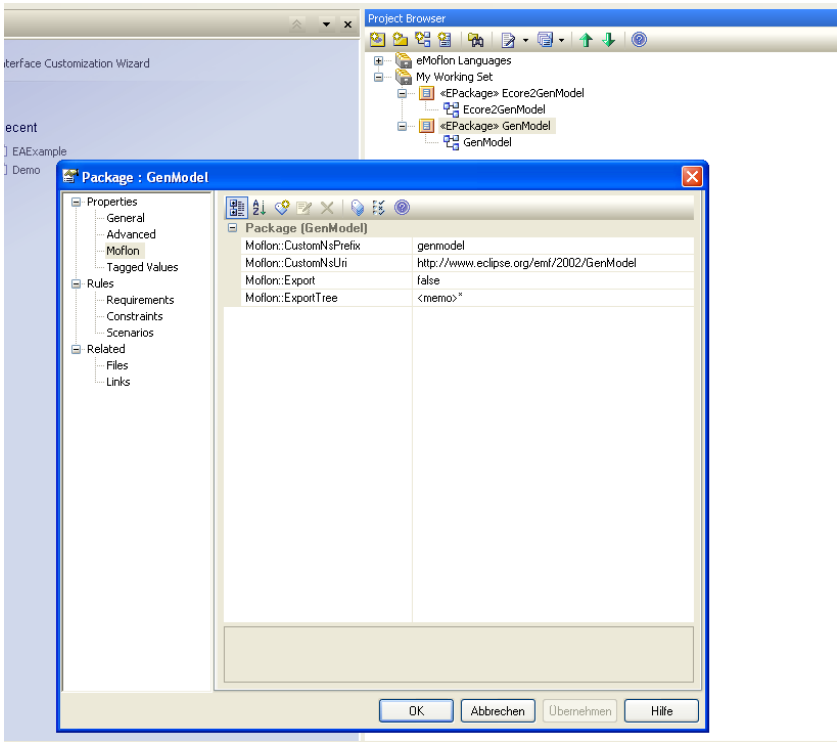


Figure A.20

- Export all projects as usual to your Eclipse workspace and update the metamodel project by pressing F5.
- Convert the generated Eclipse project `Ecore2GenModel` to a *plugin project* by right-clicking the project and selecting "Configure/Convert to Plug-in Projects...". This makes it easier to set the required dependencies for code generation.
- Now right-click `Ecore2GenModel` and choose "Plug-in Tools/Open Manifest". In the window that opens up, choose the `Dependencies` tab, click `Add`, and type in `org.eclipse.emf.codegen.ecore` (which includes both the Ecore and GenModel libraries as required).

Although we have already specified the name and URI of the existing project (in our case GenModel) in EA, we now have to tell eMoflon where to find the implementation (generated code) for the existing project.

- Open the `moflon.properties` file located in your project folder and insert the following lines:

```
ADDITIONAL_DEPENDENCIES=platform:/plugin/org.eclipse.emf.codegen.ecore/model/GenModel.ecore
ADDITIONAL_USED_GEN_PACKAGES=platform:/plugin/org.eclipse.emf.codegen.ecore/model/GenModel.genmodel
```

Finally, to compensate for some cases where our naming conventions were violated, add the following mappings as corrections:

- An *import mapping* for correct generation of the required import:

```
IMPORT_MAPPINGS=genmodel-> org.eclipse.emf.codegen.ecore.genmodel
```

- A *factory mapping* to ensure that `GenModelFactory` is used as the factory for creating elements in the transformation instead of `GenmodelFactory`, which would be the default convention:

```
FACTORY_MAPPING=genmodel-> GenModelFactory
```

Your moflon.properties file should now closely resemble Fig. A.21. Now generate code once more for the project and ensure with a JUnit test that the transformation behaves as expected.

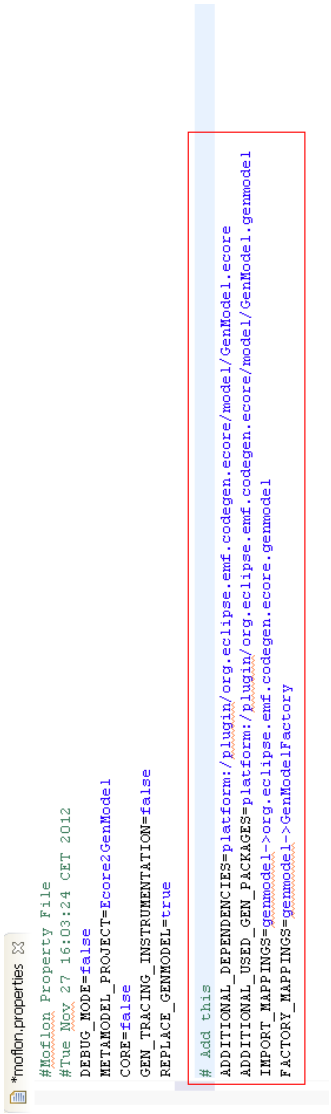


Figure A.21: Additional properties for code generation

A.7 Using the integrator with breakpoints

You can set breakpoints for the integrator in order to stop when a specific rule is called.

- Import the file `after_chapter_6.zip` into your workspace.
- Create a class, e.g., `org/moflon/tie/Breakpoint` in your `src` folder of the `LearningBoxToDictionaryIntegration` project.
- Implement it with the code given in Fig. A.22.

```
package org.moflon.tie;

import org.moflon.integrator.IntegratorObserver;
import org.moflon.integrator.builder.IntegratorObservable;

import TGGLanguage.algorithm.protocol.AppropriateCandidate;
import TGGLanguage.algorithm.protocol.Candidate;
import TGGLanguage.algorithm.protocol.CandidateProtocol;
import TGGLanguage.algorithm.protocol.TranslationStep;

public class Breakpoint extends IntegratorObserver {

  @Override
  public void update(IntegratorObservable o, TranslationStep arg) {
    if(arg instanceof CandidateProtocol)
      for(Candidate c : ((CandidateProtocol) arg).getCandidates())
        if(c instanceof AppropriateCandidate &&
            c.getRule().matches("BoxToDictionaryRule"))
          o.stop(this, "BoxToDictionaryRule reached");
  }

}
```

Figure A.22: Implementation of a breakpoint

- Replace line 25 in `StartIntegrator.java` (same package) with the following code to register the breakpoint:

```
IntegratorObservable integrator = IntegratorApp.startIntegrator(args);
integrator.addObserver(new Breakpoint());
```

- Start the integrator for `corr_BWD.xmi` and `protocol_BWD.xmi`.
- Press `Ctrl+Shift+Alt+Right` to jump to your breakpoint. This should be the first application of `BoxToDictionaryRule`.

Appendix B

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] Holger Giese, Stephan Hildebrandt, and Leen Lambers. Toward Bridging the Gap between Formal Semantics and Implementation of Triple Graph Grammars. In *2010 Workshop on Model-Driven Engineering, Verification, and Validation*, pages 19–24. IEEE, October 2010.

[5] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In Thomas Whittle, Jon and Clark, Tony and Kühne, editor, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 668–682, Berlin / Heidelberg, 2011. Springer.

[6] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Andy Schürr, C. Lewerentz, G. Engels, W. Schäfer, and B. Westfechtel, editors, *Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science*, pages 141–174. Springer, Heidelberg, November 2010.

[7] M Lauder, A Anjorin, G Varró, and A Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In *Proceedings*

of the 6th International Conference on Graph Transformation, Lecture Notes in Computer Science (LNCS), Heidelberg, 2012. Springer Verlag.

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

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

[10] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009.

[11] Terence John Parr. Enforcing strict model-view separation in template engines. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the Thirteenth International World Wide Web Conference*, pages 224–233, New York, NY, May 2004. ACM Press.

[12] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In G Tinhofer, editor, *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer Verlag.

[13] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *4th International Conference on Graph Transformation*, volume 5214 of *Lecture Notes in Computer Science (LNCS)*, pages 411–425, Heidelberg, 2008. Springer Verlag.

Appendix C

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-

AGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.