An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part 0: Introduction

For eMoflon Version 2.16.0

File built on 21st September, 2016

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

The eMoflon team
Darmstadt, Germany (September 2016)

Part 0:

Introduction

This handbook 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 at contact@emoflon.org.

`URL of this document: https://emoflon.github.io/eclipse-plugin/beta/handbook/part0.pdf`

Figure 0.1: How you should feel when you're done

To enjoy the experience, you should be fairly comfortable with Java or a comparable object-oriented language, 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, 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.

So, what is eMoflon?

eMoflon is a tool for building tools. If you wish, a "meta" tool. This means that if you're interested in building *domain-specific* tools for end users, then eMoflon could be pretty useful for you.

Why should I be interested?

To build a tool, you typically need a way for users to communicate with it, i. e., you must establish a suitable *language* for specifying input and output to and from the tool. You also need a central data structure to represent the "state" of the tool. This data structure or *model* is usually manipulated and appropriately *transformed* in some useful manner by the tool. Many tools also *generate* something useful from their internal models and keep them synchronized with other models in different tools. To achieve these goals you can use *metamodeling* to define your language (your *meta*model), *graph transformations* to transform your models. All this and much more is supported by eMoflon. Take a look at Figure 0.2 to see how all these tasks fit together.

What does this handbook cover?

On the last page, we've described each of the five parts that make up this handbook. You can work through them sequentially and become an *official*[1] eMoflon master or, depending on your interests, decide what you'd like to read and what to skip. We provide all the necessary materials (i. e., a cheat package) so you can jump right in without having to complete the previous parts. For those of you interested in further details and the mature

[1] Certificate not guaranteed

formalism of graph transformations, we give relevant references throughout the handbook.

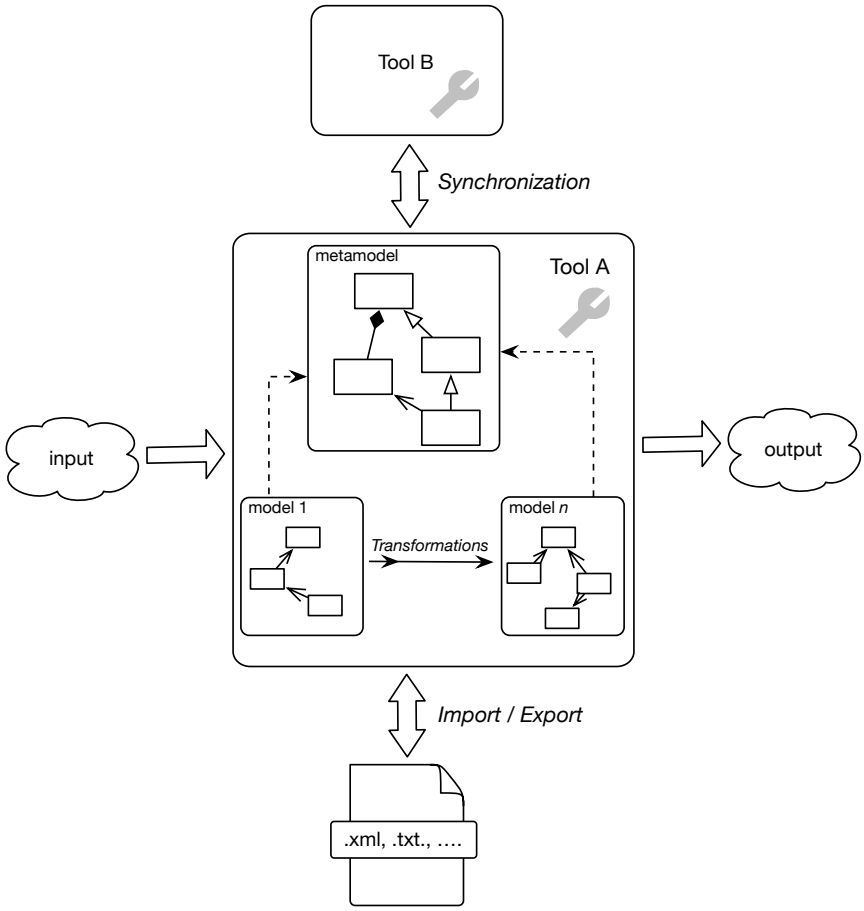


Figure 0.2: Building a tool requires language specification (metamodeling), transformations, synchronization, and code generation

Part I: Installation and set up provides a very simple example and a few JUnit tests to test the installation and configuration of eMoflon. We also explain the general workflow and the different workspaces involved.

This part can be considered *mandatory* only if you are new to eMoflon, but we recommend working through it anyway.

File `https://emoflon.github.io/eclipse-plugin/beta/handbook/part1.pdf`

Part II: Ecore takes you step-by-step through a more realistic example that showcases many of the features we currently support. Working through this part should serve as a basic introduction to model-driven engineering, and is especially recommended if you're new to metamodeling (using Ecore and the Eclipse Modeling Framework (EMF)).

File `https://emoflon.github.io/eclipse-plugin/beta/handbook/part2.pdf`

Part III: Story Driven Modelling (SDM) introduces *unidirectional* model transformation via programmed graph transformation using story diagrams (SDMs).

File `https://emoflon.github.io/eclipse-plugin/beta/handbook/part3.pdf`

Part IV: TGGs introduces *bidirectional* model transformation with Triple Graph Grammars (TGGs).

File `https://emoflon.github.io/eclipse-plugin/beta/handbook/part4.pdf`

Part V: Miscellaneous contains a collection of tips and tricks to keep on hand while using eMoflon. This can be used as a reference to help avoid common mistakes and increase productivity. If you're in a hurry, this part can be skipped and consulted only on demand.

File `https://emoflon.github.io/eclipse-plugin/beta/handbook/part5.pdf`

Well, that's it! Download Part I, grab a coffee, and enjoy the ride!

An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part I: Installation and Setup

For eMoflon Version 2.16.0

File built on 21st September, 2016

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

The eMoflon team
Darmstadt, Germany (September 2016)

Contents

Part I:

Installation and Setup

This part provides a very simple example and a JUnit test to check the installation and configuration of eMoflon. It can be considered *mandatory* if you are new to eMoflon, but we recommend working through it anyway.

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

URL of this document: https://emoflon.github.io/eclipse-plugin/beta/handbook/part1.pdf

1 Getting started

If, however, you're finding that the screenshots we've taken aren't matching your screen and you *ARE* in the right place, please send us an email at contact@emoflon.org and let us know. They get outdated so fast! They just grow up, move on, start doing their own thing and ... uh, wait a second. We're talking about pictures here.
If you have problems while using Moflon you can also have a look at our FAQs: https://github.com/eMoflon/emoflon-docu/wiki/eMoflon-FAQ.

1.1 Install our plugin for Eclipse

- Make sure that you have **Java 1.8** installed.
- Download and install Eclipse Marsfor modelling, which is called "**Eclipse Modeling Tools**" from `http://www.eclipse.org/downloads/index.php` (Figure 1.1).[1]
- Install our Eclipse Plugin from the following update site[2]:

[1] Please note that you *have to* install Eclipse Modelling Tools, or else some features won't work! Although different versions may support eMoflon, our tool is currently tested only for Mars and Java 1.8.

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

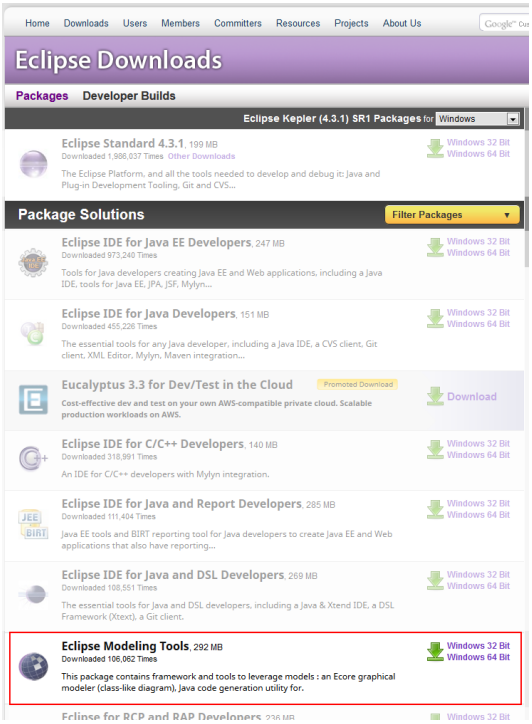


Figure 1.1: Download **Eclipse Modeling Tools**

https://emoflon.github.io/eclipse-plugin/beta/update-site2/

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

Hint: To inform you about new updates, we provide some mailing lists for you: http://www.emoflon.org/emoflon/mailing-lists/

1.2 Install our extension for Enterprise Architect

Enterprise Architect (EA) is a visual modelling tool that supports UML[3] and a host of other modelling languages. EA is not only affordable but also quite flexible, and can be extended via *extensions* to support new modelling tools – such as eMoflon!

- Download EA for Windows from http://www.sparxsystems.com/ to get a free 30 day trial and follow installation instructions (Figure 1.2).

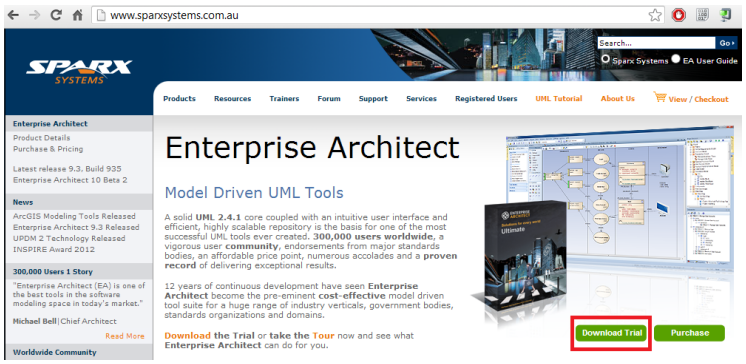


Figure 1.2: Download Enterprise Architect

- Install our EA extension (Figure 1.3) to add support for our modelling languages. Download https://emoflon.github.io/eclipse-plugin/beta/update-site2/ea-ecore-addin.zip, unpack, and run `eMoflonAddinInstaller.msi`.

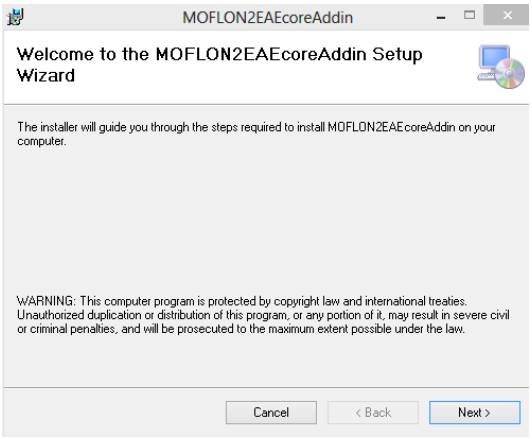


Figure 1.3: Install our extension for EA

[3]Unified Modelling Language

2 Get a simple demo running

▶ Open Eclipse to a clean, fresh workspace. Go to *Window* →*Open Perspective* →*Other...* [4] and choose `eMoflon` (Figure 2.1).

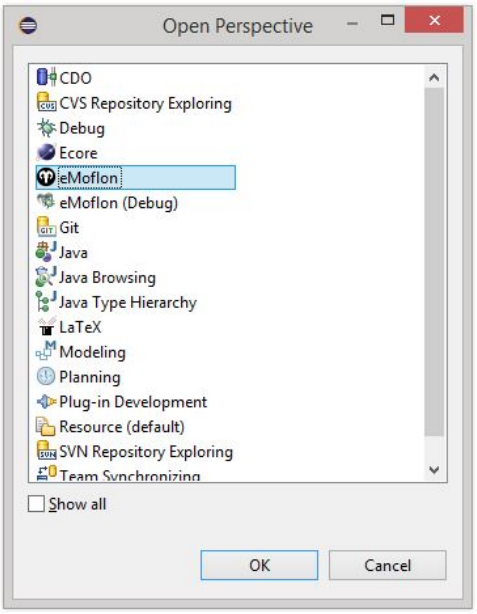


Figure 2.1: Choose the eMoflon perspective

▶ At either the far right or center of the toolbar, a new action set should have appeared. Navigate to *"eMoflon Cloud"* →*Install Workspace* (Figure 2.2).

▶ In this menu you can check out different workspaces for Eclipse. Here you can also check out workspaces for the handbook tutorials (*eMoflon Examples*). How you can create your own projects is described in Part I. As the first tutorial select *eMoflon Examples* →*Demo (Double-Linked List)*.

All our handbook examples are provided via Git and are hosted on GitHub[5]. If you encounter problems when fetching some handbook example, the reason may be that your checked out working copy is in a "dirty" state. In this case, it is safe to remove your whole working copy as follows: Navigate to the *Git* perspective in Eclipse (*Window*

[4] A path given as *foo* →*bar* indicates how to navigate in a series of menus and toolbars. New definitions or concepts will be *italicized*, and any data you're required to enter, open, or select will be given as `command`.

[5] See `https://github.com/eMoflon/emoflon-examples`.

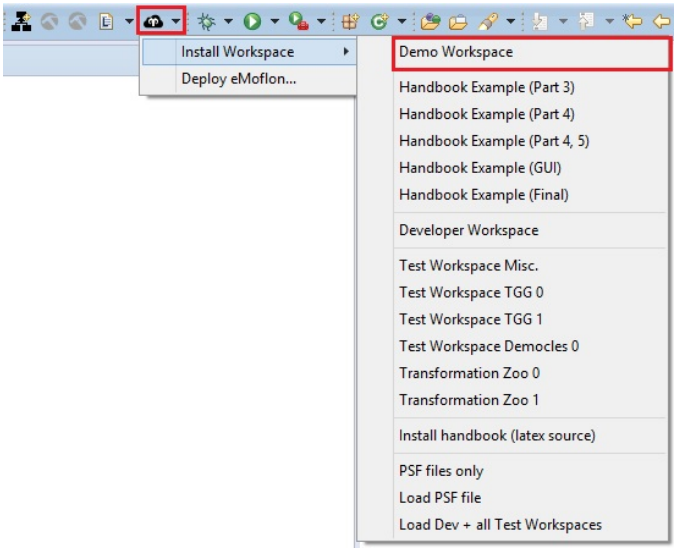


Figure 2.2: Invoking "Install Workspace → eMoflon Examples → Demo (Double-Linked List)"

→*Perspective* →*Open Perspective* →*Other...*), right-click your working copy `emoflon-examples` and choose *Delete Repository...*. Make sure to fully delete the repository by ticking all boxes.

- Another button in the new action set is *View and configure logging* represented by an `L` (Figure 2.3). Clicking this icon will open a `log4jConfig-.properties` file where you can silence certain loggers, set the level of loggers, or configure other settings.[6] All of eMoflon's messages appear in our console window, just below your main editor. This is automatically opened when you selected the `eMoflon` perspective and contains important information for us if something goes wrong!

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

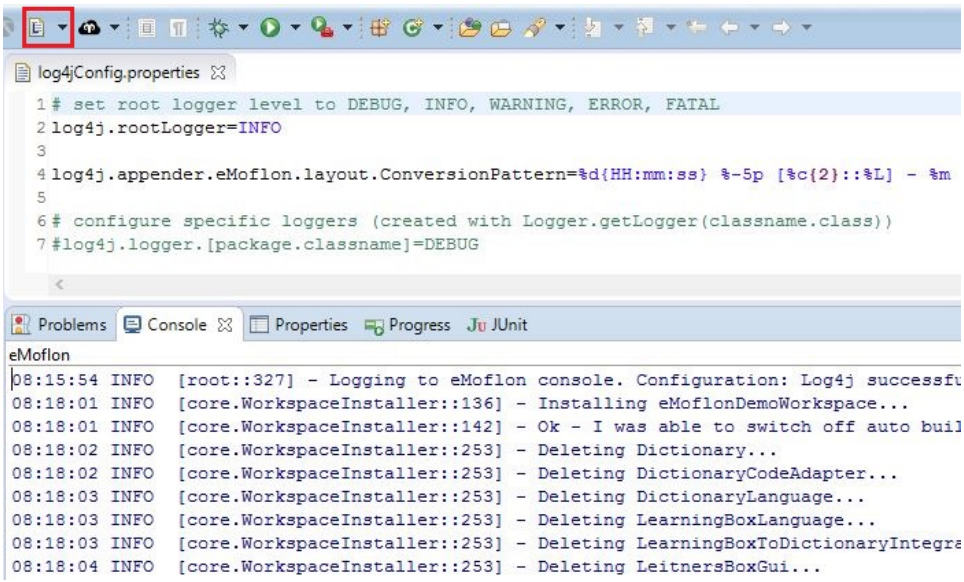


Figure 2.3: The eMoflon console with log messages

2.1 A first look at EA

- Can you locate the new `Demo.eap` file in your package explorer? This is the EA project file you'll be modelling in. Don't worry about any other folders at the moment - all problems will be resolved by the end of this section.

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

- Double-click `Demo.eap` to start EA, and choose `Ultimate` when starting EA for the first time.

- In EA, select *Extensions* →*Add-In Windows* (Figure 2.4). This will activate our tool's full control panel. If nothing happens the installation was probably not successful. Work again through the installation or have a look at the following site:
  https://github.com/eMoflon/emoflon-docu/wiki/Howto-on-Using-EA-With-A-Second-Windows-Account.

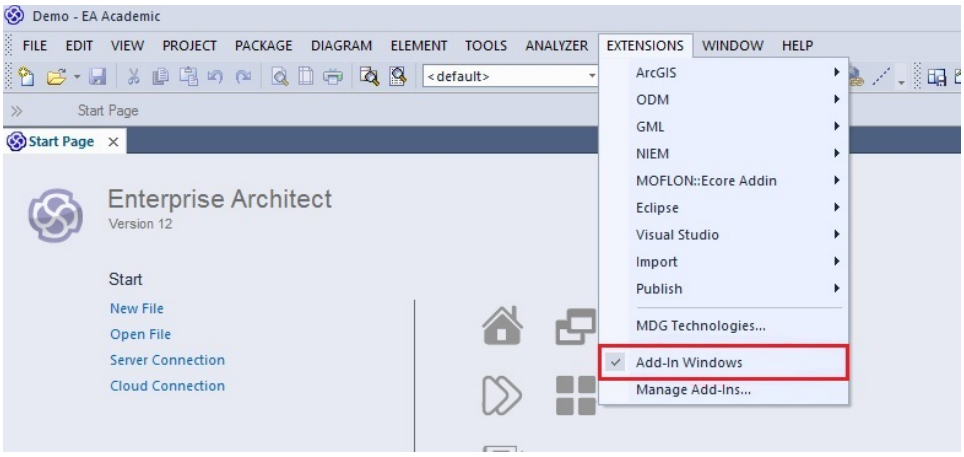


Figure 2.4: Export from EA

- This tabbed control panel provides access to all of eMoflon's functionality. This is where you can validate and export your complete project to Eclipse by pressing `All` (Figure 2.5).

- Now try exploring the EA project browser! Try to navigate to the packages, classes, and diagrams. Don't worry if you don't understand that much—we'll get to explaining everything in a moment. Just make sure not to change anything!

- Switch back to Eclipse, choose your metamodel project, and press `F5` to refresh. The export from EA places all required files in a hidden folder (`.temp`) in the project. A new, third project named

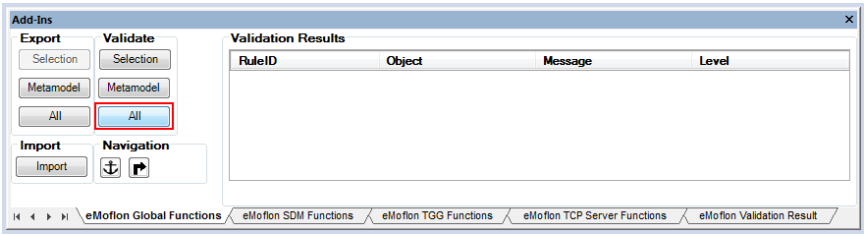


Figure 2.5: eMoflon's control panel in EA

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

- The three asterisks signal that the project still needs to be built (Figure 2.6).

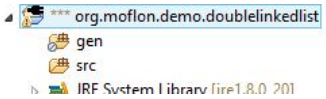


Figure 2.6: Dirty projects are marked with ***

- Now, right-click `org.moflon.demo.doublelinkedlist` and choose *eMoflon* →*Build* (or use the shortcut `Alt+Shift+E,B`[7]).

  eMoflon now generates the Java code in your repository project. You should be able to monitor the progress with the green bar in the lower right corner (Figure 2.7). Pressing the symbol opens a monitor view that gives more details of the build process. You don't need to worry about any of these details, just remember to (i.) refresh your Eclipse workspace after an export, and (ii.) rebuild projects that bear a "dirty marker" (`***`).

- If you're ever worried about forgetting to refresh your workspace, or if you just don't want to bother with having to do this, Eclipse does offer an option to do it for you automatically. To activate this, go to *Window* →*Preferences* →*General* →*Workspace* and select `Refresh on access`.

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

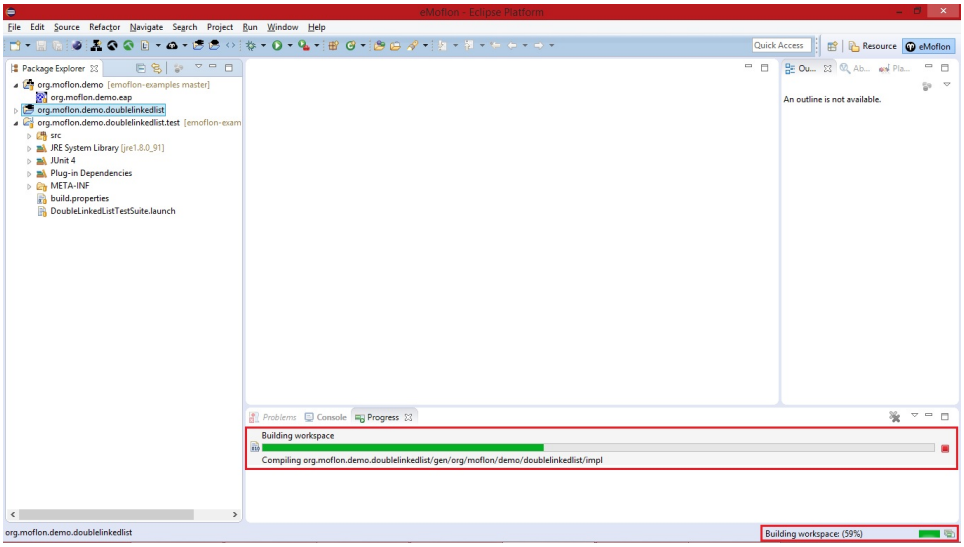


Figure 2.7: Eclipse workspace while building the demo project

3 Validate your installation with JUnit

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

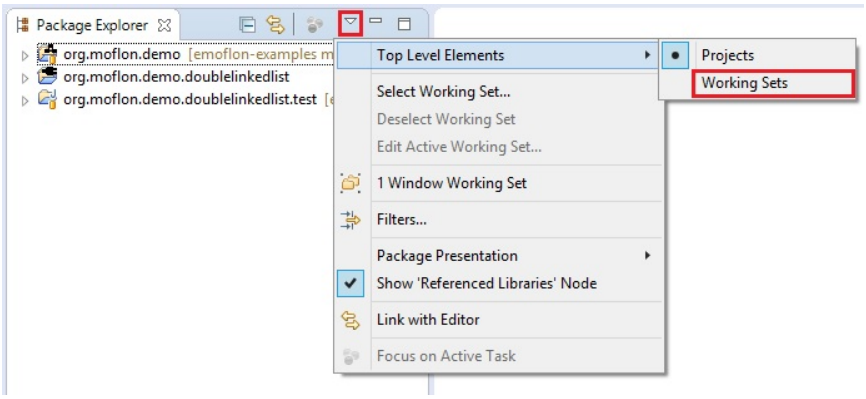


Figure 3.1: Top level elements in Eclipse

- Locate *Other Projects →org.moflon.demo.doublelinkedlist.test*. This is the testsuite imported with the demo files to make sure everything has been installed and set up correctly. Right click on the project to bring up the context menu and go to *Run As →JUnit Test*. If anything goes wrong, try refreshing by choosing your metamodel project and pressing `F5`, or right-clicking and selecting `Refresh`.

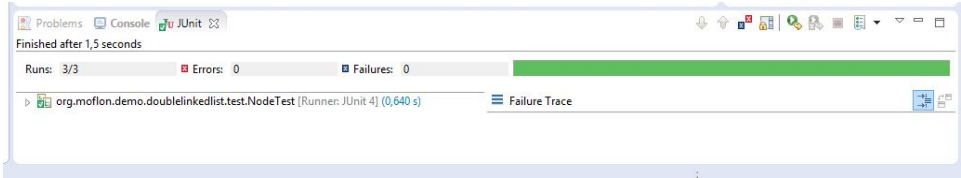


Figure 3.2: All's well that ends well...

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

4 Project setup

4.1 Your Enterprise Architect Workspace

Now that everything is installed and setup properly, let's take a closer look at the different workspaces and our workflow. Before we continue, please make a few slight adjustments to Enterprise Architect (EA) so you can easily compare your current workspace to our screenshots. These settings are advisable but you are, of course, free to choose your own colour schema.

- Select *Tools* → *Options* → *Themes* in EA, and set Diagram Theme to `Enterprise Architect 10`.
- Next, proceed to *Gradients and Background* and set *Gradient* and *Fill* to `White` (Figure 4.1).

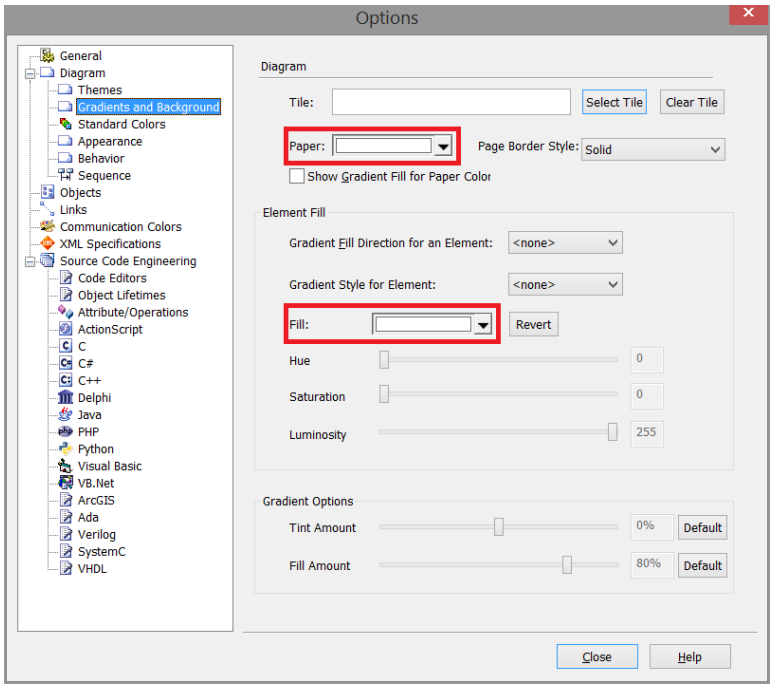


Figure 4.1: Suggested paper background and element fill

- In the *Standard Colors* tab, and set your colours to reflect Figure 4.2.
- In the same dialogue, go to *Diagram* → *Appearance* and reflect the settings in Figure 4.3. Again, this is just a suggestion and not mandatory.

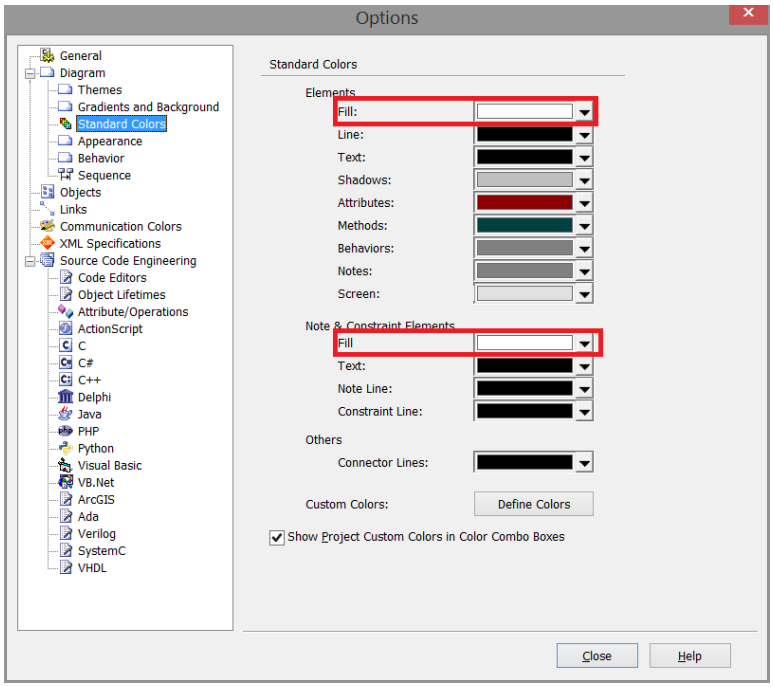


Figure 4.2: Our choice of standard colours for diagrams in EA

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

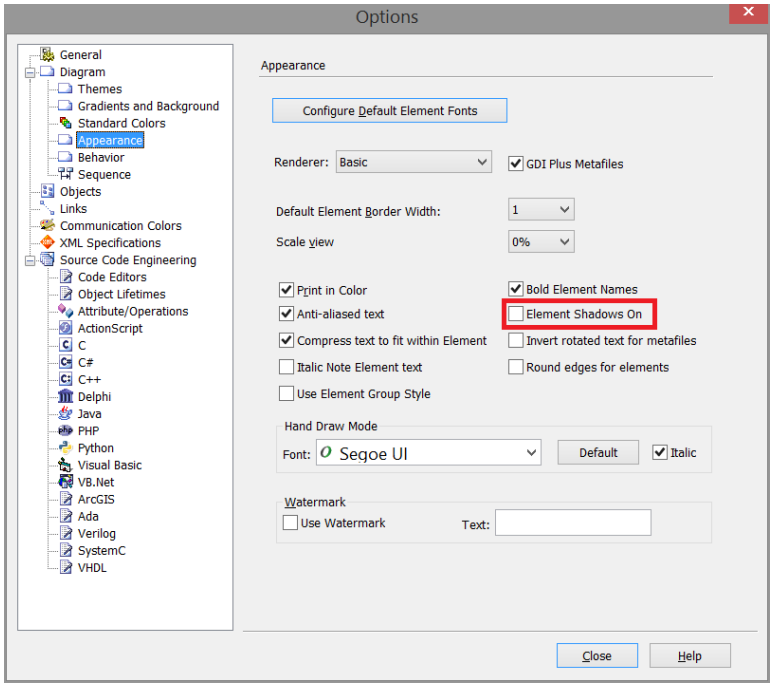


Figure 4.3: Our choice of the standard appearance for model elements

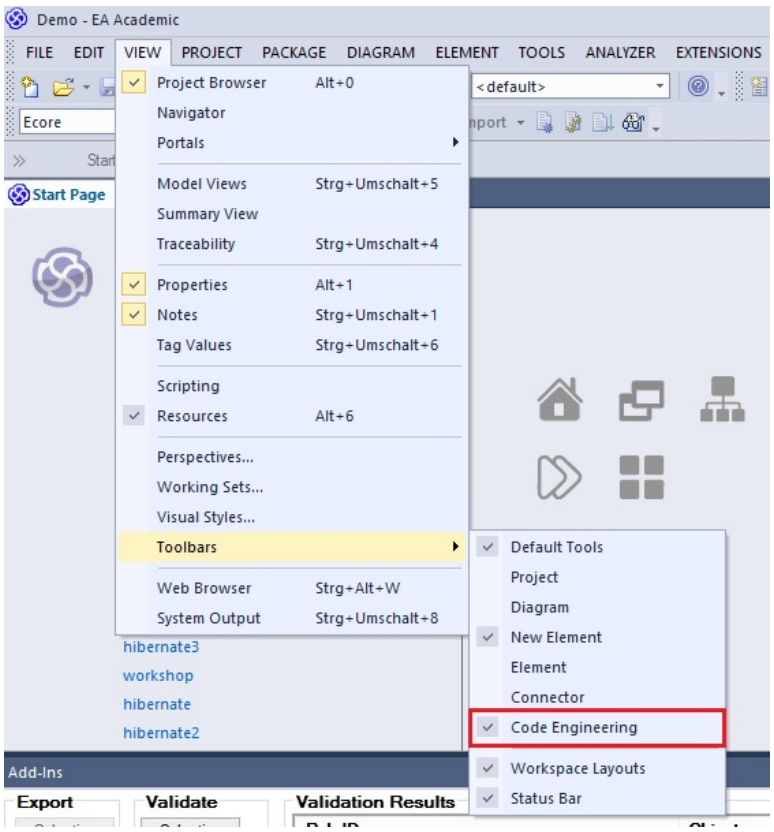


Figure 4.4: Open the *Code Engineering* toolbar

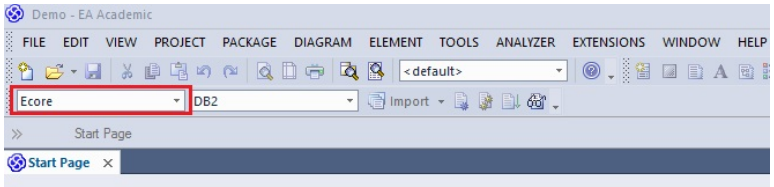


Figure 4.5: Make sure you set the standard language to `Ecore`

In your EA "workspace" (actually referred to as an *EA project*), take a careful look at the project browser: The root node `Demo` is called a *model* in EA lingo, and is used as a container to group a set of related *packages*. In our case, `Demo` contains a single package `org.moflon.demo.doublelinkedlist`. An EA project however, can consist of numerous models that in turn, group numerous packages.

Now switch back to your Eclipse workspace and note the two nodes named `Specifications` and `org.moflon.demo` (Figure 4.6).

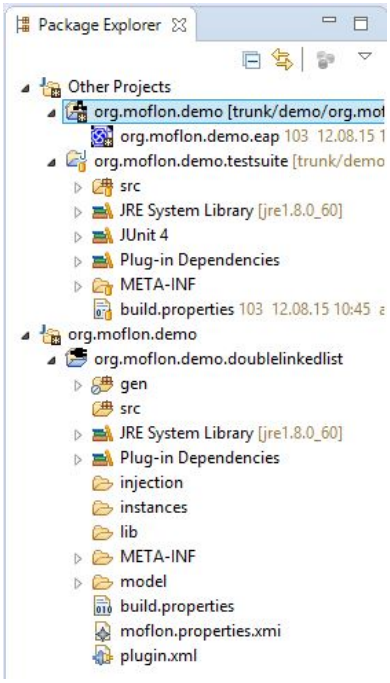


Figure 4.6: Project structure

These nodes, used to group related *Eclipse projects* in an Eclipse workspace, are called *working sets*. The working set `Specifications` contains all *metamodel projects* in a workspace. Your metamodel project contains a single EAP (EA project) file and is used to communicate with EA and initiate code generation by simply pressing `F5` or choosing `Refresh` from the context menu. In our case, `Specifications` should contain a single metamodel project `org.moflon.demo` containing our EA project file `org.moflon.demo.eap`.

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

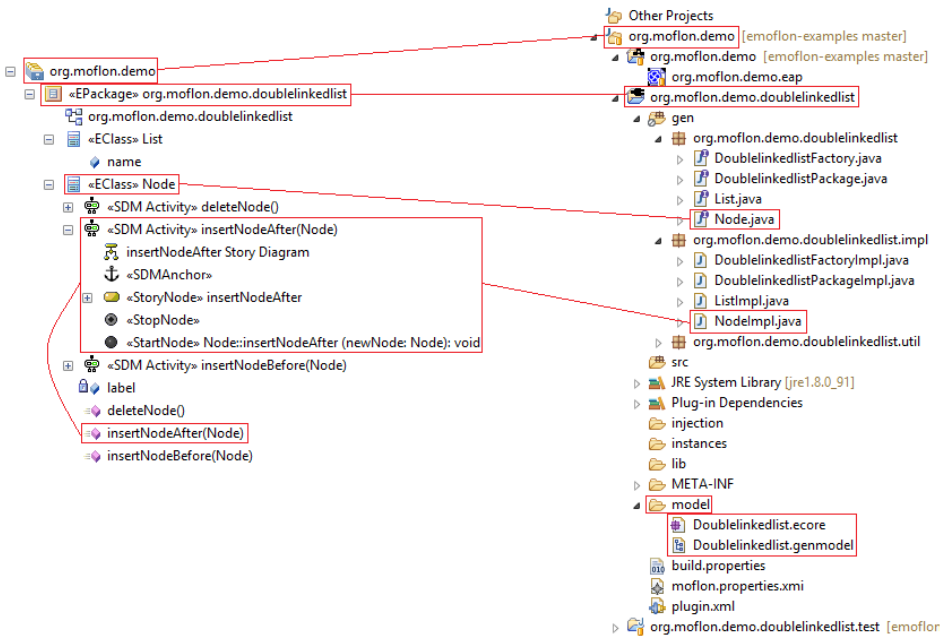


Figure 4.7: Mapping between artefacts in EA and Eclipse

These projects, however, are of a different nature than, for example, metamodel projects or normal Java projects. These are called *repository projects*. A *nature* is Eclipse lingo for "project type" and is visually indicated by a corresponding nature icon on the project folder. Our metamodel projects sport a neat little class diagram symbol. Repository projects are generated automatically with a certain project structure according to our conventions.

The `model` subfolder in the Eclipse package explorer is probably the most important as it contains the *Ecore model* for the project. Ecore is a metamodeling language that provides building blocks such as *classes* and *references* for defining the static structure (concepts and relations between concepts) of a system. This folder also contains a *genmodel*, the second model required by the Eclipse Modeling Framework (EMF) to generate Java code.

Looking back to Figure 4.7, realize that it also depicts how the class `Node` in the EA model is mapped to the Java interface `Node`. Double-click `Node.java` and take a look at the methods declared in the interface. These correspond directly to the methods declared in the modeled `Node` class.

As indicated by the source folders `src`, `injection`, and `gen`, we advocate a clean separation of hand-written (should be placed in `src` and `injection`) and generated code (automatically in `gen`). As we shall see later in the handbook, hand-written code can be integrated in generated classes via

injections. This is sometimes necessary for small helper functions.

Have you noticed the methods of the `Node` class in our EA model? Now hold on tight—each method can be *modeled* completely in EA and the corresponding implementation in Java is generated automatically and placed in `NodeImpl.java`. Just in case you didn't get it: The behavioural or dynamic aspects of a system can be completely modeled 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 *Story Diagrams* or simply *SDM*s, are placed in *SDM Containers* named according to the method they implement. For instance, ≪`SDM Activity`≫ `insertNodeAfter SDM` for the method `insertNodeAfter(Node)` as depicted in Figure 4.7. We'll dedicate Part III of the handbook to understanding why SDMs are so crazily cool!

To recap all we've discussed, let's consider the complete workflow as depicted in Figure 4.8. We started with a concise model in EA, simple and independent of any platform specific details (1). Our EA model consists not only of static aspects modelled as a class diagram (2), but also of dynamic aspects modelled using SDM (3). After exporting the model and code generation (4), we basically switch from *modelling* to *programming* in a specific general purpose programming language (Java). On this lower *level of abstraction*, we can flesh out the generated repository (5) if necessary, and mix as appropriate with hand-written code and libraries. Our abstract specification of behaviour (methods) in SDM is translated to a series of method calls that form the body of the corresponding Java method (6).

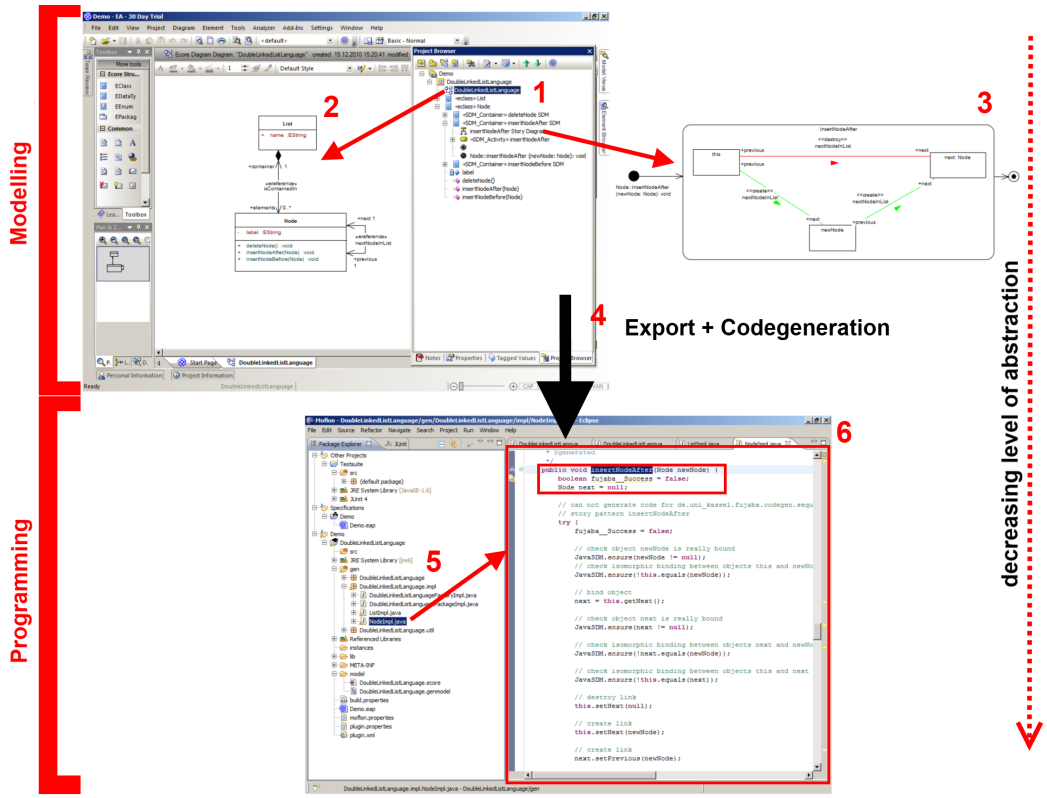


Figure 4.8: Overview

5 Generated code vs. hand-written code

Now that you've worked through the specifics of your syntax, lets have a brief discussion on code generation.

The Ecore model is used to drive a code generator that maps the model to Java interfaces and classes. The generated Java code that represents the model is often referred to as a repository. This is the reason why we refer to such projects as repository projects. A repository can be viewed as an adapter that enables building and manipulating concrete instances of a specific model via a programming language such as Java. This is why we indicate repository projects using a cute adapter/plug symbol on the project folder.

If you take a careful look at the code structure in `gen` (Figure 5.1), you'll find a `FooImpl.java` for every `Foo.java`. Indeed, the subpackage `.impl` contains Java classes that implement the interfaces in the parent package. Although this might strike you as unnecessary (why not merge interface and implementation for simple classes?), this consequent separation in interfaces and implementation allows for a clean and relatively simple mapping of Ecore to Java, even in tricky cases such as multiple inheritance (allowed and very common in Ecore models). A further package `.util` contains some auxiliary classes such as a factory for creating instances of the model.

If this is your first time of seeing generated code, you might be shocked at the sheer amount of classes and code generated from our relatively simple model. You might be thinking: "Hey – if I did this by hand, I wouldn't need half of all this stuff!" Well, you're right and you're wrong. The point is that an automatic mapping to Java via a code generator scales quite well.

This means for simple, trivial examples (like our double linked list), it might be possible to come up with a leaner and simpler Java representation. For complex, large models with lots of mean pitfalls however, this becomes a daunting task. The code generator provides you with years and *years* of experience of professional programmers who have thought up clever ways of handling multiple inheritance, an efficient event mechanism, reflection, consistency between bidirectionally linked objects, and much more.

A point to note here is that the mapping to Java is obviously not unique. Indeed there exist different standards of how to map a modelling language to a general purpose programming language such as Java. As previously mentioned, we use a mapping defined and implemented by the Eclipse Modelling Framework (EMF), which tends to favour efficiency and simplicity over expressiveness and advanced features.

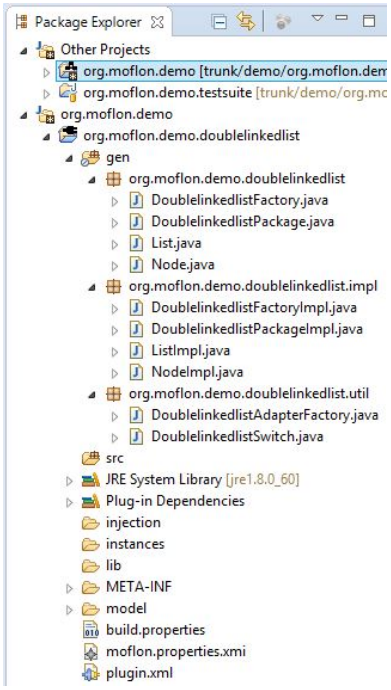


Figure 5.1: Package structure of generated code (`gen`)

6 Conclusion and next steps

Congratulations – you've finished Part I! If you feel a bit lost at the moment, please be patient. This first part of the handbook has been a lot about installation and tool support, and only aims to give a very brief glimpse at the big picture of what is actually going on.

If you enjoyed this section and wish to get started on the key features of eMoflon, Check out Part II[8]! There we will work through a hands-on, step-by-step example and cover the core features of eMoflon.

We shall also introduce clear and simple definitions for the most important metamodelling and graph transformation concepts, always referring to the concrete example and providing references for further reading.

If you're already familiar with the tool, feel free to pick and choose individual parts that are most interesting to you. Check out Story Driven Modeling (SDMs) in Part III[9], or Triple Graph Grammars (TGGs) in Part IV[10]. We'll provide instructions on how to easily download all the required resources so

[8]Download: https://emoflon.github.io/eclipse-plugin/beta/handbook/part2.pdf
[9]Download: https://emoflon.github.io/eclipse-plugin/beta/handbook/part3.pdf
[10]Download: https://emoflon.github.io/eclipse-plugin/beta/handbook/part4.pdf

you can jump right in. For further details on each part, refer to Part 0[11]. Cheers!

[11]Download: https://emoflon.github.io/eclipse-plugin/beta/handbook/part0.pdf

An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part II: Ecore

For eMoflon Version 2.16.0

File built on 21st September, 2016

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

The eMoflon team
Darmstadt, Germany (September 2016)

Contents

Part II:

Leitner's Learning Box

URL of this document: https://emoflon.github.io/eclipse-plugin/beta/handbook/part2.pdf

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 until 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 Figure 0.1, 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 returned to the first partition, regardless of how far in the box they have progressed.

These "rules" are depicted by the green and red arrows in Figure 0.1. The basic idea is to repeat difficult cards as often as necessary and not waste time on easy cards which are only repeated now and then to keep them in memory. The increasing size of the partitions represent 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

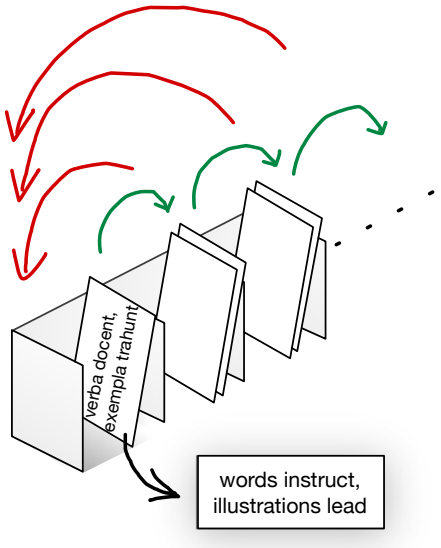


Figure 0.1: Possible *concrete syntax* of a Leitner's learning box

A learning box is an interesting system, because it consists clearly of a static structure (the box, partitions with varying sizes, and cards with two sides of content) and a set of rules that describe the dynamic aspects (behaviour) of the system. In this Part II of the eMoflon handbook, 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. We'll begin with a short discussion, then develop the box's abstract syntax, learn about eMoflon's validation, create some dynamic instances of the box, use injections to integrate a hand-written implementation of a method with generated code from the model, and get to know a small GUI that will allow us to take the implemented learning box for a spin.

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

such sytems. 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 what the building blocks of your language "look" like. Will your language be textual? Visual? This representation 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, Figure 0.1 can be viewed as a possible concrete syntax. As we are building a learning box as a software system however, our actual concrete syntax will be composed of GUI elements like buttons, drop-down menus and text fields.

Concrete Syntax

Irrespective of what 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 often use a *type graph* that declaratively defines what types and relations constitute a language.

Grammar

Graph Grammar

Type Graph

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 - 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 (OO) 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.

Abstract Syntax

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*,, which is 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).

Metamodel

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 many 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 con-

Unification

Modelling Language

forms to some *meta-metamodel* which in turn 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. Alternate modelling languages include MOF, UML and Kermeta.

Meta-metamodel
Meta-Language

Thinking back to our learning box, we can define the types and relations we want to allow. We want an entire box of flashcards where each card is contained within a partition, and each partition is contained within the box. Multiplicities are an example of static semantics that do not belong to the abstract syntax, but can nonetheless be expressed in a metamodel. An example could be that a card can only ever exist in one partition, or that a partition can have either one `next` partition, or none at all.

More complex constraints that cannot be expressed in a metamodel are usually specified using an extra *constraint language* such as the Object Constraint Language (OCL). This idea, however, is beyond the scope of this handbook. We'll stick to metamodels without an extra constraint language.

Constraint Language

In addition to its static structure, every system has certain dynamic aspects that describe the system's behaviour and how it reacts to external stimulus or evolves over time. 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 naturally fits quite nicely into the OO paradigm.

Dynamic Semantics

A short recap: We have learned that metamodelling starts with defining a suitable language. For the moment, we know that a language comprises a concrete syntax (how the language "looks"), 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 (i.e., can be typed according to the abstract syntax and must adhere to the static semantics).

Model

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

2 Abstract syntax and static semantics

The first step in creating any metamodel is defining the abstract syntax, also known as the type graph. This involves defining each class, its attributes, references, and method signatures.

If you completed the demo in Part I, your Eclipse workspace will look slightly different than ours depicted in the screenshots. In an effort to keep things as clear as possible, we have removed those files from our package explorer, but still recommend keeping them for future reference.

Additionally, if you're continuing from the demo, you can begin modelling this project in two different ways. You can either develop your metamodel in the same workspace as the demo, or create a new one. Either way, please note that the steps are exactly the same, but our project browsers in EA may not exactly match. This handbook has assumed you prefer the latter.

2.1 Getting started in EA

- To begin, navigate to "New Metamodel Project" (Figure 2.1) and start a new project named `LeitnersLearningBox` (Figure 2.2). Open the empty `.eap` file in EA.

Figure 2.1: "New Metamodel Project" button

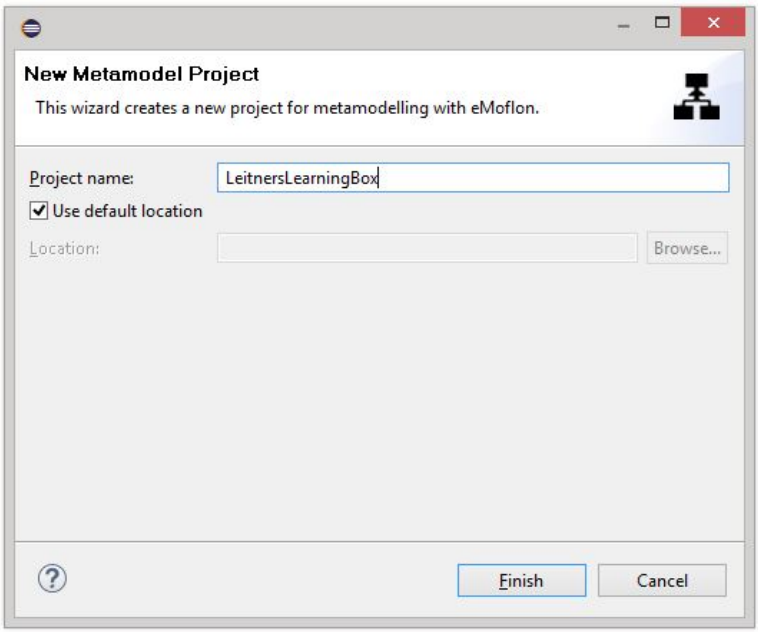


Figure 2.2: Starting a new visual project

- In EA, select your working set and press the "Add a Package" button (Figure 2.3).

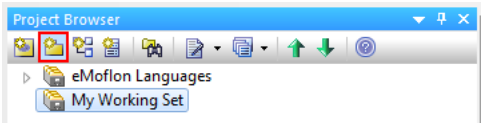


Figure 2.3: Add a new package to `MyWorkingSet`

- In the dialogue that pops up (Figure 2.4), enter `LearningBoxLanguage` as the name of the new package. In this case select `Package Only` and click `OK`. Later you can select `Create Diagram` to skip the next step.

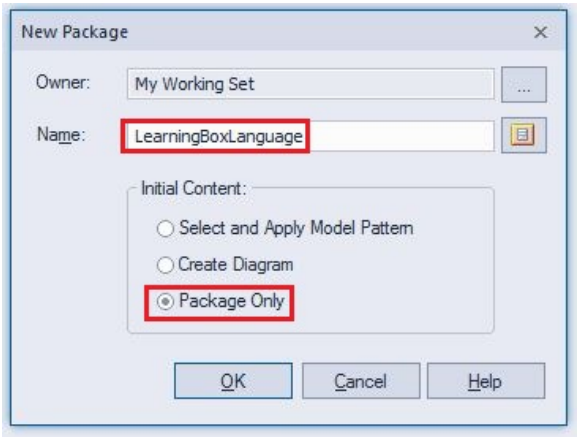


Figure 2.4: Enter the name of the new package

- Your `Project Browser` should now resemble Figure 2.5.

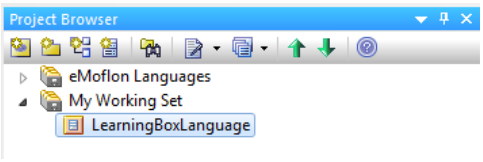


Figure 2.5: State after creating the new package

- Now select your new package and create a "New Diagram" (Figure 2.6).

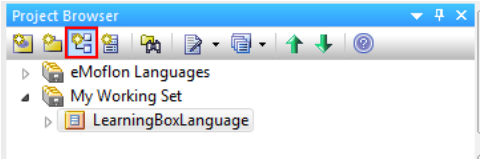


Figure 2.6: Add a diagram

- In the dialogue that appears (Figure 2.7), choose `eMoflon Ecore Diagrams` and press `OK`.

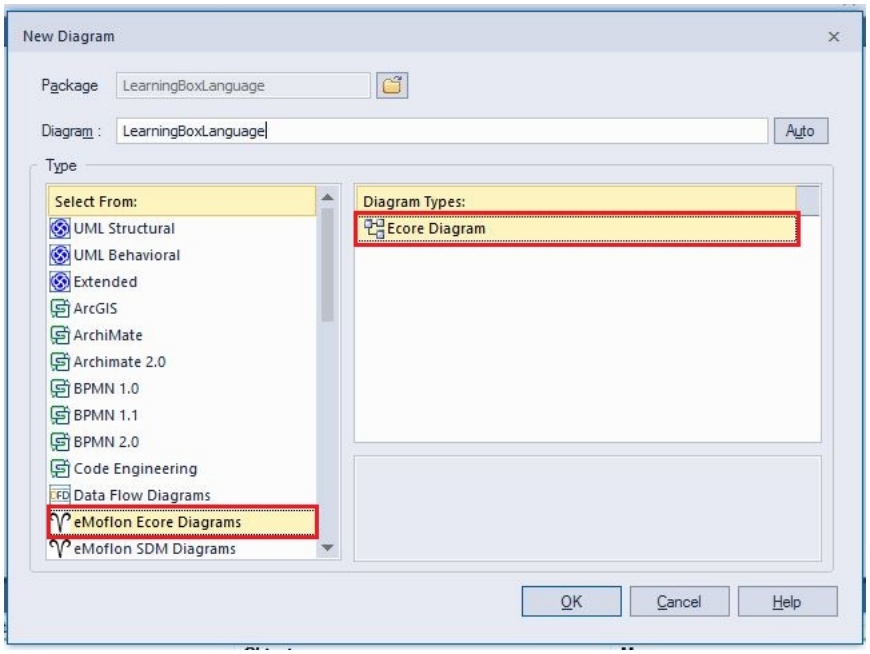


Figure 2.7: Select the ecore diagram type

- After creating the new diagram, your `Project Browser` should now resemble Figure 2.8. You'll notice that your `LearningBoxLanguage` package has been transformed into an EPackage.

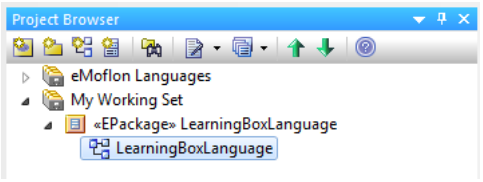


Figure 2.8: State after creating diagram

- You can now already export your project to Eclipse,[2] then refresh your `Package Explorer`. A new node, `My Working Set`[3] should have appeared containing your EPackage (Figure 2.9). You can see that a `LearningBoxLanguage.ecore` file has been generated, and placed in "model." This is your metamodel that will contain all future types you create in your diagrams.

[2] If unsure how to perform this step, please refer to Part I, Section 2.1

[3] If you do not have the two distinct nodes, ensure your "Top Level Elements" are set to `Working Sets`

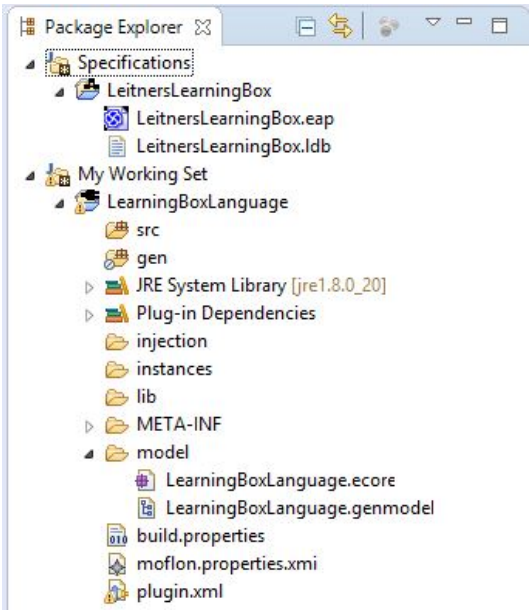


Figure 2.9: Initial export to Eclipse

- If you're interested in reviewing the overall project structure and the purposes of certain files and folders, read Section 4.1 from Part I. Otherwise, continue to the next section to learn how to declare classes and attributes.

2.2 Declaring classes and attributes

- Return to EA, and double-click your `LearningBoxLanguage` diagram to ensure it's open.

- There are two ways for you to create your first `EClass`. First, to the left of the workbench, a *Toolbox* containing the Ecore types available for metamodelling should have appeared (Figure 2.10).[4] Click on the `EClass` icon then somewhere in the diagram to create a new object. Alternatively, you can click in the diagram and press `space` to invoke the toolbox context menu, then select `EClass`.

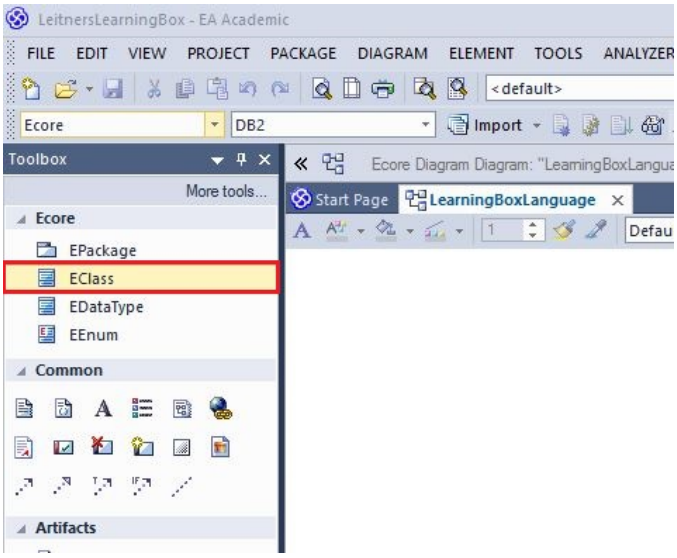


Figure 2.10: Create an EClass

- In the dialogue that pops-up, set `Box` as the name and click `OK` (Figure 2.11). This dialogue can always be invoked again by double-clicking the EClass, or by pressing `Alt` and single-clicking. It contains many other properties that we'll investigate later in the handbook. In general, a similar properties dialogue can be opened in the same fashion for almost every element in EA.

[4] If not, choose "Diagram/Diagram Toolbox" to show the current toolbox (Alt+ 5)

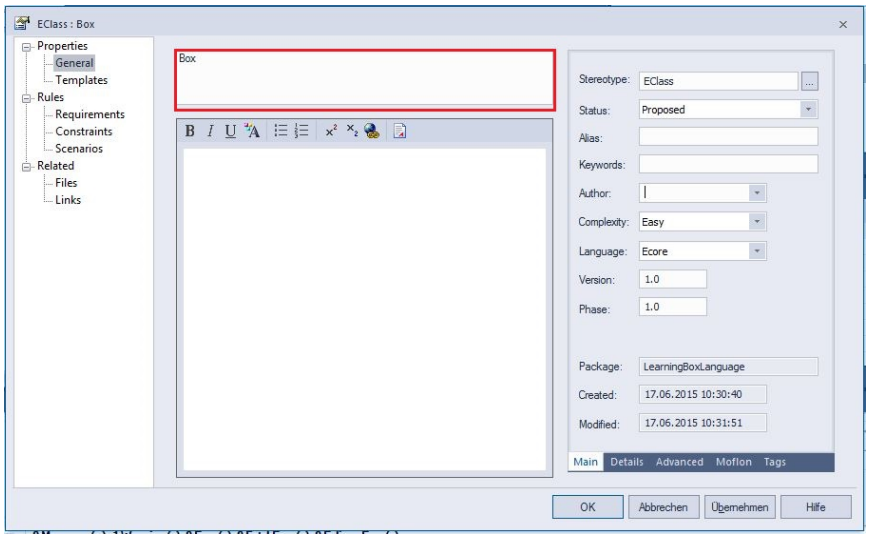


Figure 2.11: Edit the properties of an EClass

- After creating `Box`, your EA workspace should resemble Figure 2.12.

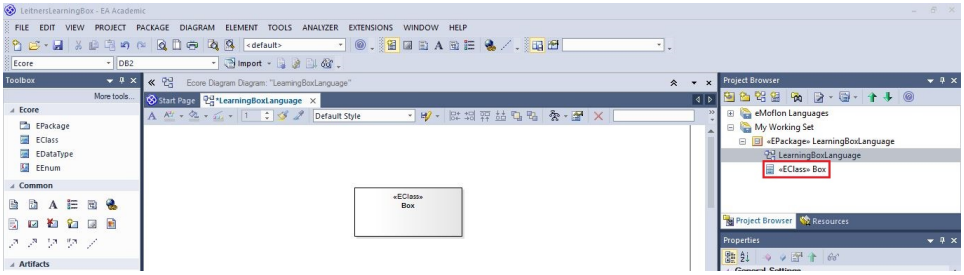


Figure 2.12: State after creating `Box`

- Now create the `Partition` and `Card` EClasses the same way, until your workspace resembles Figure 2.13. These are the main classes of your learning box metamodel.

- Lets add some attributes! Either right-click on `Box` to activate the context menu and choose "Features & Properties/Attributes.." (Figure 2.14), or press `F9` to open the editing dialogue.

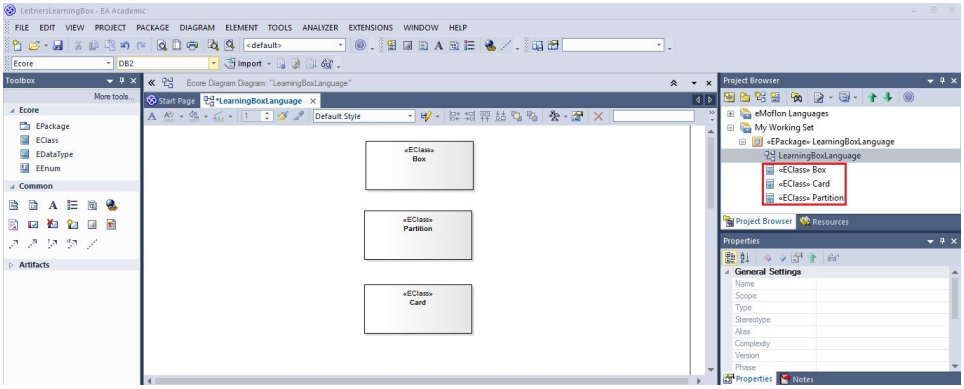


Figure 2.13: All EClasses for the metamodel

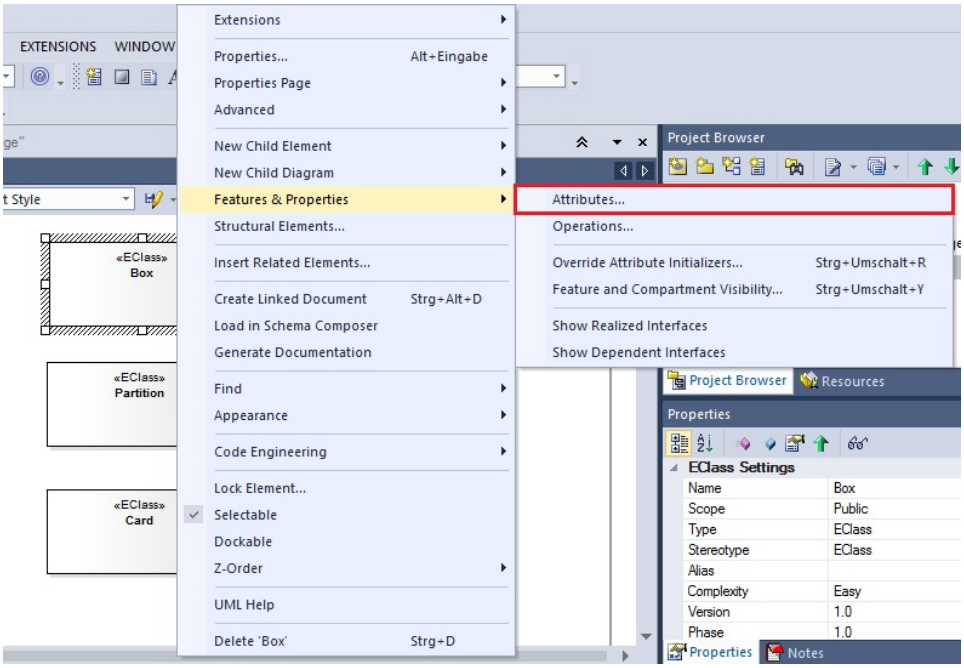


Figure 2.14: Context menu for an EClass

- Enter `name` as the name of the attribute, select `EString` as its type from the drop-down menu, and press `Close` (Figure 2.15). New attributes for the same EClass can be added by clicking on `New Attribute...`.

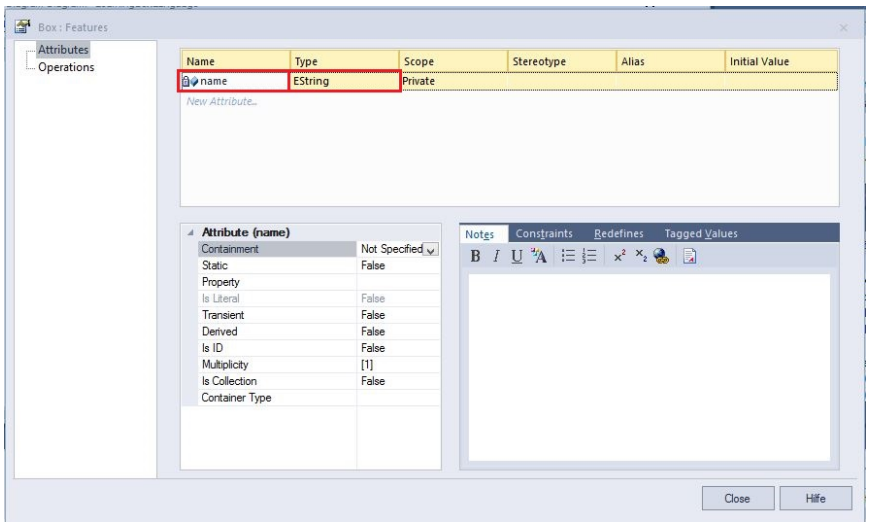


Figure 2.15: Adding attributes to an EClass

- Add the remaining attributes analogously to each EClass until your workspace resembles Figure 2.16.

- Save and export to Eclipse. After refreshing your workspace, your `.ecore` model can now be expanded as it includes every class and attribute from your metamodel. So far, so good!

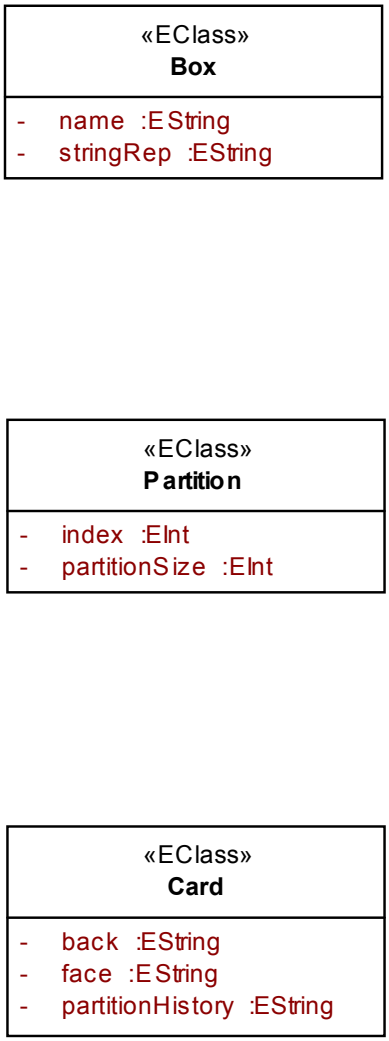


Figure 2.16: Main EClasses declared with their attributes

2.3 Connecting your classes

At this point, you've declared your types and their attributes, but what good are those if you can't connect them to each other? We need to create some *EReferences*! *EReference*

There are four properties that must be set in order to create an EReference: the target and source *Role*, *Navigability*, *Multiplicity* and *Aggregation*, all of which are declared differently in each syntax, but let's first review the concepts since they're basically the same.

A *Role* clarifies which way a reference is 'pointing.' In other words, the *source* and *target* roles determine the direction of the connection. *Role*

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. *Navigability*

The *Multiplicity* of a reference controls if the relation is mapped to a (Java) collection ('`*`', '`1..*`', '`0..*`'), or to a single valued class attribute (`'1'`, `'0..1'`). We'll explain this setting in detail later. *Multiplicity*

The *Aggregration* values of a reference can be either none or composite. Composite means that the current role is that of a *container* for the opposite role. You'll see in our example that `Box` is a container for several `Partition`s. 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. Conversely, non-composite (denoted by "none") means that the current role is not that of a container, and the rules for containment do not hold (in other words, the reference is a simple 'pointer'). *Aggregation* *Container*

Creating EReferences in EA

- A fundamental gesture in EA is *Quick Link*. Quick Link is used to create EReferences 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 (Figure 2.17). *Quick Link*

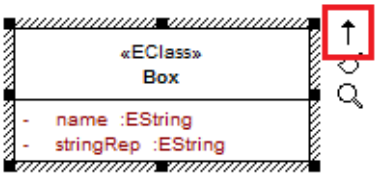


Figure 2.17: Quick Link is a central gesture in EA

- Click this black arrow and 'pull' to the element you wish to link to. To start, quick-link from `Box` to `Partition`. In the context menu that appears, select "Create Bidirectional EReference" (Figure 2.18).

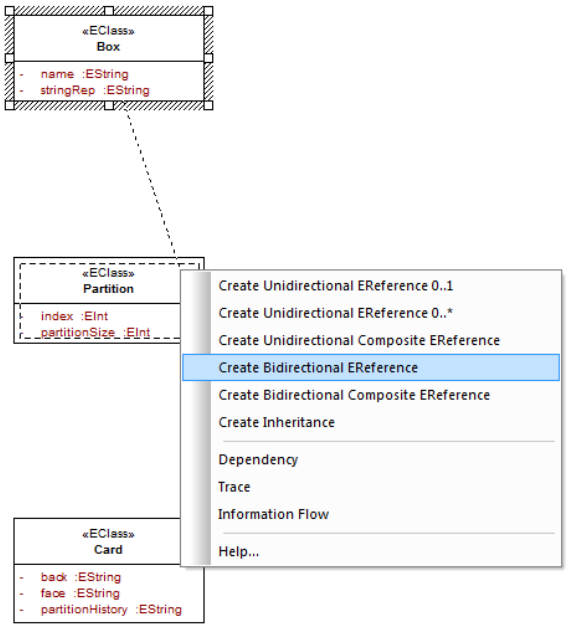


Figure 2.18: Create an EReference via Quick Link

- Double click the EReference to invoke a dialogue. In this window you can adjust all relevant settings. Feel free to leave the `Name` value blank - this property is only used for documentation purposes, and is not relevant for code generation.

- Within this dialogue, go to "Role(s)," and compare the relevant values in Figure 2.19 for the *source* end of the EReference (the `Box` role). As you can see, the default source is set to the EClass you linked from, while the default target is the EClass you linked to. In this window, do not forget to confirm and modify the `Role`, `Navigability`, `Multiplicity`, and `Aggregation` settings for the source as required. Repeat the process for the `Target Role`.

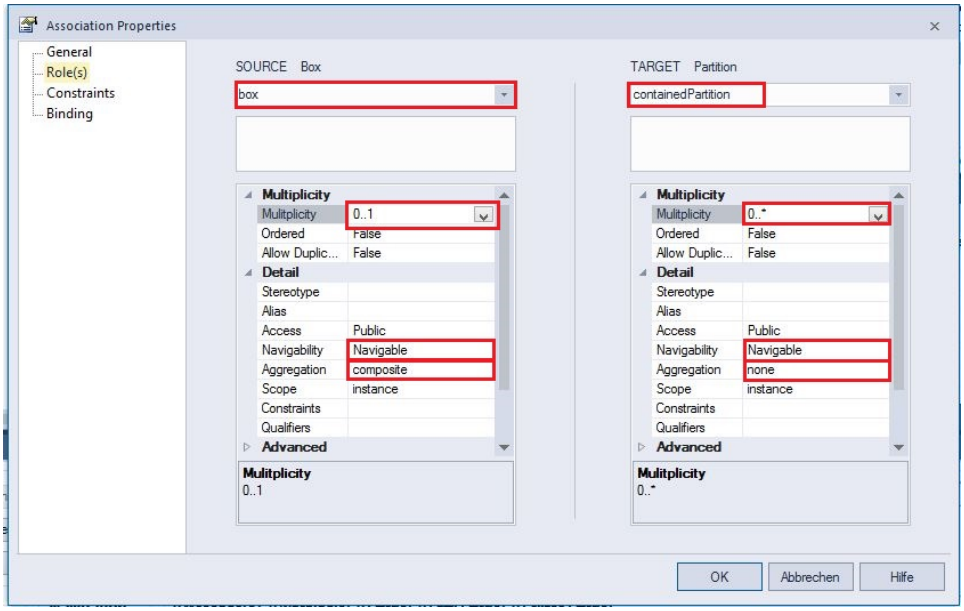


Figure 2.19: Properties for the source and target role of an EReference

To review these properties, the first value you edited was the role name. The `Navigation` value should have been automatically set to `Navigable`. Without these two settings, getter and setter methods will not be generated.

Next, you set the `Multiplicity` value. In your source role (`Box`), you have allowed the creation of up to one target (`Partition`) reference for every connected source (`box`). This means you could not have a single target connected to two sources (i.e., one partition that belongs to two boxes). In the target (`Partition`) role, you have specified that any source (in our case, `box`) can have any positive-sized number of targets. Figure 2.20 sketches this schematically.

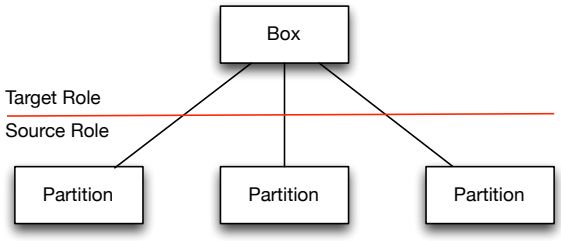


Figure 2.20: The target and source roles of Leitner's Learning Box

Finally, you set the `Aggregation` value. In this case, `box` is a container for `Partitions`, and `containedPartition` is consequently not.

- Take a moment to review how the `Aggregration` settings extend the `Multiplicity` rules. If you've done everything right, your metamodel should now resemble Figure 2.21, with a single *bidirectional EReference* between `Box` and `Partition`.

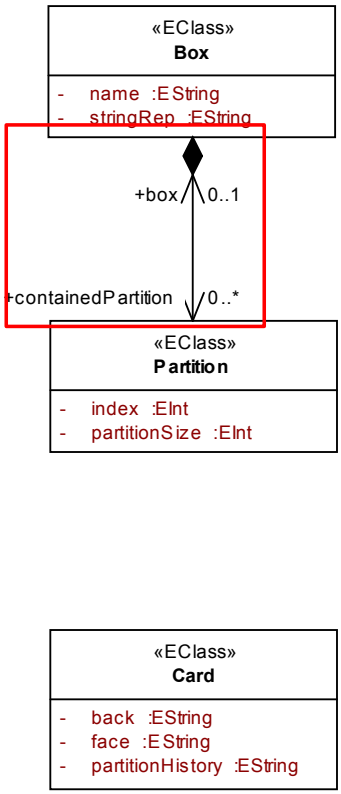


Figure 2.21: `Box` contains `Partitions`

- Following the same process, create two unidirectional self-EReferences for `Partition` (for next and previous `Partitions`), and a second bidirectional composite EReference[5] between `Partition` and `Card` (Figure 2.22).

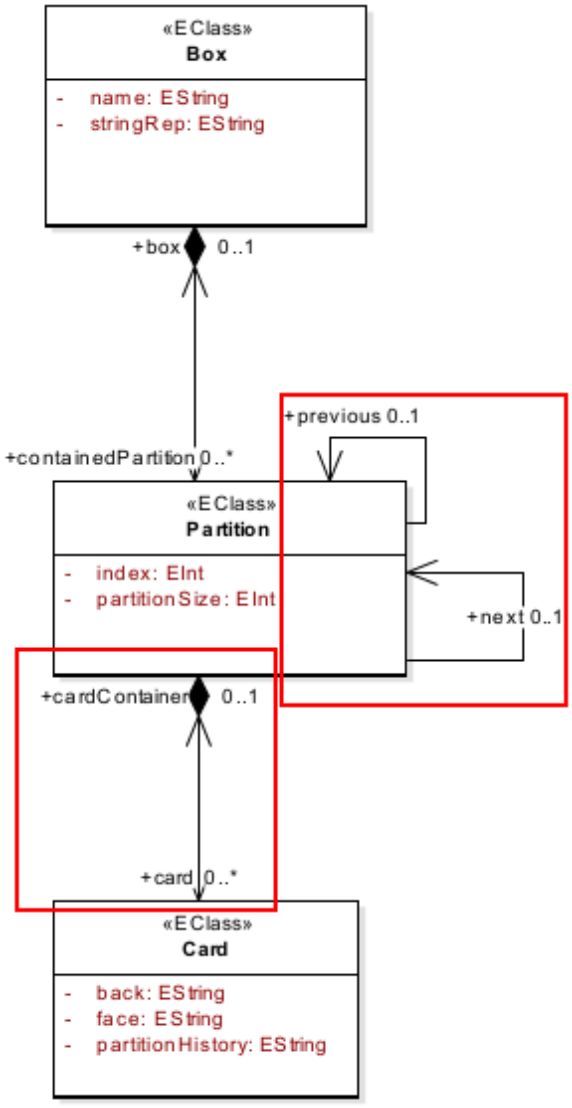


Figure 2.22: All relations in our metamodel

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

- You'll notice that the connection between `Card` and `Partition` is similar to that between `Partition` and `Box`. This makes sense as a partition should be able to hold an unlimited amount of cards, but a card can only belong to one partition at a time.

- Export your diagram to Eclipse and refresh your workspace. Your Ecore metamodel file in "model/LearningBoxLanguage.ecore" should now resemble Figure 2.23.

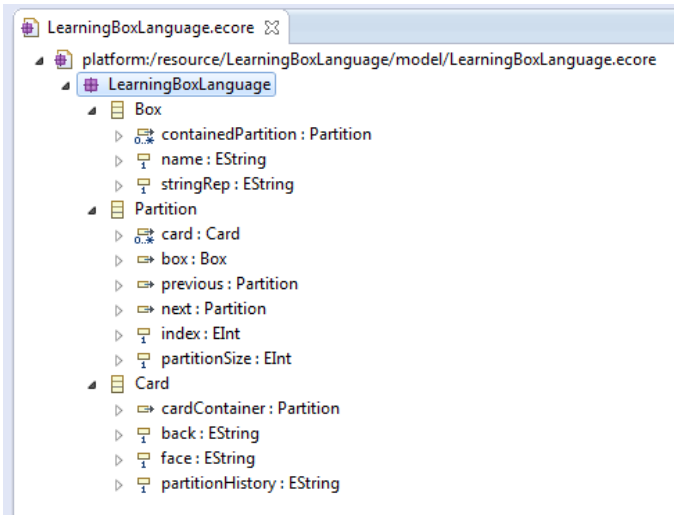


Figure 2.23: Refreshed Ecore file with all EReferences

- All the required attributes and references for your learning box have now been set up.

2.4 Method Signatures

To finish defining our types, lets define the *signatures* of some operations that they'll eventually support.

Operation Signature

- Select `Partition` and either right-click to invoke the context-menu (Figure 2.24) and choose "Features & Properties/Operations.." or simply press `F10`.

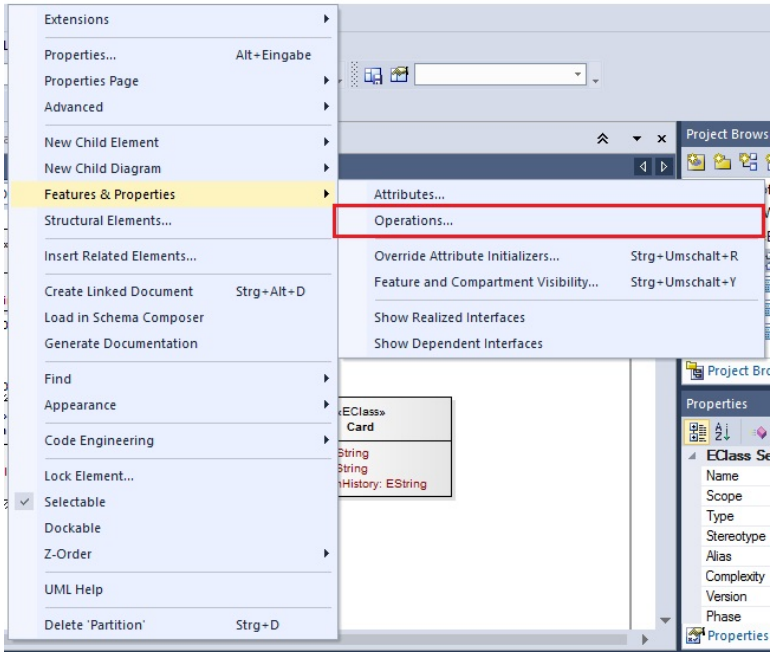


Figure 2.24: Add an operation

- In the dialogue that pops-up (Figure 2.25), enter `empty` as the `Name` of the operation and `void` as the `Return Type`.

- In the same dialogue, click on `New Operation...` to add a second operation, `removeCard`, and edit the values as seen in Figure 2.26. Notice that the `Return Type` can be chosen by either the drop-down menu, or via direct typing. For types you've established in the metamodel (e.g. `Card`) you have to use '`Select Type...`' from the drop-down menu.

  Very important: Non-primitive types *must* be chosen via '`Select Type...`' in the drop-down menu. It allows you to browse for the corresponding elements in your project. Simply typing them won't work!

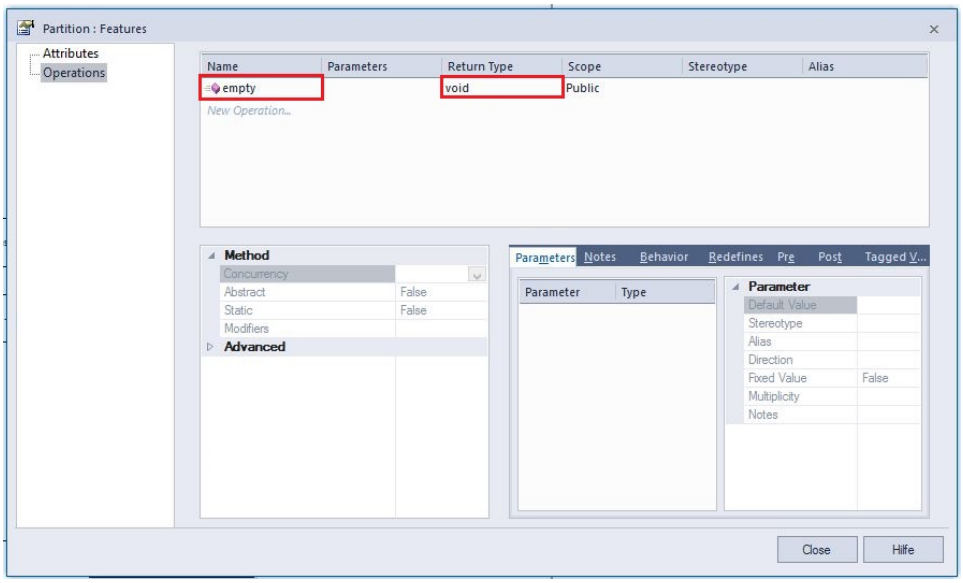


Figure 2.25: EClass properties editor

- Parameters can be added by selecting `Parameters` and completing the dialouge (Figure 2.26). Please remember that you must also use either the drop-down menu, or direct typing to select the type or else validation will fail.
- Repeat this process for the `check` operation (with the two parameters `card:Card, guess:EString`) that returns an `EBoolean`.
- If you've done everything right, your dialogue should now contain three methods - `check`, `empty`, and `removeCard` - each with the corresponding parameters and return types in Figure 2.27.
- Add all operations analogously for `Box` and `Card` until your metamodel closely resembles Figure 2.28.[6]
- To finish, export the metamodel for code generation in Eclipse, and examine the model once again. Each signature should have appeared in their respective EClass.

[6]Please note that names of parameters may not be displayed by default in EA

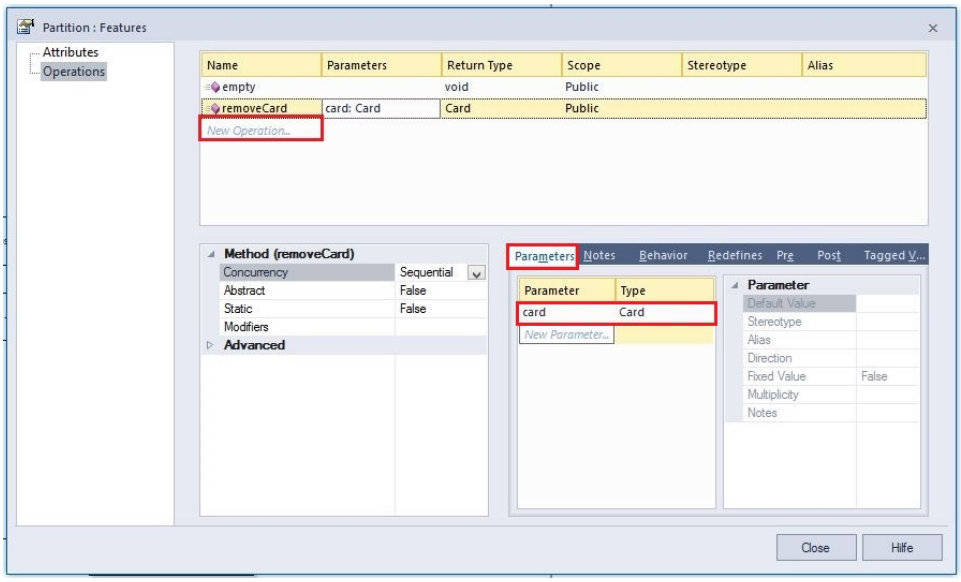


Figure 2.26: Parameters and return type

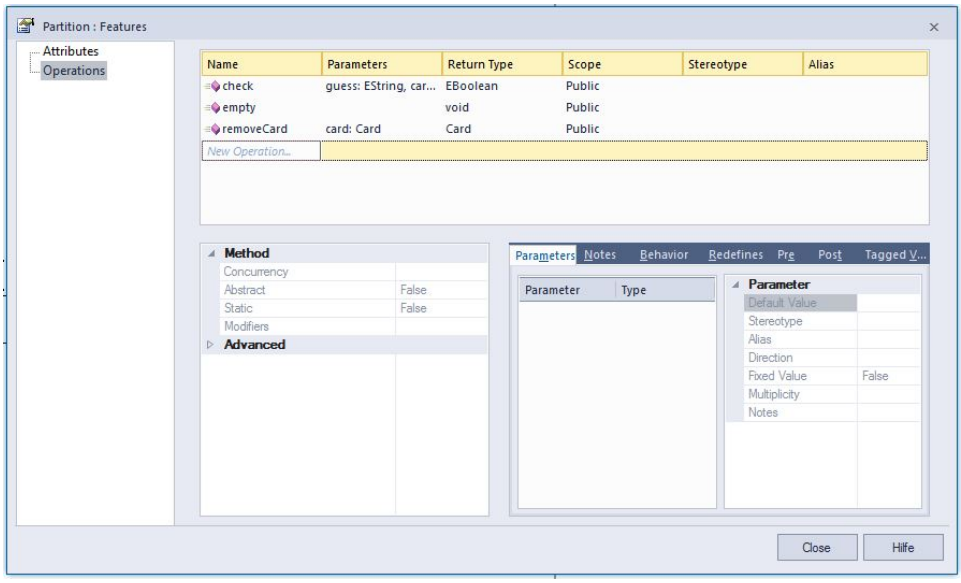


Figure 2.27: All operations in `Partition`

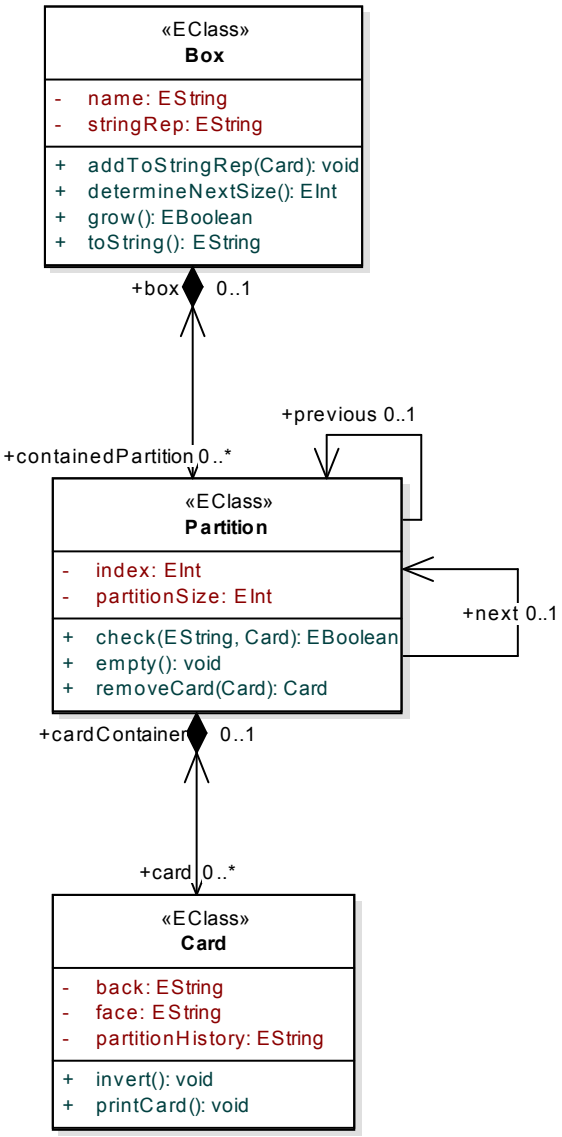


Figure 2.28: Complete metamodel for our learning box

2.5 eMoflon validation support in EA

Our EA extension provides rudimentary support for validating your metamodel. Validation results are displayed and, in some cases, even "quick fixes" to automatically solve the problems are offered. In addition to reviewing your model, the validation option automatically exports the current model to your eclipse workspace if no problems were detected.

- If not already active, make the eMoflon control panel visible in EA by choosing "Extensions/Add-In Windows". This should display a new output window, as depicted in Figure 2.29. Many users prefer this interface as it provides quick access to all of eMoflon's features, as opposed to the drop down menu under "Extensions/MOFLON::Ecore Addin" which only offers limited functionality.

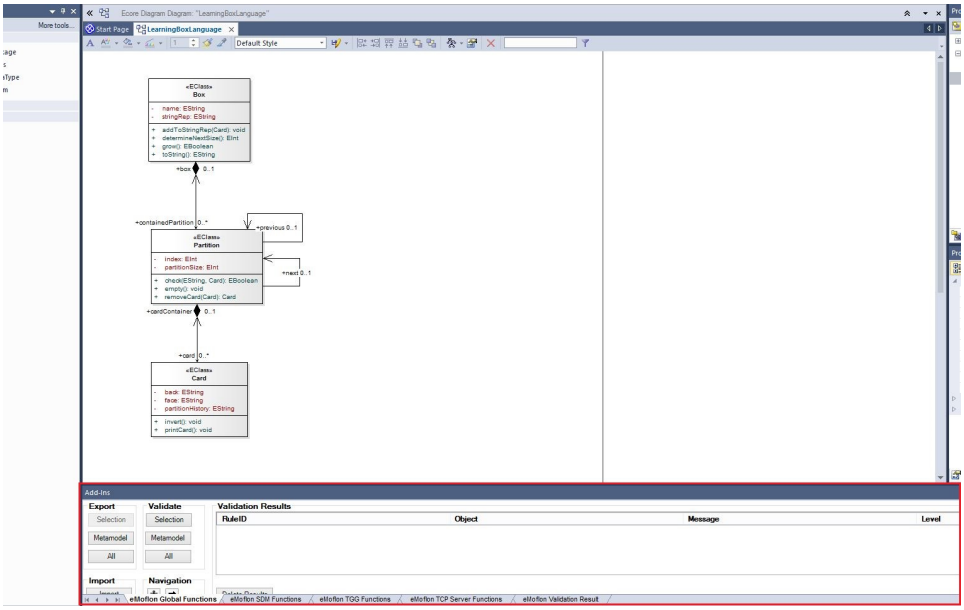


Figure 2.29: Activating the validation output window

- To start the validation, choose "Validate all" in the "Validate" section of the control panel (Figure 2.30). If you haven't made any mistakes while modelling your `LearningBoxLanguage` so far, the validation results window should remain empty, indicating your metamodels are free of errors.

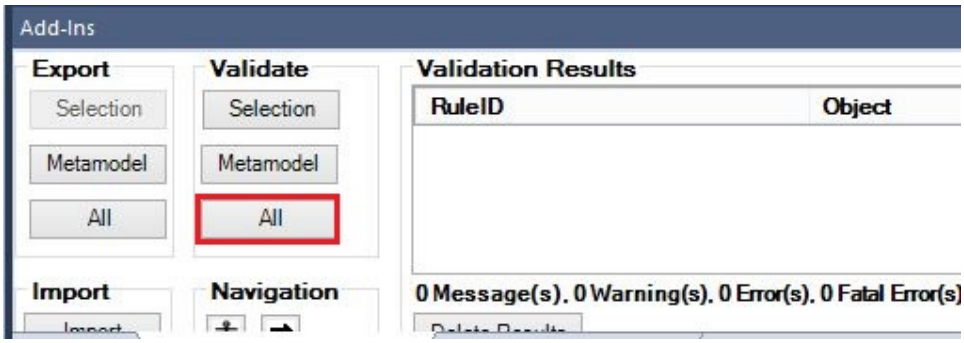


Figure 2.30: Starting the validation

If an error did appear, the validation system attempts to suggest a "Quick Fix." Why don't we examine the validation and quick fix features in detail? Let's add two small modelling errors in `LearningBoxLanguage`.

- Create a new EClass in the `LearningBoxLanguage` diagram. You can retain the default name `EClass1`. Let's assume you wish to delete this class from your metamodel.
- Select the rouge class in the diagram and press the `Delete` button on your keyboard. Note that this only deleted it from the current diagram and `EClass1` still exists in the project browser (and thus in your metamodel).
- Run the validation test, and notice the new `Information` message in the validation output (Figure 2.31).

Figure 2.31: Validation information error: element still exists

This message informs you that `EClass1` is not on any diagram, and seeing as it is still in the metamodel, that this *could* be a mistake. As you can see, just pressing the `Delete` button is not the proper way of removing an EClass from a metamodel - It only removes it from the current diagram![7]

[7] Deleting elements properly and other EA specific aspects are discussed in detail in Part VI: Micellaneous

- Suppose you were inspecting a different diagram, and were not on the current screen. To navigate to the problematic element in the `Project Browser`, click *once* on the information message.
- To check to see if there are any quick fixes available, *double*-click the information message to invoke the "QuickFix" dialogue. In this case, there are two potential solutions - add the element to the current diagram or (properly) delete the element from the metamodel (Figure 2.32). Since the latter was the original intent, click `Ok`.

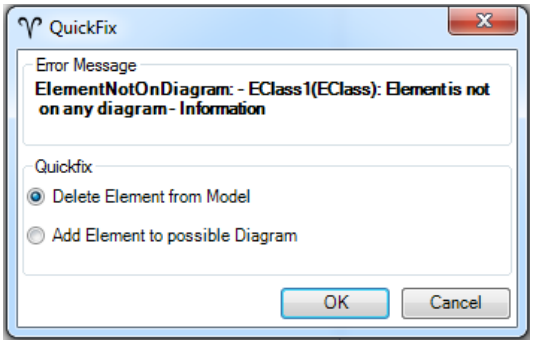


Figure 2.32: Quick fix for elements that are not on any diagram

- `EClass1` should now be correctly removed from your metamodel. Your metamodel should now be error-free again as indicated by the validation output window.
- To make an error that leads to a more critical message than "information," double-click the navigable EReference end `previous` of the EClass `Partition`, and delete its role name as depicted in Figure 2.33. 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. Close all windows, then single click on the error once to open the relevant diagram and highlight the invalid element on the diagram. Double click the error to view the quick fix menu (Figure 2.34). 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).
- Given there are no automatic solutions, correct your metamodel manually by setting the name of the navigable EReference back to `previous`.

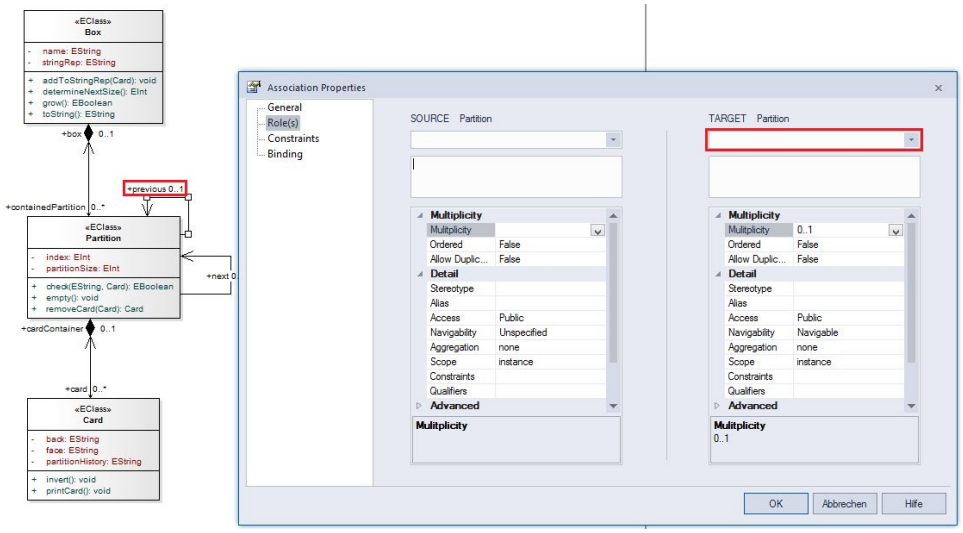


Figure 2.33: Deleting a navigable role name of an EReference

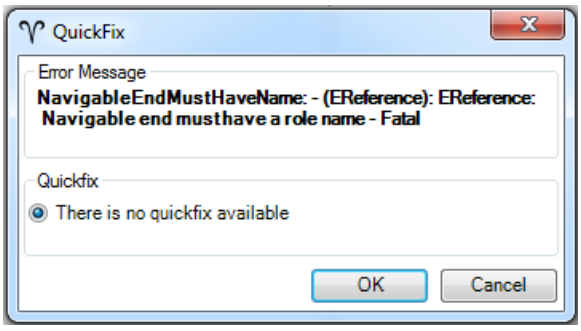


Figure 2.34: Fatal error after deleting a navigable role name

- Ensure that your metamodel closely resembles Figure 2.28 again, and that there are no error messages before proceeding.

As you may have noticed, eMoflon distinguishes 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 is incorrectly set or missing.

Eclipse Error:
Display error messages produced by our Eclipse plugin after an unsuccessful attempt to generate code.

2.6 Reviewing your metamodel

Before moving on, lets take a step back and review what we have accomplished. We have modelled a `Box` that can contain an arbitrary amount of `Partitions`. A `Partition` in the `Box` has a `next` and `previous` `Partition` that can be set. Finally, `Partitions` contain `Cards`.

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

The main method of the learning box is `Partition::check`, which takes a `Card` and the user's `EString` guess, and returns a `true` or `false` value.

A `Partition` can also `remove` a specific `Card`, or empty itself of all existing `Cards`. Last but not least, a `Partition` has a `partitionSize` to indicate how many cards it should hold. Too many cards in the first partition could indicate that not enough time has been dedicated to learning the terms. Too many near the end of the box could show that the vocabulary set is too easy, and probably already mastered.

A `Card` contains the actual content to be learned as a question on the card's `face` and the answer on the card's `back`. `Cards` also maintain a `partitionHistory`, which can be used to keep track of how often a `Card` has been answered incorrectly. This may indicate how difficult the `Card` is for a specific user, and remind them to spend more time on it. 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).

Examine the generated files in "gen", especially the default implementation for those methods that currently just throw an `OperationNotSupported` exception. We shall see in later parts of this handbook how our code generator supports injecting hand-written implementation of methods into generated methods and classes. With eMoflon however, we can actually model a large part of the dynamic semantics with ease, and only need to implement small helper methods (such as those for string manipulation) by hand.

If you have had problems with this section, and, despite firmly believing everything is correct, things *still* don't work, feel free to contact us at: contact@emoflon.org.

3 Creating instances

Before diving into modelling dynamic behaviour in Part III, let's have a brief look at how to create a concrete *instance* of your metamodel in Eclipse.

In the following section, 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 those that are expressed *in* such a language (instance models of the metamodel).

- To create an instance model, navigate to the generated `model` folder in your `LearningBoxLanguage` project. Double-click the `LearningBoxLanguage.ecore` model to invoke the *Ecore model editor*.

- To create a concrete instance of the metamodel, you must select an EClass that will become the root element of the new instance. For our example, right-click `Box`, and navigate to "Create Dynamic Instance" from the context menu, as depicted in Figure 3.1.

- A dialogue should appear asking where the instance model file should be persisted. Save your instances according to convention in a folder named "instances," which is automatically created in every new repository project. Last but not least, enter `Box.xmi` as the name of the instance model (Figure 3.2).

- Press `Finish`, and a generic model editor should open for your new instance model. This editor works just like the previous Ecore model editor except it's "generic," meaning it allows you to create and edit an instance of *any* metamodel, not only instances of Ecore (i.e., metamodels).

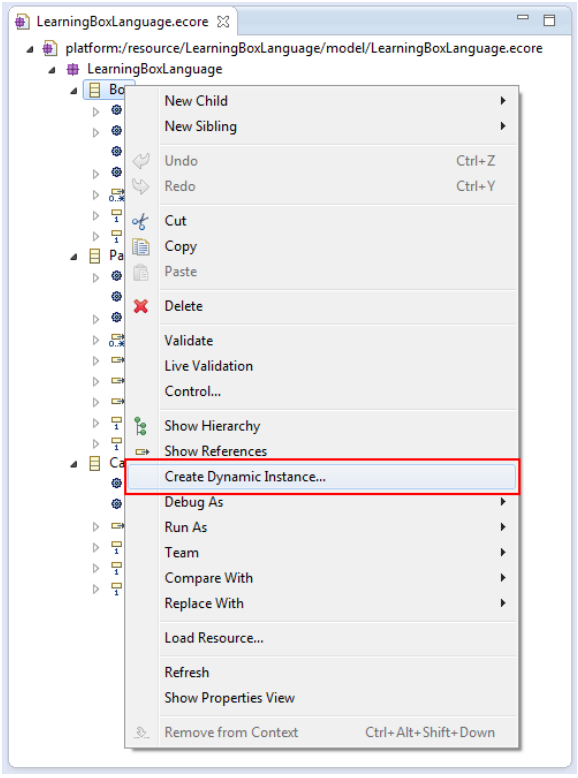


Figure 3.1: Context menu of an EClass in the Ecore editor

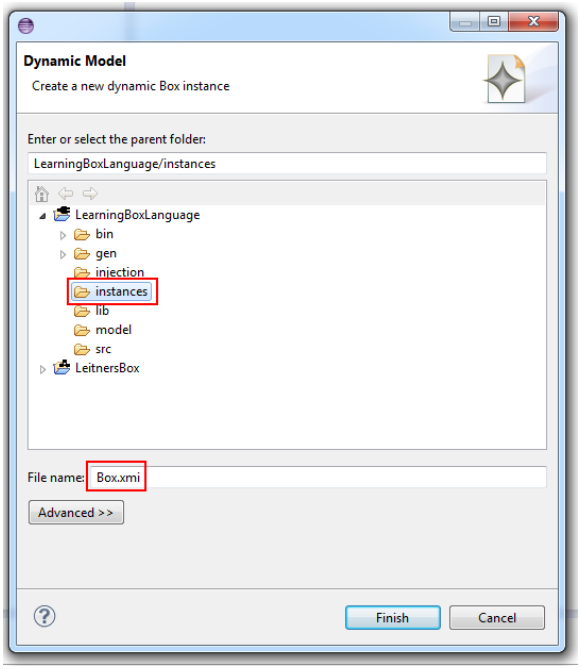


Figure 3.2: Dialogue for creating a dynamic model instance

- You can populate your instance by adding new children or siblings via a right-click of an element to invoke the context-menu depicted in Figure 3.3. Note that EMF supports you by respecting your metamodel, and reducing the choice of available elements to valid types only.[8]

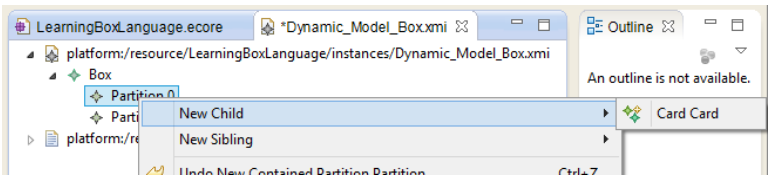


Figure 3.3: Context menu for creating model elements

- Let's try building a vocabulary set. Fill your box with three partitions, with two cards in each. Save your model by pressing `Ctrl+S` and confirm the save by closing it, then reloading via a simple double-click.

- Double-click on one of the partitions to bring up the "Properties" tab in the window below the editor (Figure 3.4). Here you'll see the attributes you defined earlier in each EClass. The `Box`, `Next`, and `Previous` values are its (undefined) EReferences,[9] while `Index` is a partition's unique idenfitication value, and `Partition Size` is the recommended number of cards the partition should contain before being tested.

[8] This depends on the current context. Try it out!

[9] Please note that the editor tab capitilizes the first letter of each property

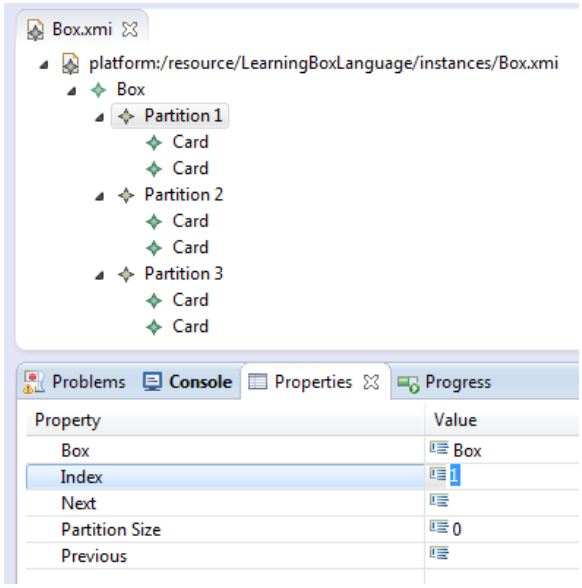


Figure 3.4: Change the `Index` value in the partition's property tab

- Pick a number - any one you like - and update each partition's `Index` value. This is their identification value! You'll notice that as soon as you press `Enter`, the values will be reflected in the `.xmi` tree.
- Now you need to set the `Next` and `Previous` EReferences which will make it possible to move cards through the box. Given that there are no partition before the first, set `Previous` values only for your second and third partition to that first partition. Similarly, only set the `Next` values for the first, and second partition to their respective 'next' partitions.
- In the same fashion, double click on each `Card` you created and modify their values. In particular, update the `Back` attribute to `One`. This is the value you'll be able to see from the partition and thus, the `.xmi` tree. You'll be experimenting with the `Face` attribute shortly, so provide a `1` value for that as well.
- Fill in the rest the cards with similar vocabulary-style words (such as two/2, three/3, ...) that you can 'test' yourself on (Figure 3.5). You now have a unique, customized learning box! Save your model, and ensure no errors exist before proceeding.

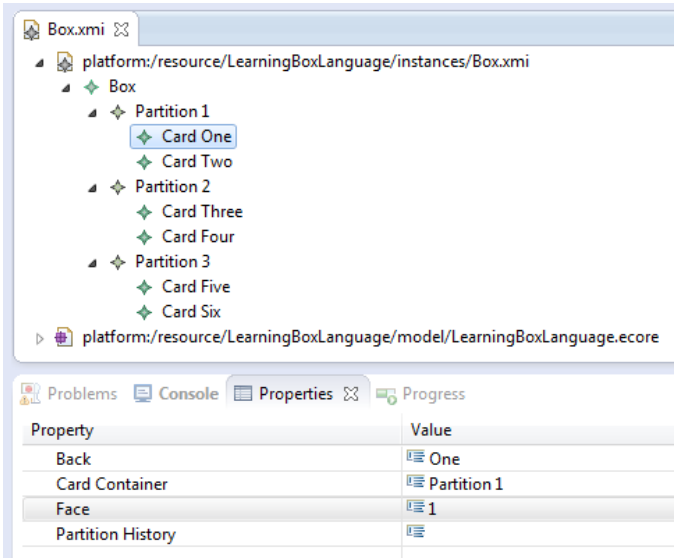


Figure 3.5: A vocabulary-style learning box

4 eMoflon's graph viewer

At this point, you should now have a small instance model. While Eclipse's built-in editor offers a nice tree structure and table to view and edit your model, wouldn't it be nice to have a visualisation? eMoflon offers a "Graph View" to do exactly that! You'll find that this is an effective feature to view how each element in your instance model interacts with others.

- When you first opened the eMoflon perspective, a small window to the right should have appeared with two tabs, "Outline" and "Graph View." [10]
- Activate the second tab and drag one of your `Partition` instances into the empty window.
- To the right you can see the partition that was dragged in, double click on it to visualise all referenced objects from `Partition`. (Figure 4.1).
- Clicking any item in the view will highlight it, and hovering over a node will list all its properties in a pop-up dialogue. If you hover over

[10] If this window is not open, you can re-activate it by right-clicking on the "eMoflon" perspective in the main toolbar and pressing "Reset," or by going to "Window/Show View/Other..," then "Other/Graph View"

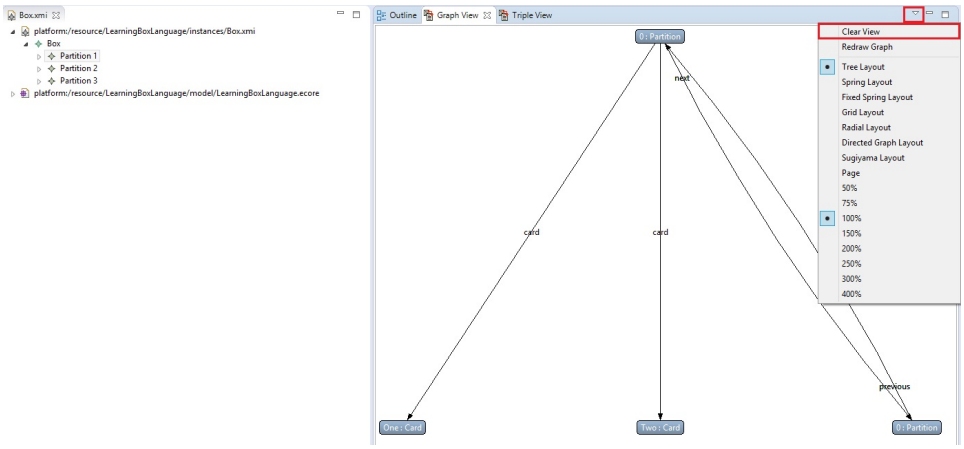


Figure 4.1: A single partition in eMoflon's graph view

a card, for example, you should be able to see their `back` and `face` values. You can also reposition elements.

- Try dragging a second `Partition` element from the model to the viewer. As you can see, it doesn't replace the current view, it simply places them side-by-side, showing the references between them partitions. To clear the screen, click the upside-down triangle in the top right corner of the window, and select "Clear View" (Figure 4.1).
- To make the graph bigger, increase the size of the window, then select "Redraw Graph" from the same upside-down arrow. As you can see, when an instance is already loaded, the graph view is not automatically updated. This option is useful if you change a property of an element (ie., the `back` value of a `Card`) and wish to see it reflected in the graph.
- Try experimenting with each of the different instance elements, viewer layouts and zoom settings found under the arrow. You'll notice for the "Spring Layout" that each time you press "Redraw Graph," the graph will re-arrange itself, even if you haven't updated any values or changed the window size.
- The "Graph View" features is not exclusively for instance models; Expand the second root node in the editor, and drag and drop the `LearningBoxLanguage` package into the viewer. Click on some or all nodes to open your entire metamodel completely. Now it is displayed in the viewer (Figure 4.2). We have found that the "Radial Layout" works best for a view of this complexity. This might be confusing at first, but the view contains your metamodel displayed in its abstract syntax, i.e., as an instance of Ecore. In contrast, the tree view displays the metamodel in UML-like concrete syntax.

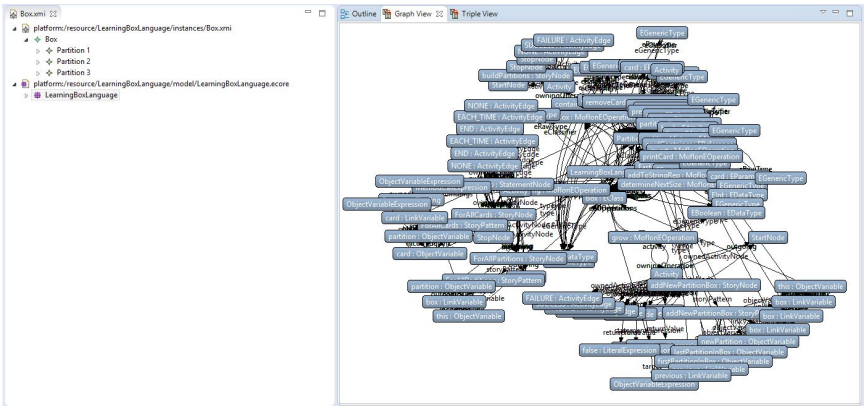


Figure 4.2: Our instance's complete type graph

5 Introduction to injections

This short introduction will show you how to implement small methods by adding handwritten code to classes generated from your model. Injections are inspired by partial classes in C#, and are our preferred way of providing a clean separation between generated and handwritten code.

Let's implement the `removeCard` method, declared in the `Partition` EClass. In order to 'remove' a card from a partition, all one needs to do is disable the link between them. Don't forget that (according to the signature) not only does `removeCard` have to pass in a `Card`, it must return one as well.

- From your working set, open "gen/LearningBoxLanguage.impl/PartitionImpl.java" and enter the following code in the `removeCard` declaration, starting at approximately line 347. Do not remove the first comment, which is necessary to indicate that this code is written by the user and needs to be extracted automatically as an injection. Please also do not copy and paste the following code – the copying process will most likely add invisible characters that eMoflon is unable to handle.

```
public Card removeCard(Card toBeRemovedCard) {
            // [user code injected with eMoflon]
            if(toBeRemovedCard != null){
                    toBeRemovedCard.setCardContainer(null);
            }
            return toBeRemovedCard;
    }
```

Figure 5.1: Implementation of `removeCard`

- Save the file, then right-click either on the file in the package explorer, or in the editor window, and choose "eMoflon/ Create/Update Injection for class" (Alt+Shift+E,I) from the context menu (Figure 5.2).
- This will create a new file in the "injection" folder of your project with the same package and name stucture as the Java class, but with a new `.inject` extension (Figure 5.3).
- Double click to open and view this file. It contains the definition of a *partial class* (Figure 5.4).

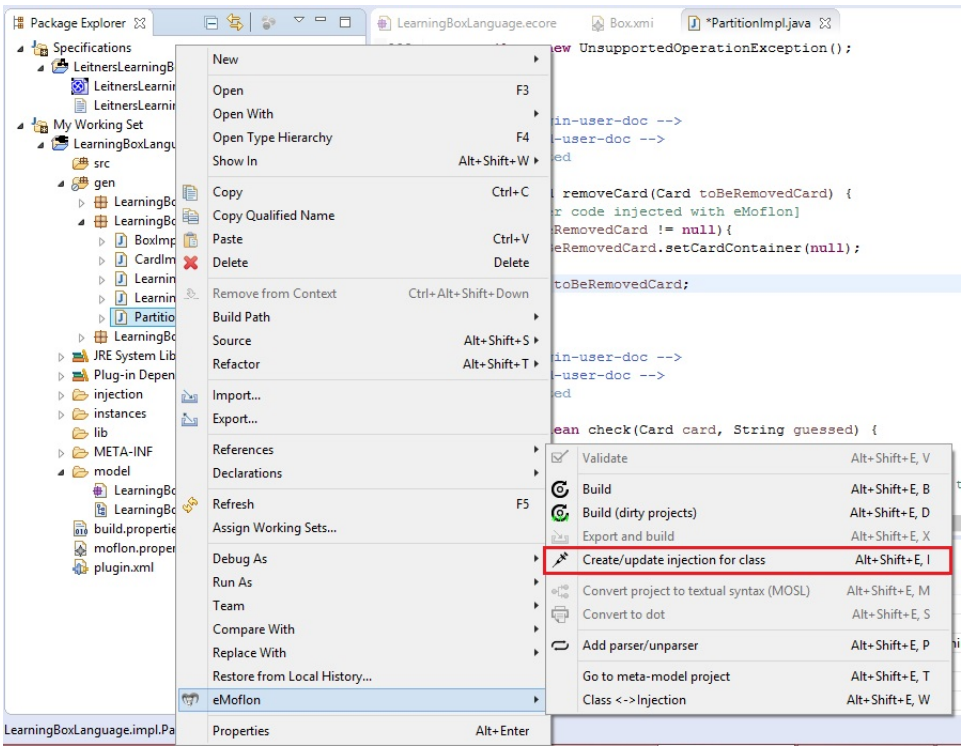


Figure 5.2: Create a new injection

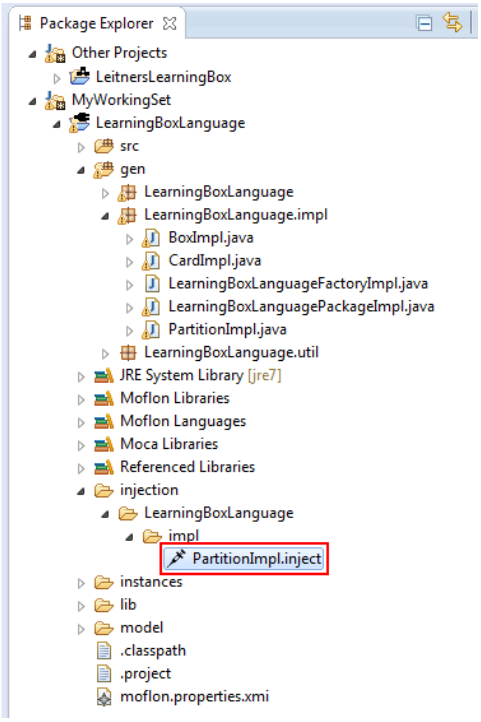


Figure 5.3: Partition injection file

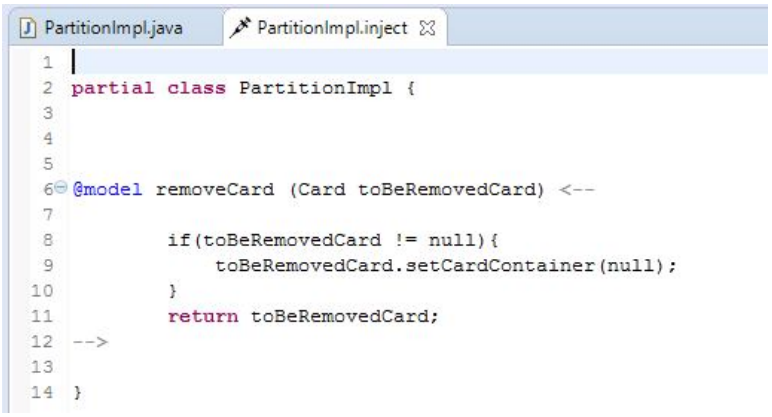


Figure 5.4: Generated injection file for `PartitionImpl.java`

- As a final step, build your metamodel to check that the code is generated and injected properly.
- By the way, eMoflon allows you to switch quickly between a Java class and its injection file. When inside "PartitionImpl.java", open the context menu and select "eMoflon/Class <-> Injection" (Alt+Shift+E,W). This brings you to "PartitionImpl.java".

  Repeat this command and you are back in "PartitionImpl.inject".
- That's it! While injecting handwritten code is a remarkably simple process, it is pretty boring and low level to call all those setters and getters yourself. We'll return to injections for establishing two simple methods in Part III using this strategy, but we'll also learn how to implement more complex methods using Story Diagrams.

Creating injections automatically with save actions

Information loss is a typical pitfall that you may encounter when working with injections: If you forget to create an injection and trigger a rebuild ("eMoflon/Build", Alt+Shift+E,B), the generated code is entirley dropped – including any unsaved injection code.

To save you from this frustrating experience, eMoflon may automatically save injections whenever you save your Java file.

- Open the preferences dialog via "Window/Preferences" and navigate to the "Save Actions" ("Java/Editor/Save Actions").
- To enable save actions, tick "Perform the selected actions on save" and "Additional actions" (Figure 5.5).
- Press "Configure", switch to the "eMoflon Injections" tab and tick "Enable injection extraction on save" (Figure 5.6).
- After confirming the dialog, the bulleted list should contain the entry "Create eMoflon injections".
- Open up "PartitionImpl.java", perform a tiny modification on it, and save the file. The eMoflon console should display a message to confirm that injections have been extracted automatically, e.g.,

  ```
  [handlers.CreateInjectionHandler::115] -  Created injection
  file for 'PartitionImpl'.
  ```

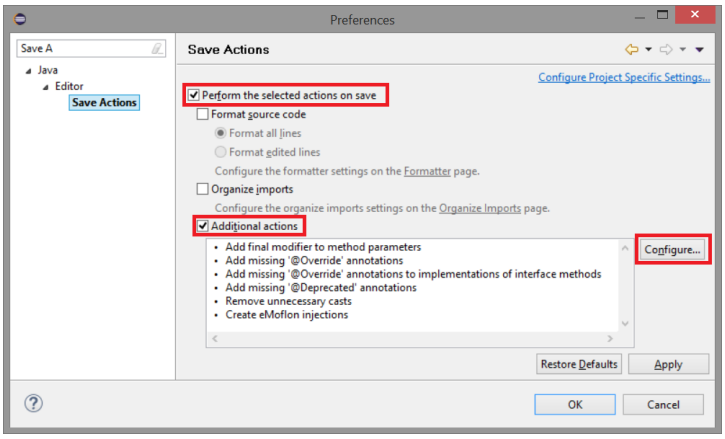


Figure 5.5: Main configuration dialog for save actions

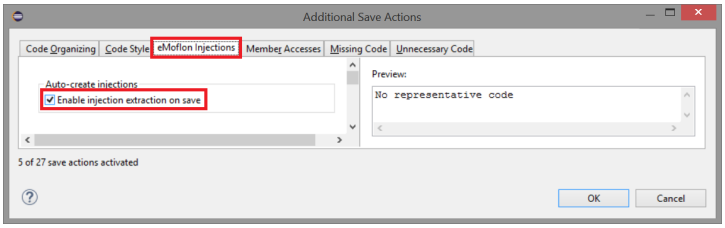


Figure 5.6: Configuration dialog for additional save actions

6 Leitner's Box GUI

We would like to now provide you with a simple GUI with which you can take your model for a spin and see it in action.

- Navigate to the "Install, configure and deploy Moflon" button and open the "Install Workspace" menu bar (Figure 6.1).
- Load "Handbook Example (GUI)" (Figure 6.1). This will load the new project into your workspace. Right click `LeitnersBox` to invoke the context menu and select "Run as/Java application."

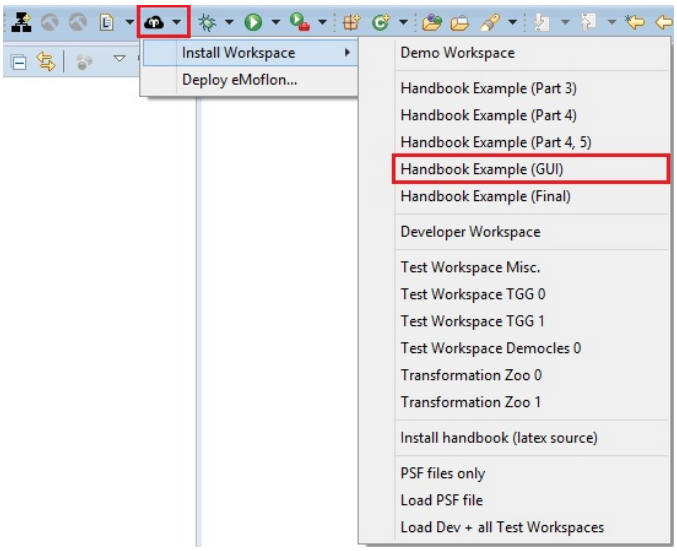


Figure 6.1: Load Leitner's Box GUI

- The GUI will automatically navigate to the instances folder where you've stored your instance model, then load your partitions and cards into the visualized box. Please note that this will only work if you named your dynamic instance `Box.xmi` and placed it in the `instances` folder as suggested.

- Navigate to any card, and you'll be able to see two options, "Remove Card" and "Check Card" (Figure 6.2). While we'll implement "Check Card" in Part III, "Remove Card" is currently active, implemented by the injection you just created.

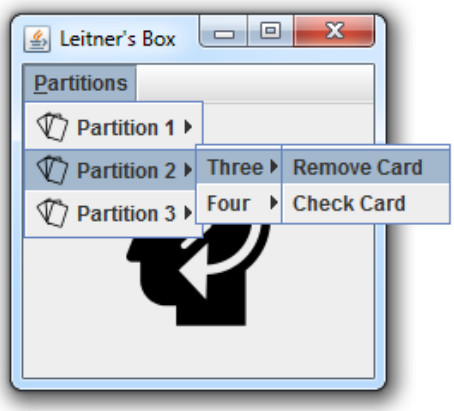


Figure 6.2: Using the GUI with your instance

- Experiment with your instance model and confirm your injection works by removing some items from the GUI. You'll notice the change immediately in the `Box.xmi` file. You can also close the GUI and add, remove, or rename more elements in the model, then observe how the changes are reflected in the GUI.

- Expand the "Other Projects" node and explore `LeitnersBoxController` and `LeitnersBoxView` files to get an understanding of the GUI. Can you find where the controller loads and connects to the model?

7 Conclusion and next steps

Whoo, this has been quite a busy handbook. Great job, you've finished Part II! This part contained some key skills for eMoflon, as we learned how to create the abstract syntax in Ecore for our learning box! The specification is now complete with all the classes, attributes, references, and method signatures that make up the type graph for a working learning box. Additionally, we also learned how to insert a small handwritten method into generated code, and tested all our work in an interactive GUI.

If you enjoyed this section and wish to fully develop *all* the methods we just declared, we invite you to carry on to Part III: Story Diagrams[11]! Story Diagrams are a powerful feature of eMoflon as we can model a large part of a system's dynamic semantics via high-level pattern rules.

Of course, you're always free to pick a different part of the handbook if you feel like skipping ahead and checking out Triple Graph Grammars (TGGs) in Part IV[12]. We'll provide instructions on how to easily download all the required resources so you can start without having to complete the previous parts.

For a detailed description of all parts, please refer to Part 0[13].

Cheers!

[11] Download: `https://emoflon.github.io/eclipse-plugin/beta/handbook/part3.pdf`

[12] Download: `https://emoflon.github.io/eclipse-plugin/beta/handbook/part4.pdf`

[13] Download: `https://emoflon.github.io/eclipse-plugin/beta/handbook/part0.pdf`

Glossary

Abstract Syntax Defines the valid static structure of members of a language.

Concrete Syntax How members of a language are represented. This is often done textually or visually.

Constraint Language Typically used to specify complex constraints (as part of the static semantics of a language) that cannot be expressed in a metamodel.

Dynamic Semantics Defines the dynamic behaviour for members of a language.

Grammar A set of rules that can be used to generate a language.

Graph Grammar A grammar that describes a graph language. This can be used instead of a metamodel or type graph to define the abstract syntax of a language.

Meta-Language A language that can be used to define another language.

Meta-metamodel A *modeling language* for specifying metamodels.

Metamodel Defines the abstract syntax of a language including some aspects of the static semantics such as multiplicities.

Model Graphs which conform to some metamodel.

Modelling Language Used to specify languages. Typically contains concepts such as classes and connections between classes.

Static Semantics Constraints members of a language must obey in addition to being conform to the abstract syntax of the language.

Type Graph The graph that defines all types and relations that form a language. Equivalent to a metamodel but without any static semantics.

Unification An extension of the object oriented "Everything is an object" principle, where everything is regarded as a model, even the metamodel which defines other models.

An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part III: Story Driven Modelling

For eMoflon Version 2.16.0

File built on 21st September, 2016

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

The eMoflon team
Darmstadt, Germany (September 2016)

Contents

Part III:

Story Driven Modelling

URL of this document: https://emoflon.github.io/eclipse-plugin/beta/handbook/part3.pdf

Welcome to Part III, an introduction to unidirectional model transformations with programmed graph transformations via Story Driven Modelling (SDM). SDMs are used to describe behaviour, so the plan is to implement the methods declared in Part II with story diagrams. In other words, this is where you'll complete your metamodel's dynamic semantics! Don't let the size of this part frighten you off. We have included thorough explanations (with an ample number of figures) to ensure the concepts are clear.

In Part II, we learnt that we can implement methods in a fairly straightforward manner with injections and Java, so why bother with SDMs?

Overall, SDMs are a simpler, pattern-based way of specifying behavior. Rather than writing verbose Java code yourself, you can model each method and generate the corresponding code.

If you're just joining us, read the next section for a brief overview of our running example so far, and how to download some files that will help you get started right away. Alternatively, if you've just completed Part II, click the link below to continue right away with your constructed learning box metamodel. Please note that the handbook has been tested only with the prepared cheat packages provided in eMoflon.

▷ Continue from Part II...

1 Leitner's learning box reviewed

Leitner's learning box[1] is a simple, but ingenious little contraption to support the tedious process of memorization, especially prominent when trying to learn, for example, a new language. As depicted in Figure 1.1, a box consists of a series of partitions with a strict set of rules. The contents to be memorized are written on little cards and placed in the first container. Every time the user correctly answers a card, that card is promoted to the next partition. Once it reaches the final partition, it can be considered memorized, and no longer needs to be practiced. Every time the user incorrectly answers a card however, it is returned to the original starting partition, and the learning process is restarted.

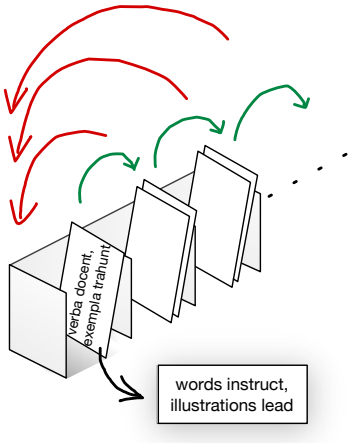


Figure 1.1: Static Structure of a Leitner's Learning Box

For a more detailed overview of the box and our goals, we recommend you read the introduction to Part II. But for now, enough discussion!

- To get started in Eclipse, press the `Install, configure and deploy Moflon` button and navigate to "Install Workspace". Choose "Handbook Example (Part 3)". (Figure 1.2).
- The cheat package contains all files created up to the example in this point, as well as a small GUI that will enable you to experiment with your metamodel.

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

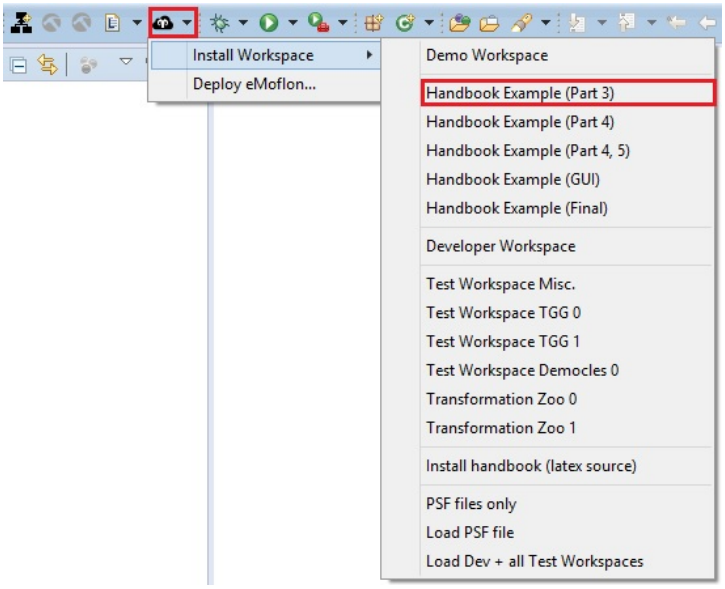


Figure 1.2: Choose a cheat package

- In order to start working with the cheat package, you have to generate code by (i) opening the `.eap` file in `LeitnersLearningBoxVisual`, (ii) exporting it using Enterprise Architect, (iii) refreshing the project containing the `.eap` and (iv) rebuilding the `LearningBoxLanguage` project. For more details on the code generation process, refer to Part I, Section 2.
- Inspect the files in both projects until you feel comfortable with what you'll be working with. In particular, look at the files found under "gen." Each Java file has a corresponding `.impl` file, where all generated method implementations will be placed.
- Be sure to also review the Ecore model in "LearningBoxLanguage/-model/" and the dynamic model found in "instances." While you can make and customize your own instances,[2] we have included a small sample to help you get started.

Well, that's it! A quick review, paired with a fine cheat package makes an excellent appetizer to SDMs. Let's get started.

[2]To learn how to make your own instance models, review Part II, Section 4

2 Transformations explained

The core idea when modeling behaviour is to regard dynamic aspects of a system (let's call this a model from now on) as bringing about a change of state. This means a model in state S can evolve to state S^* via a transformation $\Delta : S \overset{\Delta}{\rightarrow} S^*$. In this light, dynamic or behavioural aspects of a model are synonymous with *model transformations*, and the dynamic semantics of a language equates simply to a suitable set of model transformations. This approach is once again quite similar to the object oriented (OO) paradigm, where objects have a 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 in which we simply say how the system morphs in a step-by-step manner. 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's got to be a better way!

From the relatively mature field 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 (postcondition). The actual task of going from precondition to postcondition should be taken over by a transformation engine, where all related details are basically regarded as a black box.

So, inspired by string grammars and this new, refined idea of a model transformation (which is of the form $(pre, post)$), let's call this black box transformation a *rule*. It follows that the precondition is the left-hand side of the rule, L, and the postcondition is 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 or *Destroying* $(L \setminus R)$, i.e., the elements that are present in the precondition but not in the postcondition are deleted from G to form $(G \setminus Destroy)$
3. *Pasting* or *Creating* $(R \setminus L)$, i.e., new elements that are present in the postcondition but not in the precondition and are to be created in the hole left in $(G \setminus Destroy)$ to form a new graph, $H = (G \setminus Destroy) \cup Create$ (Figure 2.1).

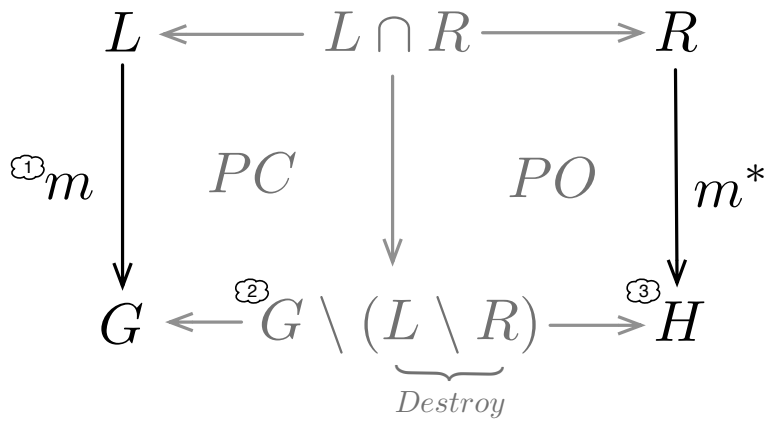


Figure 2.1: Applying a rule $r : (L, R)$ to G to yield H

Let's review this application.

(1) is determined by a process called *graph pattern matching* i.e., finding an occurrence or *match* of the precondition or *pattern* in the model G. *Pattern Matching*

(2) is determined by building a *push-out complement* $PC = (G \setminus Destroy)$, such that $L \cup PC = G$.

(3) is determined by building a *push-out* $PO = H$, so that $(G \setminus Destroy) \cup R = H$.

A push-out (complement) is a generalised union (subtraction) defined on typed graphs. Since we are dealing with graphs, it is not such a trivial task to define (1) – (3) in precise terms, with conditions for when a rule can or cannot be applied. A substantial amount of theory already exists to satisfy this goal.

Since this black box formalisation involves two push-outs - one when cutting $Destroy := (L \setminus R)$ from G to yield $(G \setminus Destroy)$ (deletion), and one when inserting $Create := (R \setminus L)$ in $(G \setminus Destroy)$ to yield H (creation) - this construction is referred to as a *double push-out*. We won't go into further details in this handbook, but the interested reader can refer to [**?**] for the exciting details.

Now that we know what rules are, let's take a look at a simple example for our learning box. What would a rule application look like for moving a card from one partition to the next? Figure 2.2 depicts this *moveCard* rule.

As already indicated by the colours used for *moveCard*, we employ a compact representation of rules formed by merging (L, R) into a single *story*

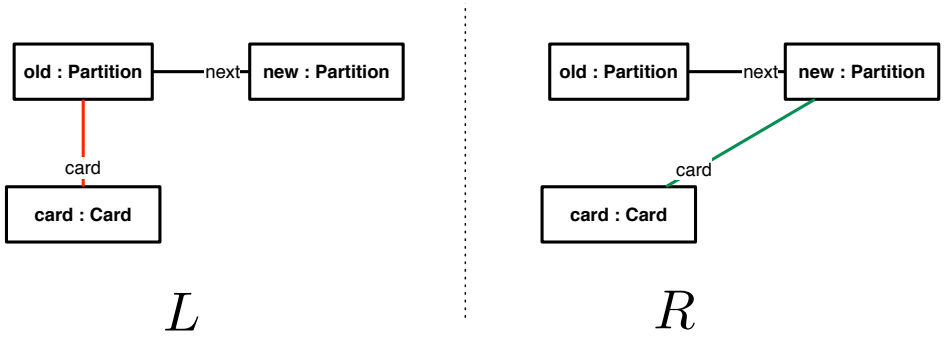


Figure 2.2: *moveCard* as a graph transformation rule

pattern composed of $Destroy := (L \setminus R)$ in red, $Retain := L \cap R$ in black, and $Create := (R \setminus L)$ in green (Figure 2.3). *Story Pattern*

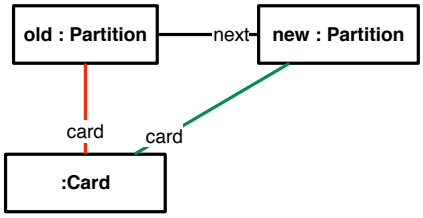


Figure 2.3: Compact representation of *moveCard* as a single *story pattern*

As we shall see in a moment, this representation is quite intuitive, as one can just forget the details of rule application and think in terms of what is to be deleted, retained, and created. We can therefore apply *moveCard* to a learning box in terms of steps (1) – (3), as depicted in Figure 2.4.

Despite being able to merge rules together to form one story pattern, the individual rules still have to be applied in a suitable sequence to realise complex model transformations consisting of many steps! This can be specified with simplified activity diagrams, where every *activity node* or *story node* contains a single *story pattern*, and are combined with the usual imperative constructs to form a control flow structure. The entire transformation can therefore be viewed as two separate layers: an imperative layer to define the top-level control flow via activities (i.e., if/else statements, loops, etc.), and a pattern layer where each story pattern specifies (via a graph transformation rule) how the model is to be manipulated. *Activity Node*

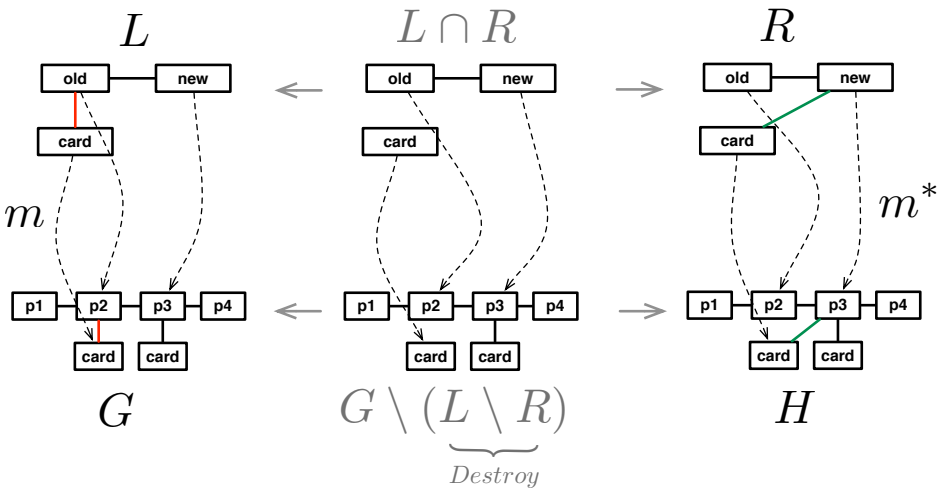


Figure 2.4: Applying *moveCard* to a learning box

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

3 Removing a card

Since we're just getting started with SDMs,[3] let's re-implement the method previously specified directly in Java as an injection.[4] The goal of this method is to remove a single card from its current partition, which can be done by destroying the link between the two items (Figure 3.1).

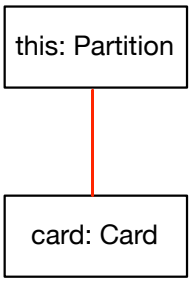


Figure 3.1: Removing a card from its partition

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 the card as an argument, it still makes sense as it allows for chaining method calls:

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

Before we implement this change as a story diagram, let's remove the old injection content to avoid potential conflicts.

- Delete the `PartitionImpl.inject` file from your working set (Figure 3.2).
- Now select `LearningBoxLanguage` and click on the "Build" button.
- You'll be able to see the changes in `PartitionImpl.java`. The `removeCard` declaration should now be empty and look identical to the other unimplemented methods.

[3] As you may have already noticed, we use "SDM" or "Story Diagrams" interchangeably to mean both our graph transformation language *or* a concrete transformation used to implement a method, consisting of an activity with activity nodes containing story patterns.

[4] Refer to Part II, Section 6

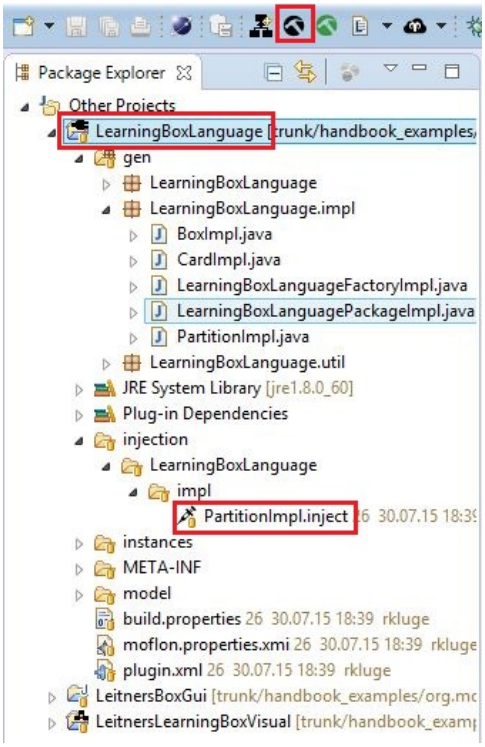


Figure 3.2: Remove injection content

That's it! We now have a fresh start for `removeCard`. Let's briefly discuss what we need to establish the transformation.

One of the goals of SDM is to allow you to focus less on *how* a method will do something, but rather on *what* the method will do. Integrated as an atomic step in the overall control flow, a single graph transformation step (such as link deletion) can be embedded as a *story pattern*.

Object Variable

These patterns declare *object variables*, place holders for actual objects in a model. During *pattern matching*, objects in the current model are assigned to the object variables in the pattern according to the indicated type and other conditions.[5]

[5] We shall learn what further conditions may be specified in later SDMs.

In `removeCard`, the SDM requires just two object variables: a `this` partition (named according to Java convention) referring to the object whose method is invoked, and `card`, the parameter that will be removed.

Link Variable

Patterns also declare *link variables* to match references in the model. Given that we're concerned with removing a certain card from a specific partition, `removeCard` will therefore have a single link variable that connects these two objects together.

In general, pattern matching is non-deterministic, i.e., variables in the pattern are bound to *any* objects that happen to match. How can this be influenced so that, as required for `removeCard`, the pattern matcher chooses the correct `card` (that which is passed in as a parameter)?

Binding State *Free Variable* *Bound*

The *binding state* of an object variable determines how it is found. By default, every object variable is *unbound*, or a *free variable*. Values for these variables can be determined automatically by the pattern matcher. By declaring an object variable that is to be *bound* however, it will have a fixed value determined from previous activity nodes. The appropriate binding is implicitly determined via the *name* of the bound object variable. As a rule, `this` variables, and any method parameters (i.e., `card`) are always bound.

On a final note, every object or link variable can also set its *binding operator* to `Check Only, Create, or Destroy`. For a rule $r : (L, R)$, as discussed in Section 2, 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 ($L \setminus R$).

If you're feeling overwhelmed by all the new terms and concepts, don't worry! We will define them again in the context of your chosen syntax with the concrete example. For quick reference, we have also defined the most important terms at the end of this part in a glossary.

3.1 Implementing removeCard

- Open `LearningBoxLanguage.eap` in Enterprise Architect (EA) by double clicking it in Eclipse. Carefully do the following: (1) Click *once* on `Partition` to select it, then (2) Click *once* on the method `removeCard` to highlight it (Figure 3.3), and (3) *Double-click* on the chosen method to indicate that you want to implement it.

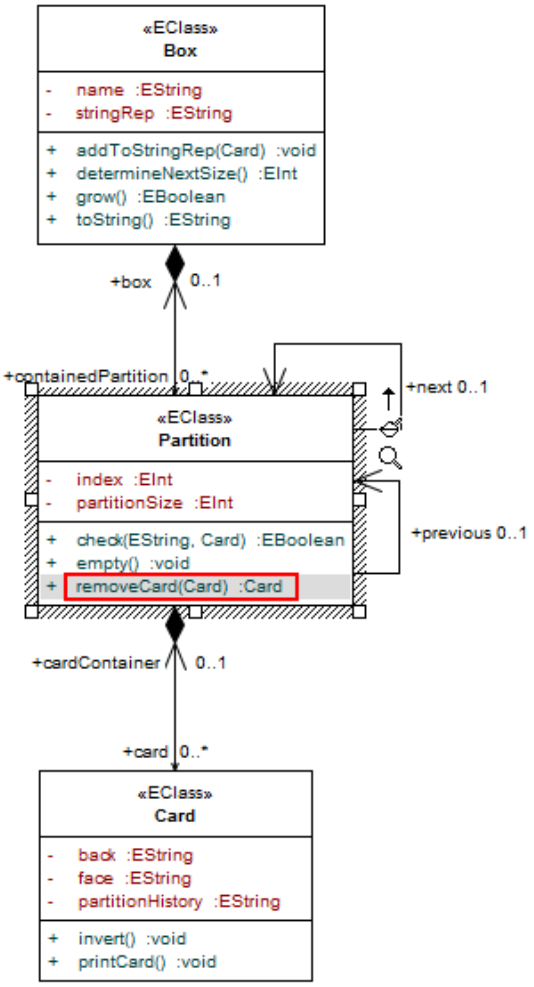


Figure 3.3: Double-click a method to implement it

- If you did everything right (and answered the question which popped up about creating a new SDM diagram with Yes), a new *activity diagram* should be created and opened in a new tab with a cute anchor in the corner, and a *start node* labelled with the signature of the method

(Figure 3.4).

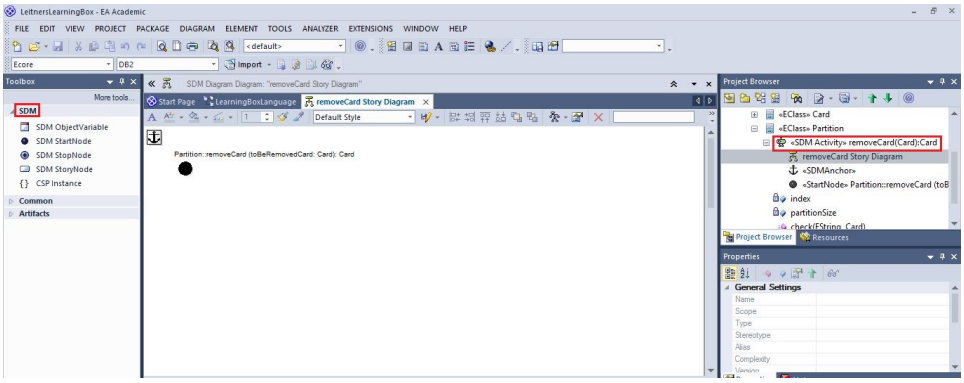


Figure 3.4: Generated SDM diagram and start node

- This diagram is where you'll model `removeCard`'s *control flow*. In other words, this is `removeCard`'s imperative top-level diagram. We refer to the whole activity diagram simply as the *activity*, which always starts with a *start node*, contains *activity nodes* connected via *activity edges*, then finally terminates with a *stop node*. Before creating these however, let's quickly familiarise ourselves with the EA workspace.

- First, inspect the project browser and notice that an `<<SDM Activity>>` container has been created for the method `removeCard`. This container will eventually host every artifact related to this pattern (i.e., object variables, stop nodes, etc.). Please note that if you're ever unhappy with an SDM, you can always delete the appropriate container in the project browser (such as this one), and start from scratch.

- Next, note the new `SDM` toolbox that has been automatically opened for the diagram and placed to the left above the common toolbox. This provides quick access to SDM items that you'll frequently use in your diagram. You can also invoke the active toolbox in a pop-up context menu anywhere in the diagram by pressing the `space` bar.

- Finally, in the top left corner of the diagram, you'll notice a small anchor. Double click on this icon to quickly jump back to the meta-model. From there, double click the method again to jump back to the SDM. This is just a small trick to help you quickly navigate between diagrams.

- To begin, select the start node, and note the small black arrow that appears (Figure 3.5).

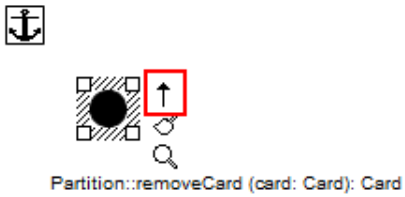


Figure 3.5: Quick link in SDM diagram to create new activity node

- Similar to quick linking,[6] a second fundamental gesture in EA is *Quick Create*. To quick-create an element, pull the arrow and click on an empty spot in the diagram. This is basically "quick linking" to a non-existent element.
- EA notices that there is nothing to quick-link to, and pops up a small, context-sensitive dialogue offering to create an element which can be connected to the source element.
- As illustrated in Figure 3.6, choose `Append StoryNode` to create a *Story Node*.

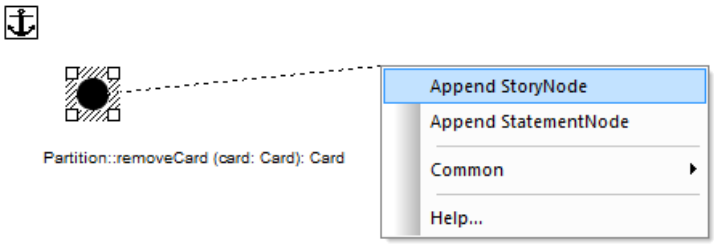


Figure 3.6: Create new activity node

- If you quick-created correctly, you should now have a start node, one node called `ActivityNode1`, and an edge connecting the two items. Complete the activity by quick-creating a stop node (Figure 3.7).

[6] This was discussed in Part II, Section 2.5

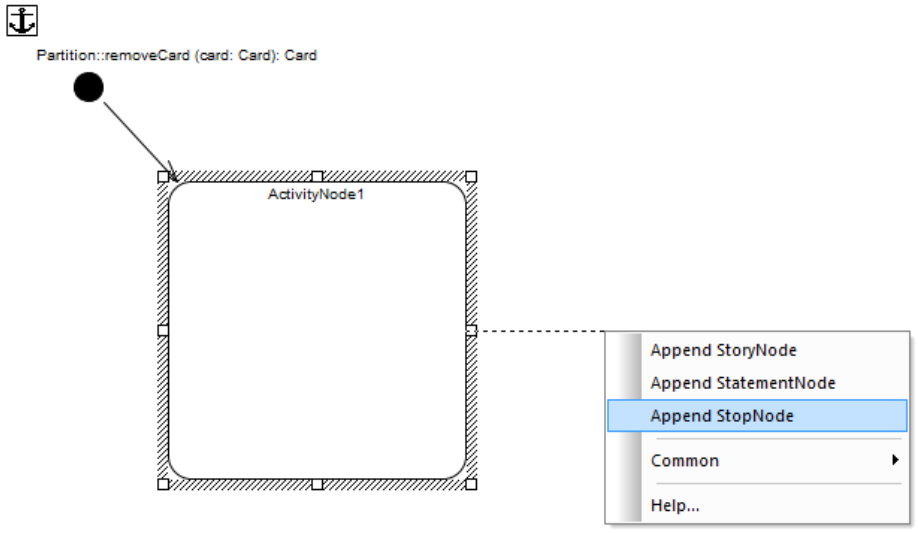


Figure 3.7: Complete the activity with a stop node

- If everything is correct, you should now have a fully constructed activity that models the method's control flow.
- While a *stop node* is rather self explanatory, you may be wondering about the differences between the other two menu options, the *story node* and *statement node*. Since not all activity nodes can contain story patterns (e.g., start and stop nodes), those that *can* are called story nodes. Statement nodes cannot and are used instead to invoke an action, such as method execution. We'll encounter this in a later SDM.

Story Node
Statement Node

- To complete this activity, double-click `ActivityNode1` to prompt the dialogue depicted in Figure 3.8. Enter `removeCardFromPartition` as the name of the story node, and select `Create this Object`. Click `OK`. The activity node now has a single *bound object variable*, `this`.
- To create a new object variable, choose `SDM ObjectVariable` from the toolbox then click inside the activity node (Figure 3.9). A properties window will automatically appear (Figure 3.10).

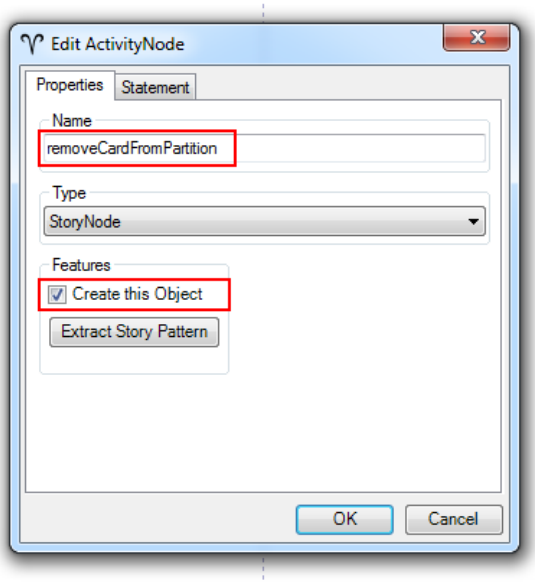


Figure 3.8: Initializing a story node

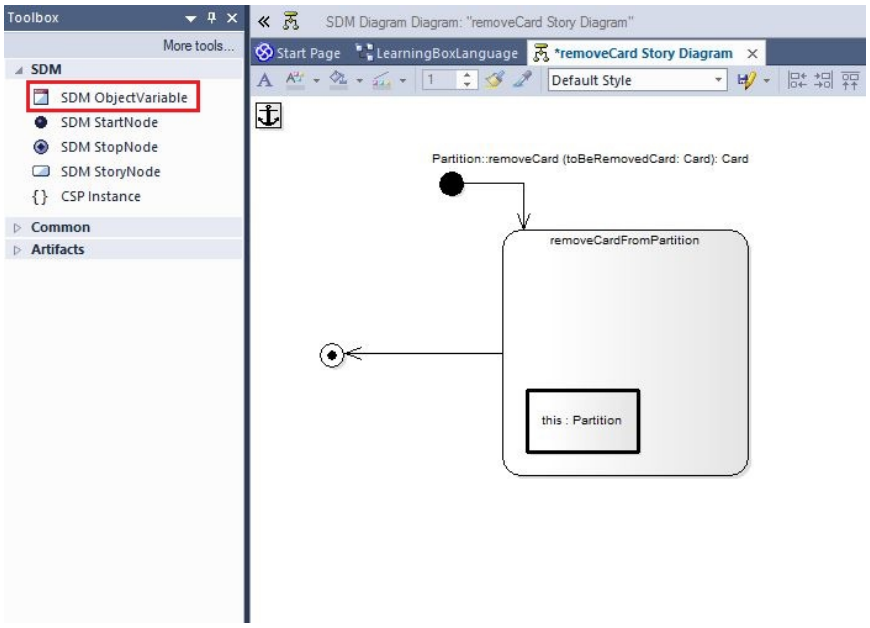


Figure 3.9: Add a new object variable from the toolbox

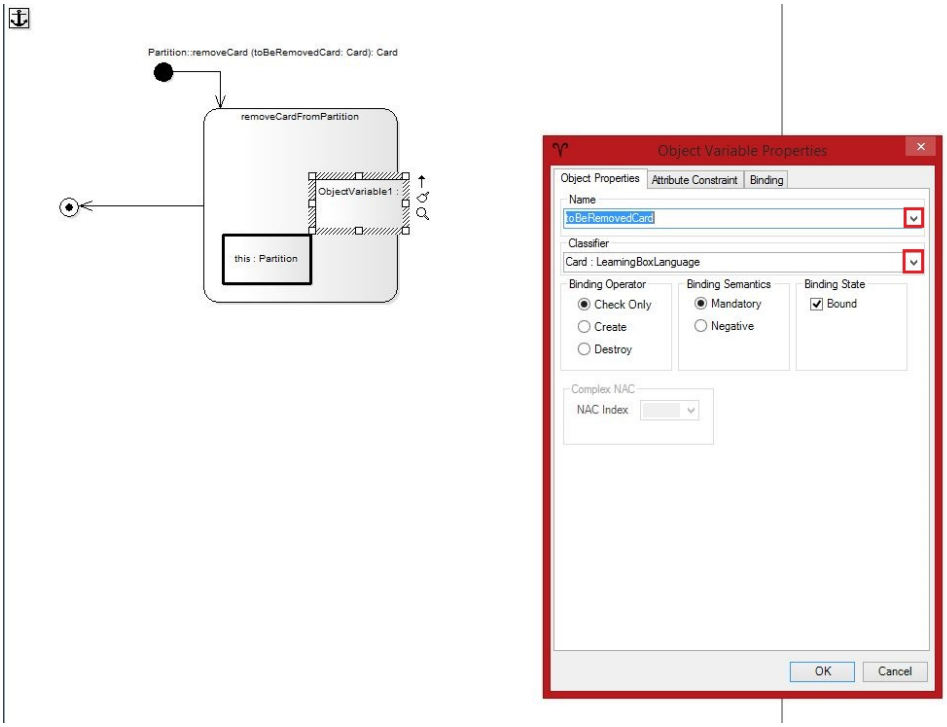


Figure 3.10: Specify properties of the added object variable

- Using the drop-down menus, choose `toBeRemovedCard` as the name of the object, and set `Card` as its type. Since `card` is a parameter of the method, it is offered as a possible name which can be directly chosen to avoid annoying typing mistakes.
- In this dialogue, note that the `Bound` option is automatically set. We have now seen two cases in this activity for bound object variables: an assignment to `this`, and an assignment to a method parameter. Setting `toBeRemovedCard` to bound means that it will be implicitly assigned to the parameter with the same name.
- To create a *link variable* between the current partition and the card to be removed, choose the object variable `this` and quick link it to `toBeRemovedCard` (Figure 3.11).
- According to the metamodel, there is only one possible link between a partition and card. Select this and set the `Binding Operator` to `Destroy` (Figure 3.12). The reference names will automatically appear in the diagram.

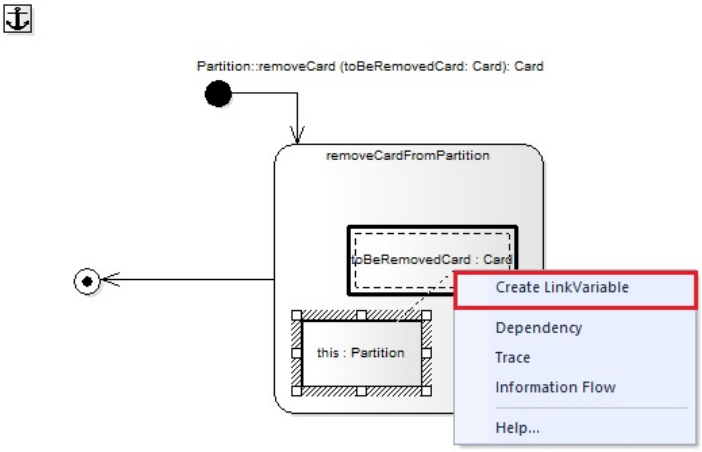


Figure 3.11: Create a link variable

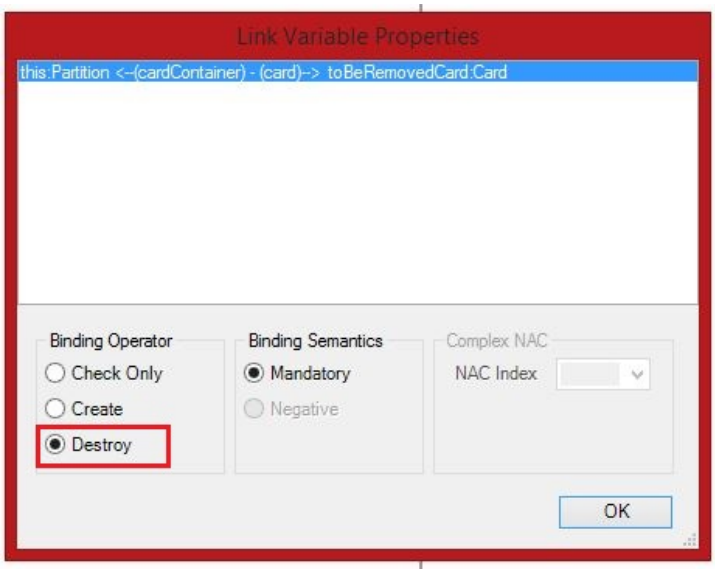


Figure 3.12: Specify properties for created link variable

- Remember how we said that this method should return the same card that was passed in? As luck would have it, a return value for an SDM can be specified in the stop node. As depicted in Figure 3.13, double-click the stop node to prompt the `Edit StopNode` dialogue.

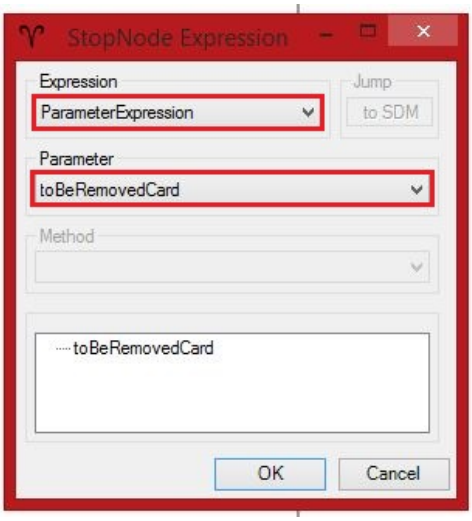


Figure 3.13: Adding a return value to the stop node

- In the `Expression` field, choose the `ParameterExpression` option. As suggessted by its name, a *ParameterExpression* is an expression that exclusively accesses method parameters. Given that `toBeRemovedCard` is the sole parameter, the `object` will be automatically set to this value. In other words, the returned object is now implicitly *bound* by having the same name. *ParameterExpression*

  We're nearly done! As you can see, eMoflon uses a series of dialogues to provide a simple context-sensitive expression language for specifying values. In the following SDM implementations, we'll learn and discuss some other expression types eMoflon supports.

- Returning to the activity, if you've done everything right, your first SDM should resemble Figure 3.14, where `removeCard`'s entire pattern layer is modeled inside the sole *activity node*. The method's return value is now indicated below the stop node.

- Don't forget to save your files, validate and export your pattern to the Eclipse workspace,[7] then build your metamodel's code from the package explorer.

[7] Go to "`Extensions`" and select `Add-In Windows` to activate eMoflon's console. If you're unsure how to validate, export, or use this window, review Part I, Section 2.1.

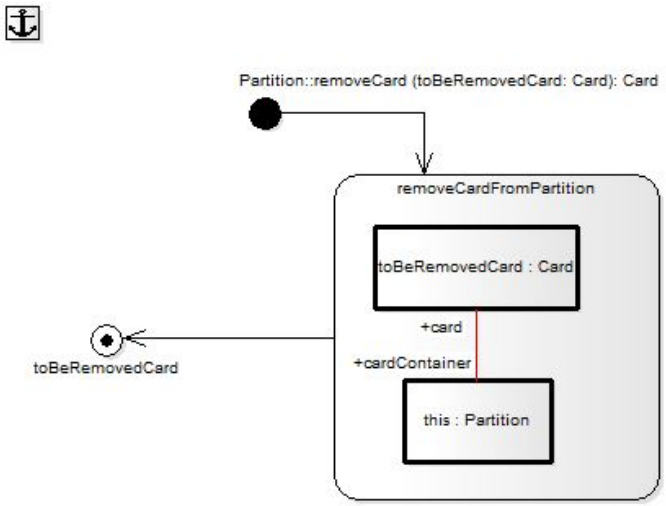


Figure 3.14: Complete SDM for `Partition::removeCard`

- If you're unable to export or generate code successfully, compare your SDM carefully with Figure 3.14 and make sure you haven't forgotten anything.

Removing SDMs

Imagine that you have just created an SDM for a method, but now you want to implement the method using an injection. This section describes how to get rid of this old SDM.

Let's assume that you want to remove the SDM of `Partition::removeCard`, which you just created.

- In the project browser, navigate to the method of which you want to remove the SDM. This can be accomplished by right-clicking the method in the Ecore diagram and then selecting "Find in project browser" (Figure 3.15), or by navigating manually through the project browser.
- Inside of the EClass "Partition", you find the one entry for the method ("removeCard(Card)") and one entry for its SDM ("≪SDM Activity≫ removeCard(Card)") (Figure 3.16).
- Select and remove the SDM Activity to get rid of the SDM associated with "removeCard(Card)" either by using the context menu entry

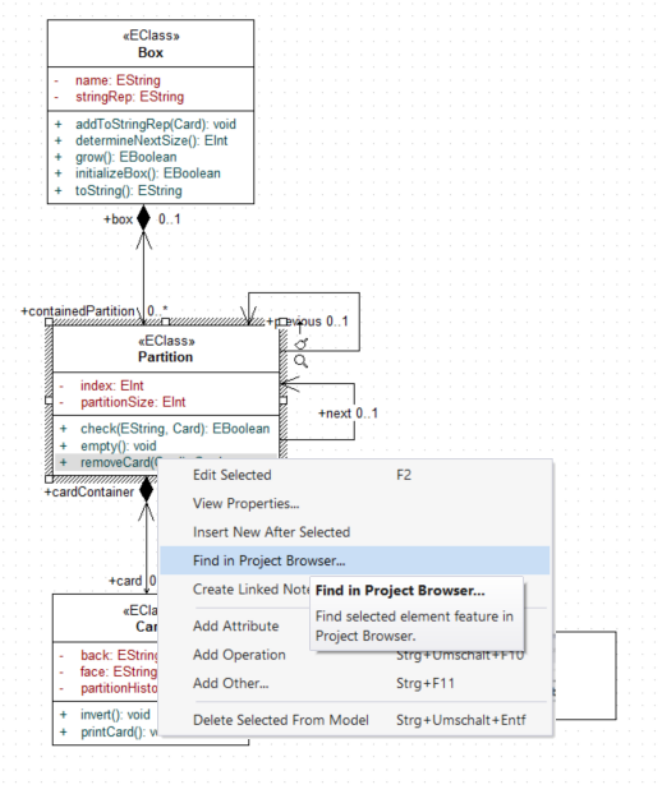


Figure 3.15: Navigate to method in project browser

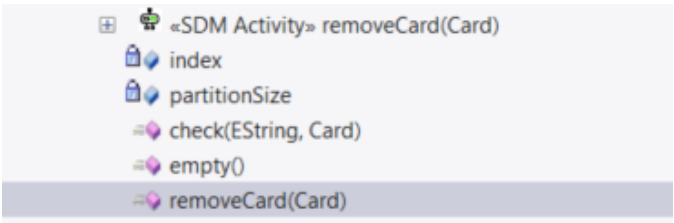


Figure 3.16: Project browser entries for method “removeCard” and its SDM

"Delete '≪SDM Activity≫ removeCard(Card)'" or via *Ctrl+Delete*. **Important:** This action cannot be undone!

- After a successful export and rebuild of your project, you may add an injection for "Partition::removeCard".

Concluding removeCard

Fantastic work! You have now implemented a simple method via patterns. As you can see, SDMs are effective for implementing structural changes in a high-level, intuitive manner.

Let's take a step back and briefly review what we have specified: if `p.removeCard(c)` is invoked for a partition `p`, with a card `c` as its argument, the specified pattern will *match* only if that card is contained in the partition. After determining matches 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 will happen. In both cases, the card that's passed in is returned.

- If your code generation was successful, navigate to "LearningBoxLanguage/gen/LearningBoxLanguage/impl/PartitionImpl.java" to the `removeCard` declaration (approximately line 347). Inspect the generated implementation for your method (Figure 3.17). Notice the null check that is automatically created - only a very conscientious (and probably slightly paranoid) programmer would program so defensively!

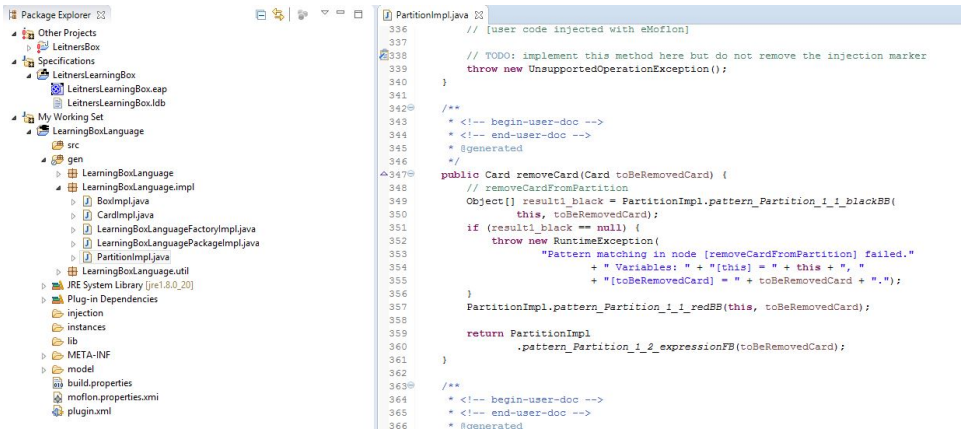


Figure 3.17: Generated implementation code

Near the end of Part II (after using injections), you were able to test this method's implementation using our `LeitnersBoxGui`. Let's run it again to make sure *this* version of `removeCard` works!

- Load and run the GUI as an application,[8] then go to any partition and select `Remove Card` (Figure 3.18). It should immediately refresh, and you'll no longer be able to see the card in either the GUI or in the `Box.xmi` tree in the "instances" folder. Pretty cool, eh?

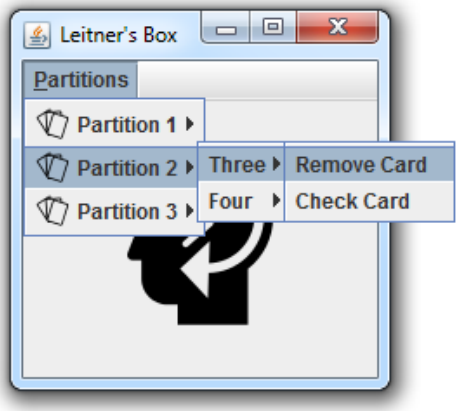


Figure 3.18: Testing `removeCard`

[8] Refer to Part II, Section 6 for details on our GUI

4 Checking a card

The next method we shall model is probably the most important for our learning box. This method will be invoked when a user decides to test themselves on a card in the learning box. They'll be able to see the `back` attribute of a card from the box, make a guess as to what's on the `face`, then check their answer (Figure 4.1). Following our rules established in Figure 1.1, if their guess was correct the card will be *promoted* to the next partition. If wrong, the card will be *penalized* and returned to the first.

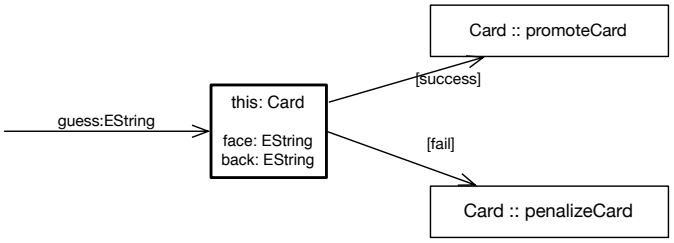


Figure 4.1: Checking a card with a guess

As you can see, checking `guess` is a simple assertion on string values. The actual movements of the card however, must be implemented as separate patterns. Figure 4.2 briefly shows the intended create and destroy transformations.

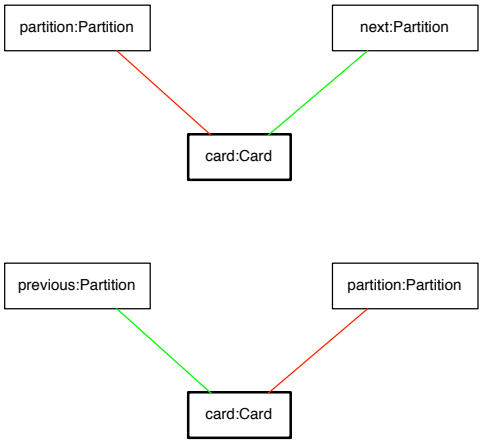


Figure 4.2: Promote (above) and penalize (below) card patterns

Overall, this means the control flow must utilize an *if/else* construct. The `guess` conditional also needs to be an *attribute constraint*, a non-structural condition that must be satisfied in order for a story pattern to match.

Attribute Constraint

4.1 Implementing check

- Since you're nearly an SDM wizard already, try using concepts we have already learnt to create the control flow for `Partition::check` as depicted in Figure 4.3.

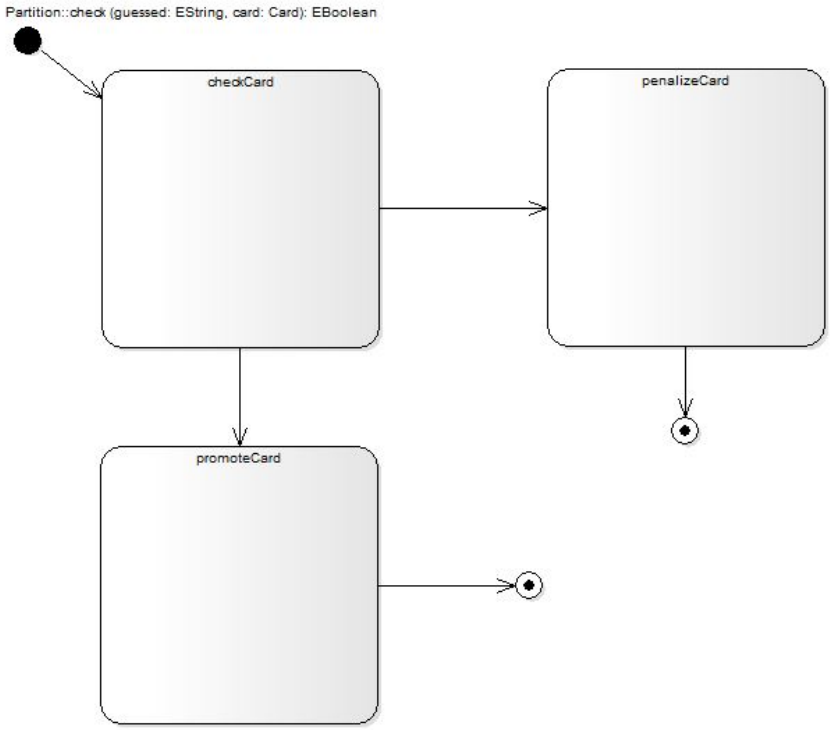


Figure 4.3: Activity diagram for `Partition::check`

- In `checkCard`, create an object variable that is bound to the parameter argument, `card` (Figure 4.4). This will represent the card the user picked from the learning box. Remember, the binding for this variable is implicitly defined because its name is the same as the argument's.

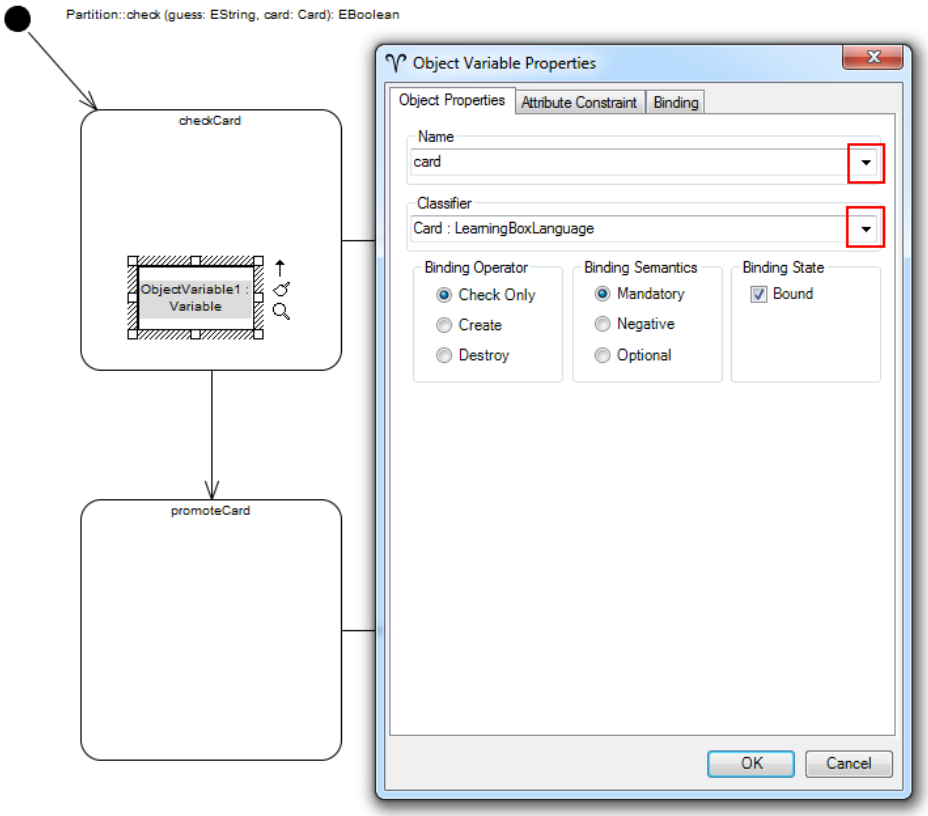


Figure 4.4: Creating the card variable

- Now that the pattern has the correct card to check, it needs to compare the user's guess against the unseen `face` value on the opposite side. To do this, we need to specify an *attribute constraint*. Open the `attribute constraint` tab for `card` as depicted in Figure 4.5, and select the correct `Attribute` and `Operator`.

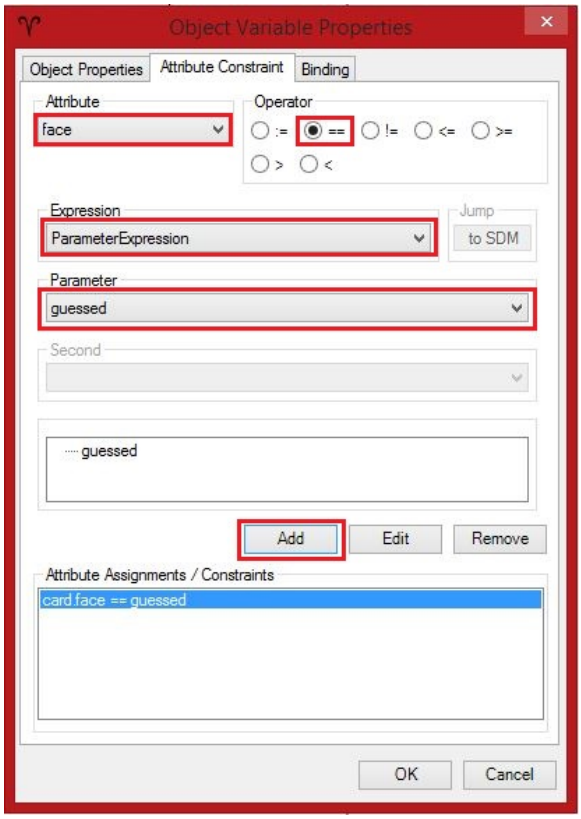


Figure 4.5: Creating an attribute constraint

- Similar to how the return value was specified in the previous SDM, set the `ParameterExpression` to refer to `guess`, i.e., the user's EString input. Press `Add`, and admire your first conditional.

Before building the other two activity nodes, let's quickly return to the control flow. Currently, the pattern branches off into two separate patterns after completing the initial check, and it is unclear how to terminate the method. As the code generator does not know what to do here, this is flagged as a validation error (you're free to press the validation button and take cover). We need to add *edge guards* to change this into an *if/else* construct based on the results of the *attribute constraint*.

Edge Guards

- To add a guard to the edge leading from `checkIfGuessIsCorrect` to `penalizeCard`, double click the edge and set the *Guard Type* to `Failure` (Figure 4.6). Repeat the process for the `Success` edge leading to `promoteCard`.

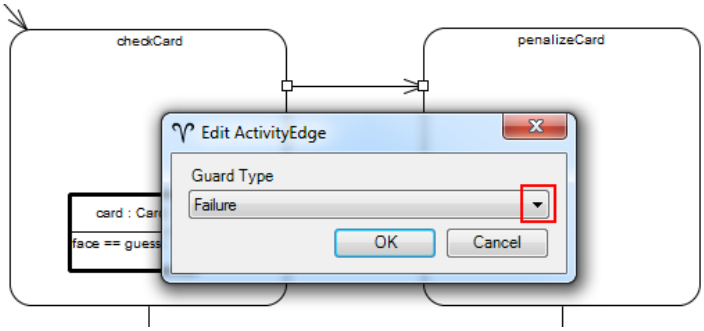


Figure 4.6: Add a transition with a guard

Edge guards: Success, Failure, or None? You may now wonder why we did not use edge guards already for implementing `removeCard`. In fact, an unguarded activity edge is equivalent with with a `Success` edge. More precisely, if you create an activity node only one outgoing unguarded edge you assert that the pattern contained in this activity node will *always* be successful. In case the pattern is not applicable, a runtime exception will be thrown. In contrast, having an outgoing `Success` and `Failure` edge describes that you allow your pattern to not match.

Extracting Story Patterns One great feature of eMoflon (with EA) is a means of coping with large patterns. It might be nice to visualise *small* story patterns directly in their nodes (such as `removeCardFromPartition`), but for large patterns or complex control flow, such diagrams would get extremely cumbersome and unwieldy *very* quickly! This is indeed a popular argument against visual languages and it might have already crossed your mind – "This is cute, but it'll *never* scale!" With the right tools and concepts however, even huge diagrams can be mastered. eMoflon supports *extracting* story patterns into their own diagrams, and unless the pattern is really concise with only 2 or 3 object variables, we recommend this course of action. In other words, eMoflon supports separating your transformation's pattern layer from its imperative control flow layer.

- To try this, double-click the `promoteCard` story node and choose `Extract Story Pattern` (Figure 4.7). Note the new diagram that is immediately created and opened in the project browser (Figure 4.8).

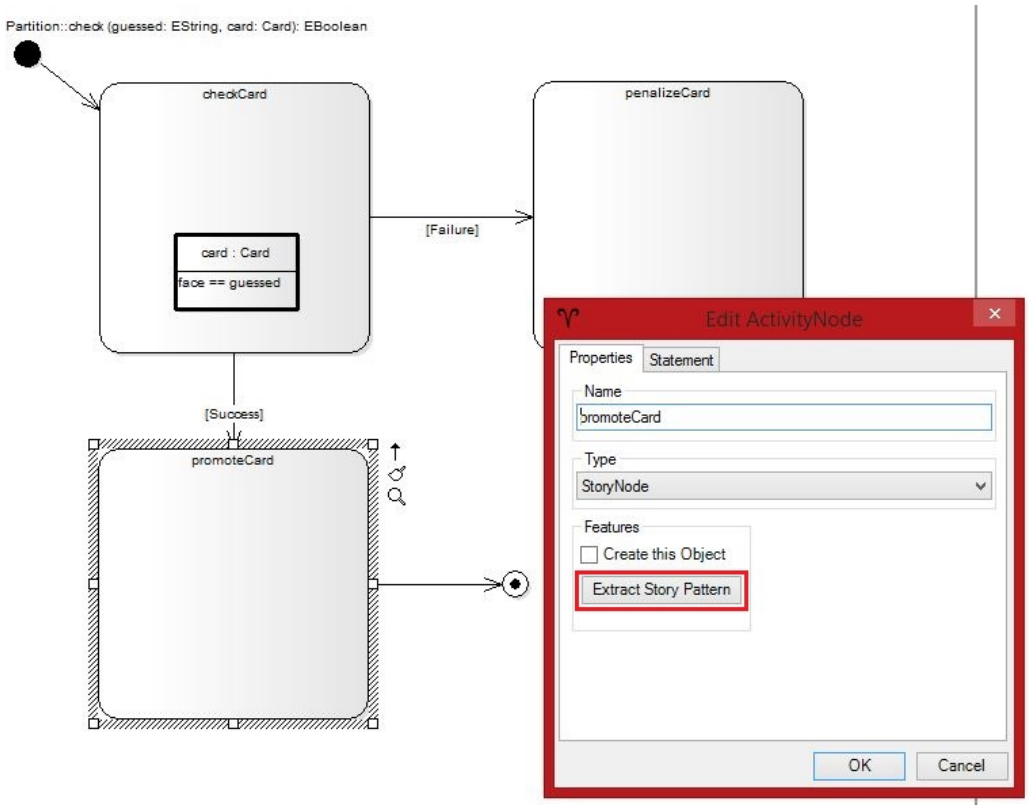


Figure 4.7: Extract a story pattern for more space and a better overview

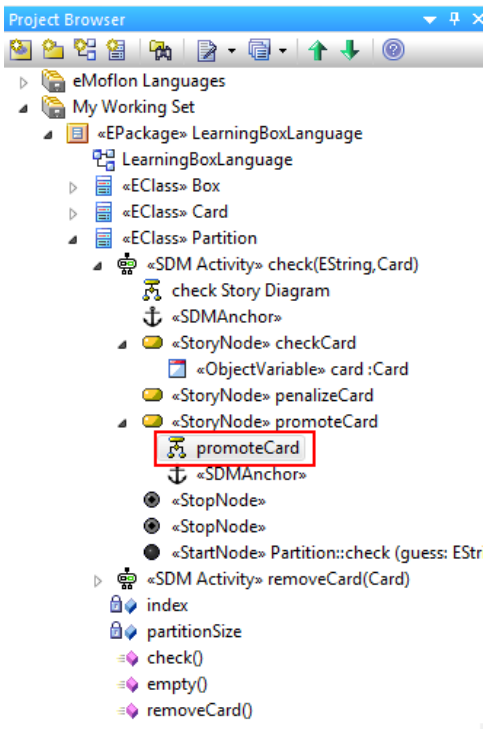


Figure 4.8: A new subdiagram is created automatically

Another EA gesture[9] you could start to take advantage of here is good ol' *Drag-and-Drop* from the project browser into a diagram. We can use this action as an alternative to creating new objects (with known types) from the SDM toolbox.

The main advantage of drag-and-drop is that the `Object Variable Properties` dialogue will have the type of the object pre-configured. Choosing the type in the project browser and dragging it in is (for some people) a more natural gesture than choosing the type from a long drop-down menu (as we had to when using the SDM toolbar). This can be a great time saver for large metamodels.[10]

- To put this into practice, create a new `Card` object variable by drag-and-dropping the class from the project browser into the new (extracted) pattern diagram (Figure 4.9).

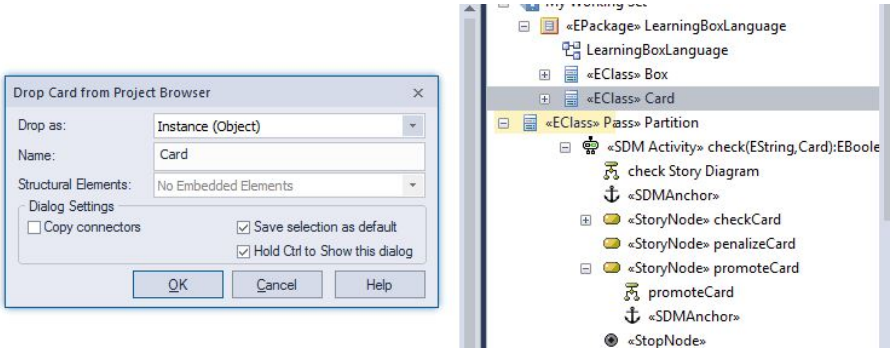


Figure 4.9: Add a new object variable per drag-and-drop

- A dialogue will appear asking what kind of visual element should be created. You can create (1) a simple link (which would refer to and be represented by the class `Card`), (2) create an instance of `Card` as an object variable, or (3) as an invocation (which has no meaning for eMoflon diagrams). Paste `Card` as an `Instance`, and select `Autosave Selection as Default` under "Options" so option (2) will be used next time by default. You should also select `Use Ctrl + Mouse drag to show this Dialog`, so this dialogue doesn't appear every time you use this gesture. Don't worry – if you ever need option (1), hold `Ctrl` when dragging to invoke the dialogue again.

[9]The other two gestures we have learnt are "Quick Link" and "Quick Create"

[10]Drag-and-drop is also possible in embedded story patterns (those 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.

- After creating the object, the object properties dialogue will open. Set the `Name` to `card` and confirm its `Binding State` is `Bound` (Figure 4.10).

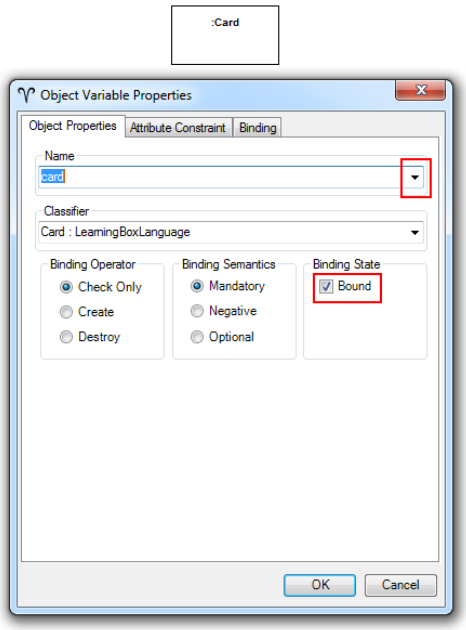


Figure 4.10: Object variable properties of the new card

- Currently, we have the single `card` that we want to promote through the box. Drag-and-drop two partition objects, `this`, and `nextPartition` as depicted in Figure 4.11.

  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* from *unbound* (*free*) variables. We already know that matches for bound variables are completely determined by the current context. On the other hand, matches for unbound variables have to be determined by the pattern matcher. Such matches are "found" by navigating and searching the current model for possible matches that satisfy all specified constraints (i.e., type of the variable, links connecting it to other variables, and attribute constraints). In our case, `nextPartition` should be determined by navigating from `this` via the `next` link variable.

- To specify this, quick link from `this` to `nextPartition` (or vice-versa)

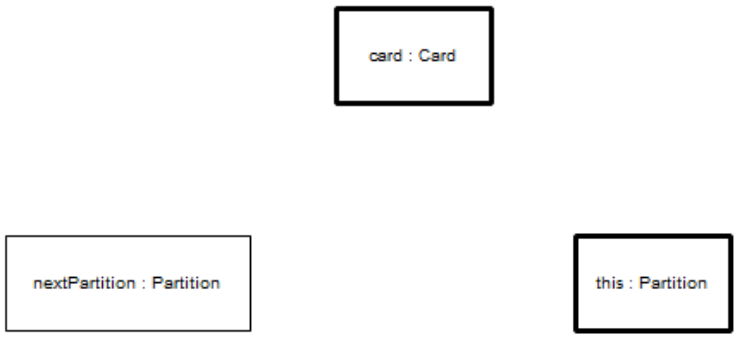


Figure 4.11: All object variables for story pattern `promoteCard`

to establish `next`, as shown in Figure 4.12. As you can see, there are several more options than what was seen in `removeCard`. The goal is to have the current partition to proceed (or point) to the `nextPartition` via the `next` reference, so select the second option. Alternatively, you could define the reference from `nextPartition` by setting the link variable `previous` to `this`.

- Continue by creating links between `card` and each partition. Remember - you want to *destroy* the reference to `this`, and *create* a new connection to `nextPartition`. If everything is set up correctly, `promoteCard` should now closely resemble Figure 4.13.

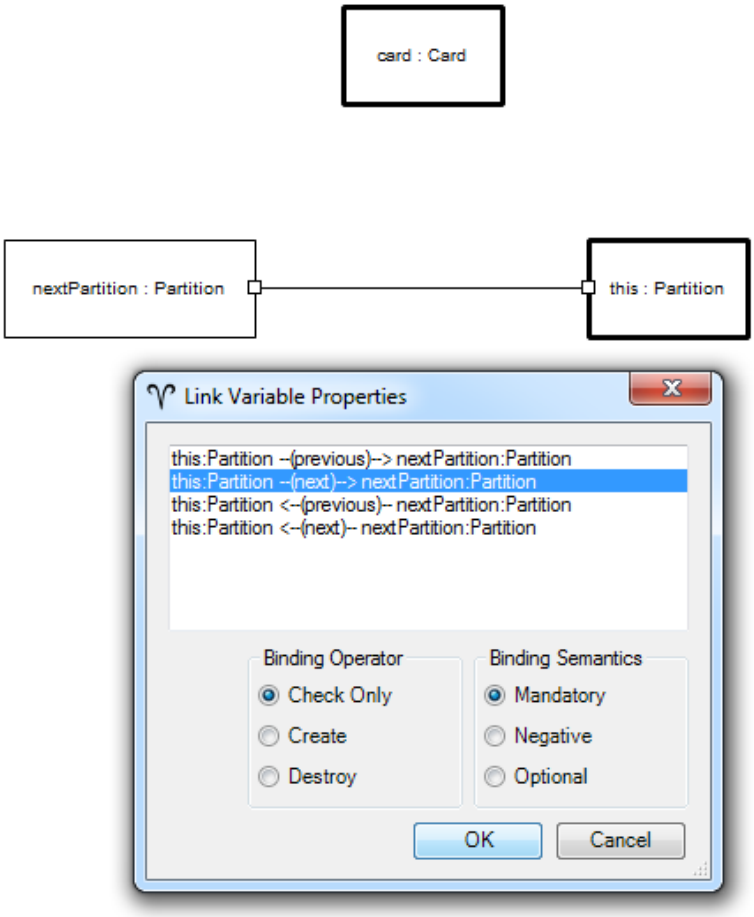


Figure 4.12: Possible links between `this` and `nextPartition`

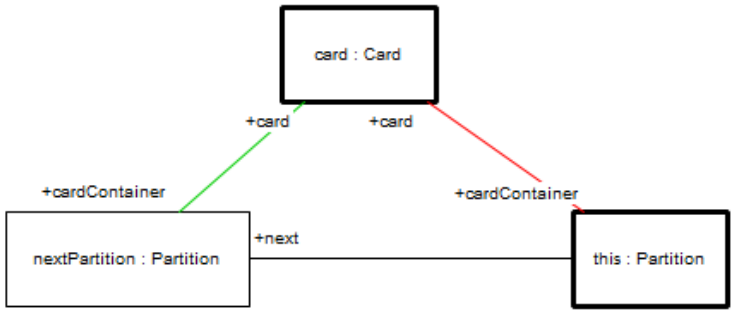


Figure 4.13: Complete story pattern for `promoteCard`

- Double click the anchor in the top left corner and repeat the process for `penalizeCard`: First extract the story pattern, then create the necessary variables and links as depicted in Figure 4.14. As you can see, this pattern is nearly identical to `promoteCard`, except it moves `card` to a `previousPartition`.

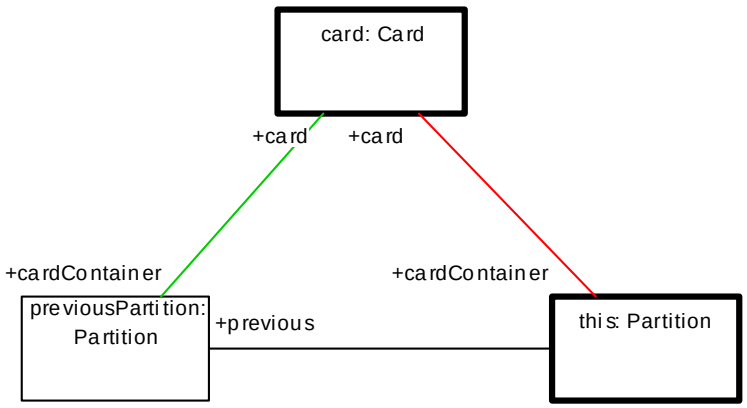


Figure 4.14: Complete story pattern for `penalizeCard`

To complete the `check` activity, we need to signal (as a return value) the result of the check - was the card promoted or penalized? We have no object to return so instead, we need to edit the stop nodes so they return a *LiteralExpression*. This expression type can be used to specify arbitrary text, but should really only be used for true literals like 42, "foo" or `true`. It can be (mis)used for formulating any (Java) expression that will simply be transferred "literally" into the generated code, but this is obviously really dirty[11] and should be avoided when possible.

LiteralExpression

- To implement a literal, double click the stop node stemming from `promoteCard`, and change the expression type from `void` to `LiteralExpression` (Figure 4.15). Change the value in the window below to `true`. Press `OK`, then finish the SDM by returning `false` after `penalizeCard` in the same manner. Note that it is also possible that `promoteCard` or `penalizeCard` fails[12]. This is handled by placing parallel `Success`

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

[12] For example, if the actual partition doesn't have a "proper" successor or predecessor – in this case, unfortunately, we can't give a real reward or penalty for the guess, which means that we still return `true` or `false`.

and `Failure` edges after the `promoteCard` and `penalizeCard` activities. After doing these, your diagram should resemble Figure 4.16.

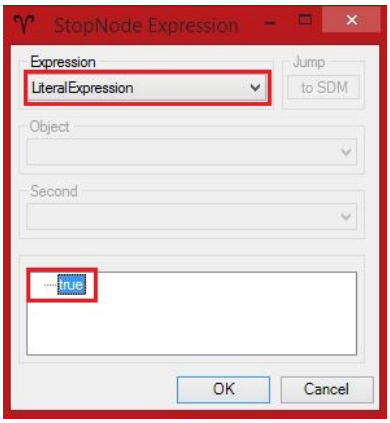


Figure 4.15: Add a return value with a literal expression

- Great job – the SDM is now complete! Validate and export your project, then inspect the implementation code for `check`. We strongly recommend that you even write a simple JUnit test (take a look at our simple test case from Part I for inspiration) to take your brand new SDM for a test-spin.

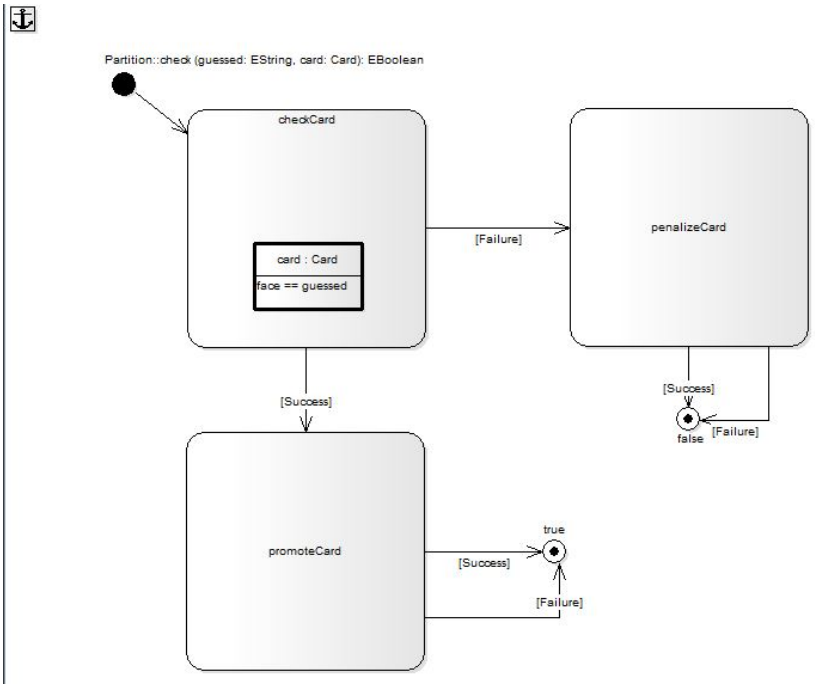


Figure 4.16: Complete SDM for `Partition::check`

5 Running the Leitner's Box GUI

In addition to `removeCard`, the GUI is already able to access and execute the `check` method based on your SDM implementation. First, double check that your metamodel is saved and built, then run the GUI.

- Pick a card from any of your partitions, then run `check`. You'll be prompted with a dialogue box to make your guess in (Figure 5.1).

- Enter your word, then press `OK`. You should immediately see any movement changes in the drop-down menus, and shortly in `box.xmi` after refreshing.

- Fully test your implementation by making right and wrong guesses. Watch how the cards move around – do they behave as expected, following the rules of Leitner's Box?

- At this point, we invite you to browse the `LeitnersBoxControl-`

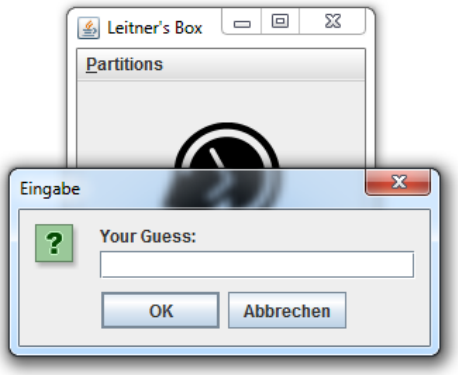


Figure 5.1: Enter your guess

`ler.java` file. Can you see how `removeCard` and `check` were called and executed? You are encouraged to modify this file so that you may be able to test your future SDM implementations.

6 Emptying a partition of all cards

This next SDM should *empty* a partition by removing every card contained within it. Since we can assume that there is more than one card in the partition,[13] we obviously need some construct for repeatedly deleting each card in the partition (Figure 6.1).

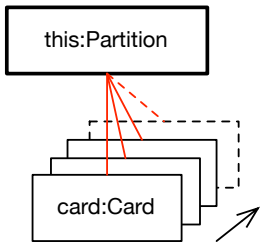


Figure 6.1: Emptying a partition of every card

For Each

In SDM, this is accomplished via a *for each* story node. It performs the specified actions for *every* match of its pattern (i.e., every `Card` that matches the pattern will be deleted). This however, gives us two interesting points to discuss. Firstly, how would the pattern be interpreted if the story node were a normal, simple control flow node, not a *for each* node?

The pattern would specify that *a* card should be matched and deleted from the current partition - that's it. The *exact* card is not specified, meaning that the actual choice of the card is *non-deterministic* (random), and it is only done once. This randomness is a common property of graph pattern matching, and it's something that takes time getting used to. In general, there are no guarantees concerning the choice and order of valid matches. The *for each* construct however, ensures that *all* cards will be matched and deleted.

The second point is determining if we actually need to destroy the link between `this` and `card`. Would the pattern be interpreted differently if we destroyed `card` and left the link?

Dangling Edges

The answer is no, the pattern would yield the same result, regardless of whether or not the link is explicitly destroyed! This is due to the transformation engine eMoflon uses.[14] It ensures that there are never any *dangling edges* in a model. Since deleting just the `card` would result in a dangling edge attached to `this`, that link is deleted as well. Explicitly destroying the links as well is therefore a matter of taste, but ... why not be as explicit as possible?

[13] If there was only one, we would just invoke `removeCard`

[14] CodeGen2, a part of the Fujaba toolsuite `http://www.fujaba.de/`

6.1 Implementing empty

- Create a new activity diagram for `Partition.empty()`. To begin building the *for each* pattern, quick create a new story node and edit its properties. Name it `deleteCardsInPartition` and change its `Type` from `StoryNode` to `ForEach`. You'll also want to create the invoking `Partition` object, `this` (Figure 6.2). Press `OK`, and you'll see that a *for each* node is represented as a stacked node to indicate the potential for repetition.

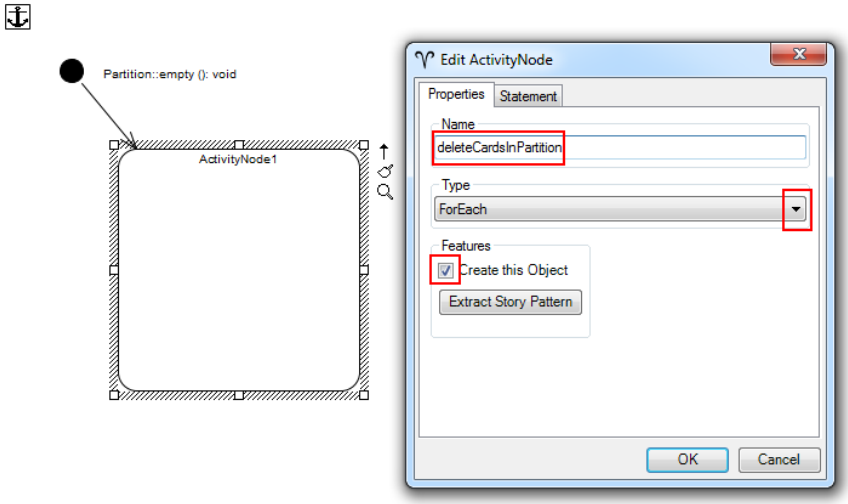


Figure 6.2: Creating a looping story node

- Now create the `card` object variable needed to complete this SDM. Unlike `removeCard` (Figure 3.14) however, the goal of `emptyCards` is not just to remove the link between the selected partition and card, we want the matched `card` to be *completely* deleted. This means in the properties tab, after setting the name and binding state, you'll need to set the `Binding Operator` to `Destroy` (Figure 6.3).

- Complete the story pattern as indicated in Figure 6.4. Notice that the guard that terminates the looping node has an `[end]` edge guard. Indeed, a *for each* story node *must* execute an `end` activity when all matches in the pattern have been handled. `empty` is defined as a `void` method, so don't worry about setting any return value in the stop node.

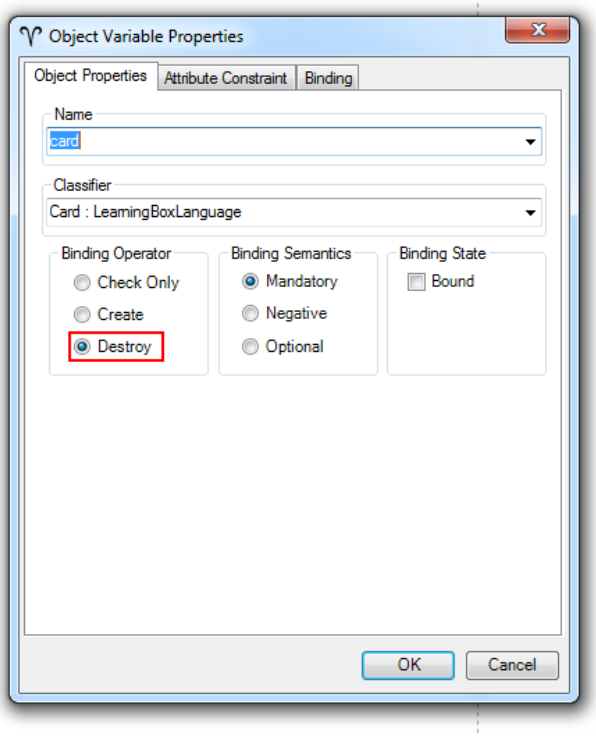


Figure 6.3: Editing `card` so it gets destroyed

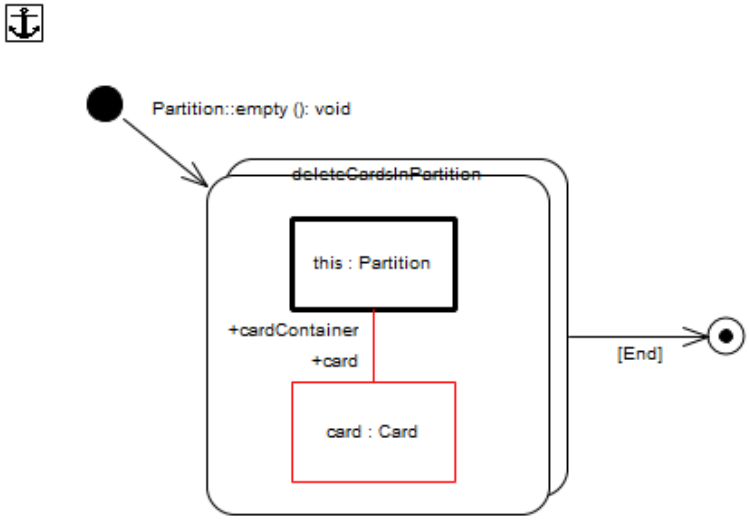


Figure 6.4: Completed `empty` story pattern

- Done! You've now learnt that in order to create a repeating action, all you need to do is change a standard story node into a `for each` node, and use appropriate *edge guards*.

- As always, save and build your metamodel.

- Although the Learning Box GUI does not have an explicit action that invokes this SDM, feel free to extend it and see your SDM in action!

7 Inverting a card

This next SDM *inverts* a card by swapping its back and face values (Figure 7.1). This therefore "turns a card around" in the learning box. This action makes sense if a user wants to try learning, for example, the definition of a word in the other (target) language. Instead of guessing the definition of every word when presented with the term, perhaps they would like to guess the term when presented with the definition. This method doesn't need to accept any parameters – it'll use a bound `this` object variable.

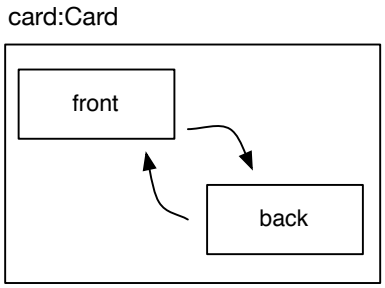


Figure 7.1: Inverting the attributes of a `Card`

Assignments

Something new that we'll use in this SDM are *assignments* to set the attributes of a `temp` object variable with `card`, then again to actually swap the `card` values. An assignment is simply an attribute constraint[15] with a `':='` operator. Though it may be slightly confusing to refer to an assignment as a constraint, if you think about it, *everything* can be considered as a constraint that must be fulfilled using different strategies.

With `invert`, a successful match is achieved not by searching as you would with a comparison (`==, >, <, ...`), but by *performing* the above assignment. If the assignment cannot be completed, the match is invalid. Similarly, non-context elements (set to create or destroy) can be viewed as structural constraints that are fulfilled when the corresponding element is created or destroyed. A constraint is therefore a unifying concept similar to "everything is an object" from OO, and "everything is a model" from metamodelling. If you're interested in why *unification* is considered cool, check out [**?**].

[15] Which we first encountered in `check`

7.1 Implementing invert

- If you've completed all the work so far, you've got to be *really* good at SDMs now. Model the simple story diagram depicted in Figure 7.2.

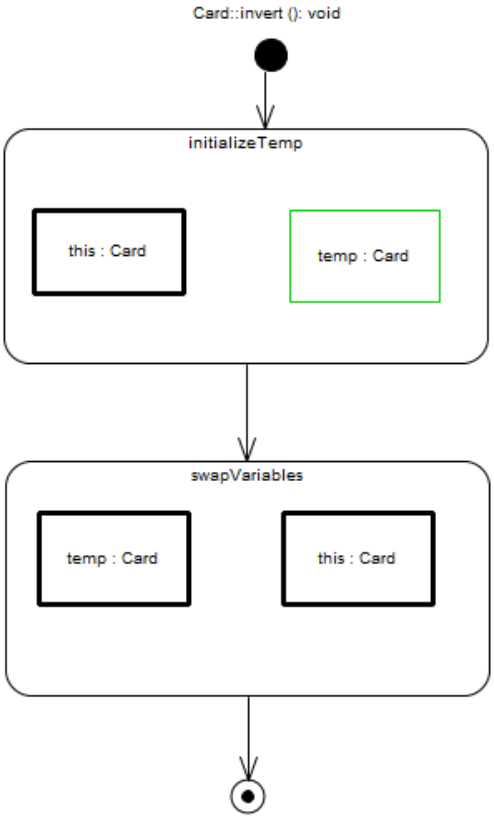


Figure 7.2: Imperative control layer for inverting a card

- Note that the binding operator on the first `temp` object variable is set to `create` (thus the green border). This means that we actually *create* a new object, and do not pattern match to an existing one in our model.
- This activity will need four assignment constraints - two in `initializeTemp` (to store the "opposite" values), and two in `swapVariables` (to switch the values). Create your first assignment constraint by going to the created `temp` card and using the '`:=`' operator to set the `temp.back` value to `this.face` (Figure 7.3).

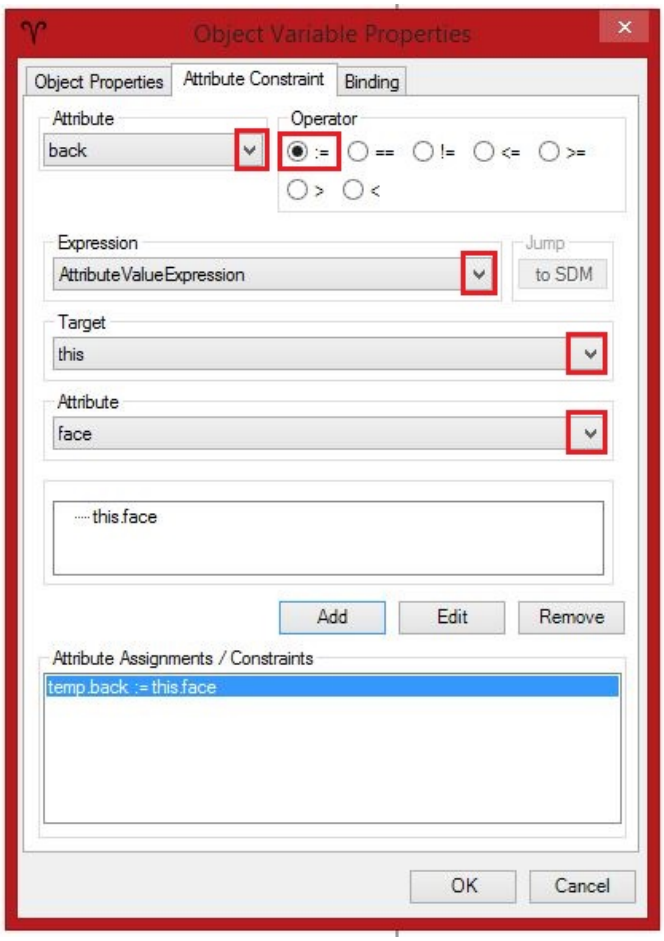


Figure 7.3: Store the `back` and `face` values of the card in `temp`

- Complete the SDM with the remaining constraints according to Figure 7.4 below.

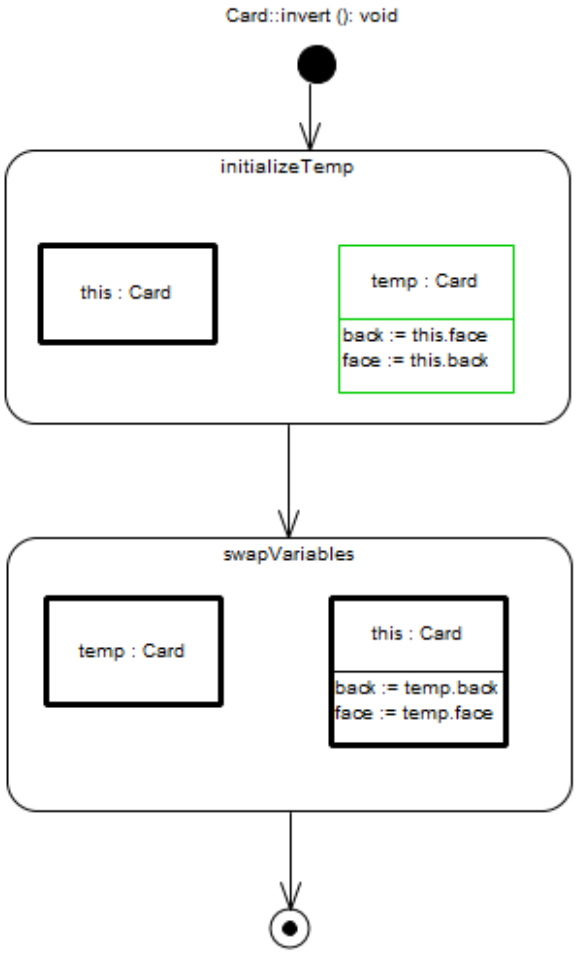


Figure 7.4: Swap back and face of the card

- Believe or not, that's it! You don't *have* to export and build to Eclipse, but it's always nice to confirm your work is error free.

Inversion review

Before we start the next SDM, let's quickly review one point. Have you considered why the `temp` object variable is bound in the second pattern for `invert`, (`swap variables`), but not where it's first defined in `initialize temp`?[16] This is a new case for bound variables that we haven't treated yet!

Until now, we have seen object variables that can be bound to (1) an argument of the method (set when the method is invoked), or (2) 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 before the pattern matcher starts. This means that it does not need to be determined or found by the pattern matcher.

Setting `temp` as bound in `Swap variables` is a third case in which an object variable is bound to a value determined in a *previous* activity node without using a special expression type. In this SDM, this means `temp` will be bound to the value determined for a variable of the same name in the previous node, `Initialize temp`. This binding feature enables you to refer to previous matches for object variables in the preceding control flow.

On a separate note, you're just over halfway through completing this part of the eMoflon handbook, so give your brain a small break. Take a walk, pour yourself another coffee, and check out one of my favourite jokes:

```
How do you wake up Lady Gaga?



Poke her face!
```

[16] See Figure 7.4

8 Growing the box

Ok, back to business. In this SDM, we shall explicitly specify how our learning box is to be built up. We create a specific pattern that will append new partition elements to the end of a `Box` that follow our established movement rules (Figure 1.1). This means the new partition will become the `next` reference of the current last partition, and its `previous` reference must be connected to the first partition in the box (Figure 8.1).

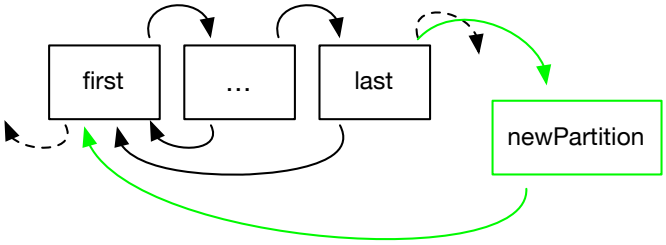


Figure 8.1: Growing a box by inserting a new partition

SDMs provide a declarative means of identifying specific partitions via *Negative Application Conditions*, simply referred to as NACs.[17] NACs express structures that are forbidden to exist before applying a transformation rule. In this SDM, the NAC will be an object variable that must not be assigned a value during pattern matching. In the theory of algebraic graph transformations [**?**], NACs can be arbitrarily complex graphs that are much more general and powerful than what we currently support in our implementation,[18] namely only single negative elements (object or link variables). *NAC*

As depicted in Figure 8.1, to create an appropriate NAC that constrains possible matches, we'll need to check to see if the currently matched pattern can be extended to include the negative elements. Suppose the current potential last partition has a `nextPartition`. This means it is *not* the absolute last partition, and so the match becomes invalid. We only want to insert a new partition when the `nextPartition` of the current potential last partition is null. Similarly, if the current potential first partition has a `previousPartition`, the match is invalid. The complete match is therefore made unique through NACs and thus becomes *deterministic* by construction. In other words, if you *grow* the box with this method, there will always be exactly one first and one last partition of the box.

Of course, to complete this method we still need to determine the size of the

[17] Pronounced $\backslash' nak \backslash$

[18] To be precise, in CodeGen2 from Fujaba

new partition. Since the size must be calculated depending on the rest of the partitions currently in the box (partitions usually get bigger) we'll need to call a helper method, `determineNextSize` via a *MethodCallExpression*. As the name suggests, it is designed to access any method defined in *any* class in the current project. *MethodCallExpression*

Due to the algorithmic and non-structural nature of `determineNextSize`, it will be easier to implement this method via a Java *injection*, rather than an SDM. We've already declared this method in our metamodel, so its signature will be available for editing in `BoxImpl.java`.

- Open "gen/LearningBoxLanguage.impl/BoxImpl.java." Scroll to the method declaration, and replace the contents with the code in Figure 8.2. Remember not to remove the first comment, which is necessary to indicate that the code is handwritten and needs to be extracted automatically as an injection. Please do not copy and paste the following code – the copying process from your pdf viewer to the Eclipse IDE will likely add invisible characters to the code that eMoflon is unable to handle.

```
public int determineNextSize() {
    // [user code injected with eMoflon]
    return getContainedPartition().size()*10;
}
```

Figure 8.2: Implementation of `removeCard`

- Save the file, then right-click on it, either in the package explorer or in the editor window, and choose "eMoflon/ Create/Update Injection for class" from the context menu.
- Confirm the update in the new `BoxImpl.inject` file's partial class. `determineNextSize` is now ready to be used by your metamodel!

8.1 Implementing grow

- Start by creating the simple story pattern depicted in Figure 8.3. This matches the box and *any* two partitions.[19]

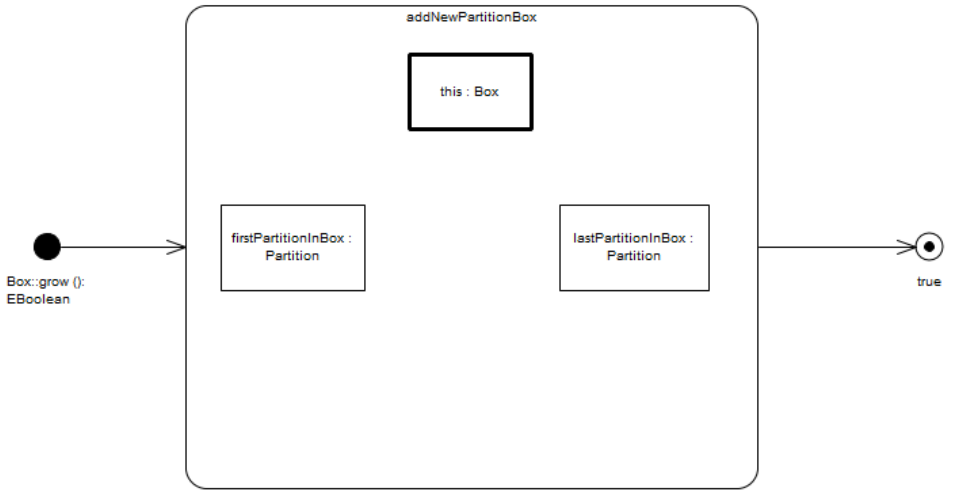


Figure 8.3: Context elements for SDM

- To create an appropriate NAC to constrain the possible matches for `lastPartitionInBox`, create a new `Partition` object variable `nextPartition` and set its *binding semantics* to `negative` (Figure 8.4). The object variable should now be visualised as being cancelled or struck out.

- Now, quick link `nextPartition` to `lastPartitionInBox`. Be sure to choose the link type carefully! The `nextPartition` should play the role of `next` with respect to `lastPartitionInBox`. This combination (the negative binding and reference) tells the pattern matcher that if the (assumed) last partition has an element connected via its `next` reference, the current match is invalid.

- Great work – the first NAC is complete! In a similar fashion, create the NAC for `firstPartitionInBox`. Name the negative element `previousPartition`, and again, be sure to double-check the link variable.

- Finally, complete the pattern so that it closely resembles Figure 8.5.

[19] Remember, the *pattern matcher* is non-deterministic.

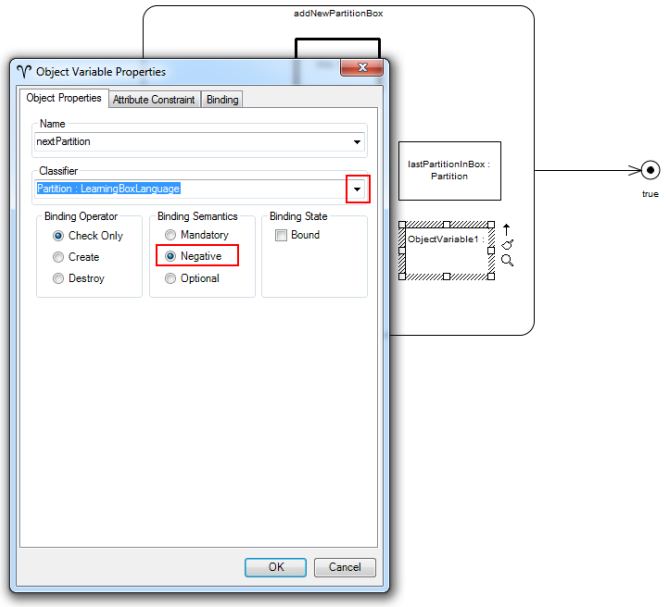


Figure 8.4: Adding a negative element

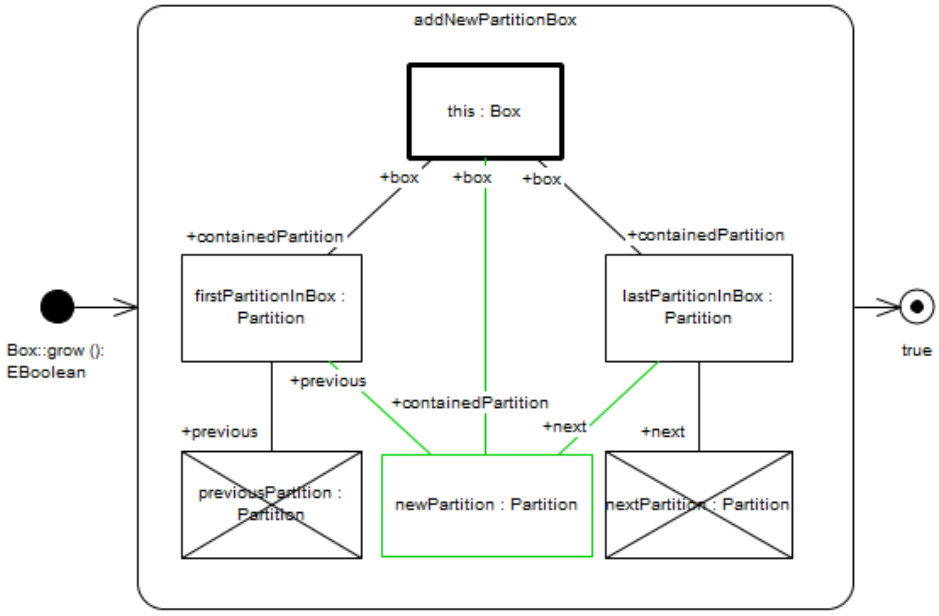


Figure 8.5: Determining the first and last partitions with NACs

- Notice how the created partition `newPartition` is 'hung' into the box. It becomes the next partition of the current *last* partition, and its previous partition is automatically set to the first partition in the box (as dictated by the rules set in Figure 1.1). In other words, the new partition is appended onto the current set of partitions.

- In order to complete `grow`, we need to set the size of the `newPartition`. Given that the new size is calculated via the helper function `determineNextSize`, we need to use a *MethodCallExpression*. Go ahead and invoke the corresponding dialogue, activate the assignment (`:=`) operator, and match your values to Figure 8.6.

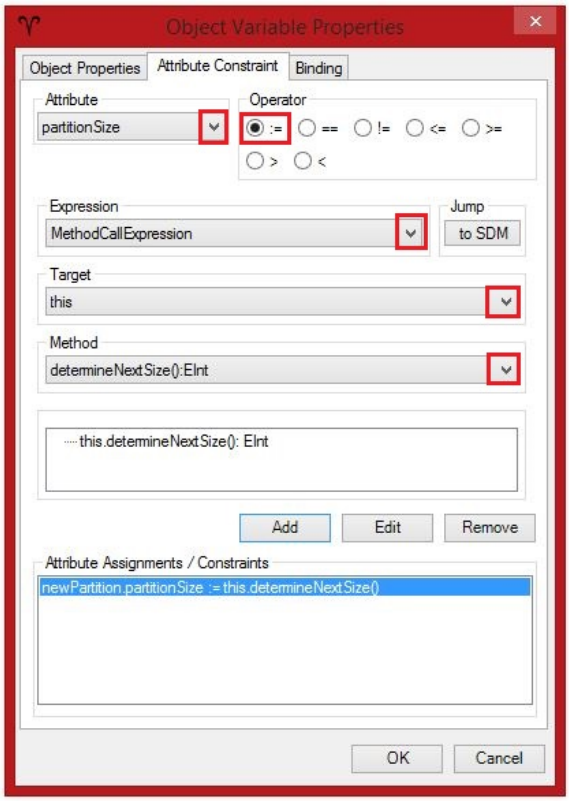


Figure 8.6: Invoking a method via a `MethodCallExpression`

- Since `determineNextSize` doesn't require any parameters, you can ignore the `Parameter Values` field this time.

- If you've done everything right, your SDM should now closely resemble Figure 8.7. As usual, try to export, generate code, and inspect the method implementation in Eclipse.

▶ That's it – the `grow` SDM is complete! This was probably the most challenging SDM to build so give yourself a solid pat on the back. If you found it easy, well then . . . I guess I'm doing my job correctly.

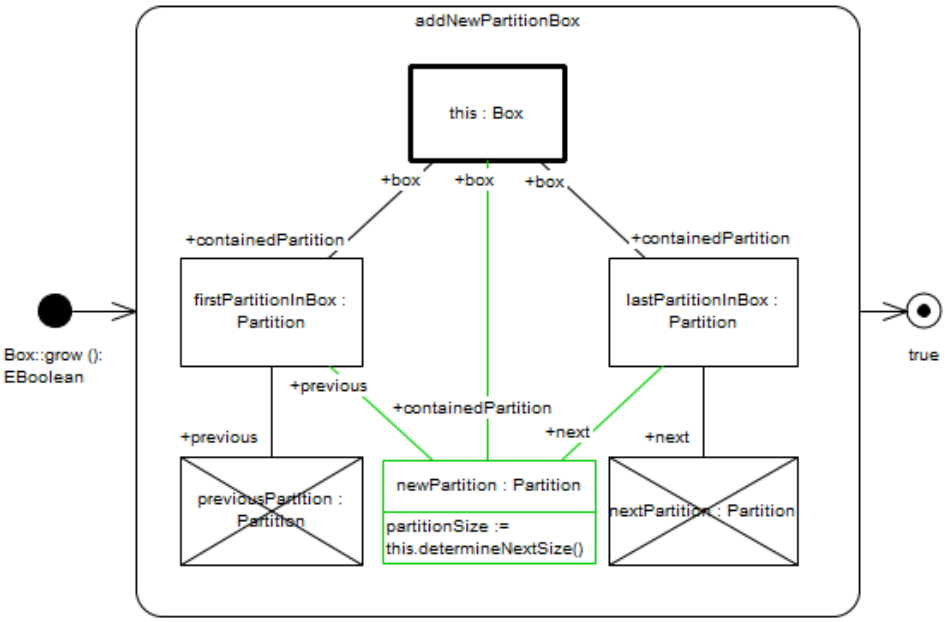


Figure 8.7: Complete SDM for `Box::grow`

9 Conditional branching

When working with SDMs, you'll often find yourself needing to decide which statement(s) to execute based on the return value of an arbitrary (black box) operation, as we saw in `check`. In our example so far, we have implemented these constructs via SDM *pattern matching*.

With eMoflon however, there is an alternate way to construct these black boxes. In fact, this feature is yet another way of integrating handwritten Java code with your SDM. We can invoke methods directly from an *if* statement. The only "rule" of this feature is that the method must return an `EBoolean` to indicate `Success` or `Failure`, corresponding to `true` or `false`, respectively. Any other types imply `Failure` if the return value of the method is `null`. It follows that void methods cannot be used for branching – an exception will be thrown during code generation (if you ignored the validation error).

Statement Node

Unfortunately, you can't simply invoke a method from a standard activity node. Instead, you must use a new type of activity node, a *statement node*. Statement nodes can be used to invoke methods 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 by hand or via an SDM. Thus, statement nodes can of course be used to invoke other SDMs via a *MethodCallExpression*. Most importantly, statement nodes enable *recursion*, as the current SDM can be invoked on `this` with appropriate new arguments. In essence, this type of node is only used to guarantee a specific action between *activity nodes*, and does not extend the current set of matched variables. They can however, be used as a conditional by branching on whatever value the method returns.

Let's reconsider `grow`, the method we just completed that adds a new partition to our box. Reviewing Figure 8.7, the current pattern assumes there are already at least two partitions in `box` (the `firstPartitionInBox` and `lastPartitionInBox`). What would happen if `box` had only one, or even no partitions at all? The pattern would *never* find a match!

To fix this problem, let's modify `grow` so that if the original match fails, we initialize two new partitions (the first and last), but *only* if it failed due to the box being completely empty. In other words, if `box` has e.g., only one partition (an invalid state that cannot be reached by growing from zero partitions), it is considered invalid and no longer be grown.

9.1 A short note on the `initializeBox` method

Pretend you've just updated the control flow in your `grow` SDM, and haven't specified `initializeBox` yet. After saving and building, you will be able to see the changes in `BoxImpl.java`, the source file containing the generated code. In fact, open this file now and navigate to `grow`, which starts at (approximately) line 207 (Figure 9.1). This is the generated *statement node* code and, as you can see, all it does is invoke your method and branch based on its result.

```
} else {
    // initialize
    if (BoxImpl.pattern_Box_2_3_expressionFB(this)) {
        return BoxImpl.pattern_Box_2_4_expressionF();
    } else {
        return BoxImpl.pattern_Box_2_5_expressionF();
    }

}
```

Figure 9.1: Code generated for branching with a statement node

Go to `initializeBox`, it should look like Figure 9.2.

```
256  public boolean initializeBox() {
257      // [user code injected with eMoflon]
258
259      // TODO: implement this method here but do not remove the injection marker
260      throw new UnsupportedOperationException();
261  }
```

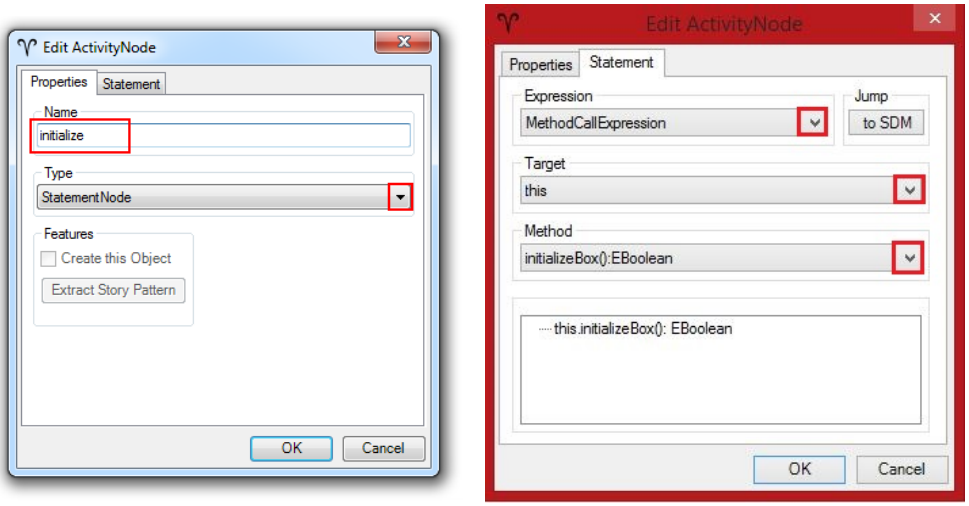
Figure 9.2: The `initializeBox` declaration

You have the choice of either implementing the method by hand here in Java as an injection, or you can return to the metamodel and implement it there as an SDM. The statement node will work just fine in both cases.

Using Java and injections makes sense if the method is non-structural, but seeing as we must check to see if there is a single partition, then create the first two partitions of the box if it succeeds, `initializeBox` is actually quite structural and can be described beautifully as a pattern. This is why we opted to specify it as an SDM.

9.2 Branching with statement nodes

- Currently, there is no method to help us initialize `box` from its pristine state (no partitions). Create one by editing your metamodel (the `LearningBoxLanguage` diagram) and invoking the `Operations` dialogue by first selecting `Box`, then pressing `F10`.[20]
- Name the new method `initializeBox` and, recalling the one rule of conditional branching, set its return type to `EBoolean`.
- Save and close the dialogue, then re-open the `grow` SDM and *Quick Create* a new story node from `addNewPartition`.
- This will be the node we'll use to invoke our helper method. Double click the node to invoke its properties editor and switch the `Type` to a `StatementNode`. Name it `initialize` (Figure 9.3a).
- Before closing the dialogue, switch to the `Statement` tab, and create a `MethodCallExpression` to invoke your newest method (Figure 9.3b). We want to access the `Box` object (`this`) and its `initalizeBox` method. It doesn't require any parameters, so leave the values field empty.



(a) Create a new *StatementNode* (b) Edit the `MethodCallExpression`

Figure 9.3

[20] To review creating new operations, review Section 2.6 of Part II

- Now we need to update the edge guards stemming from `addNewPartitionInBox`. Given that we only want to call `initializeBox` if the pattern fails, change the edge guard leading to your statement node to `Failure`. Similarly, update the edge guard returning `true` to `Success`.
- Finally, attach two stop nodes – `true` and `false` – along with their appropriate edge guards from `initialize`. These indicate that if the method execution worked, the box could be initialized. If it failed however, `box` was in an invalid state (by e.g., having only one partition) and returns `false`. Overall, the new additions to `box.grow()` should resemble Figure 9.4.

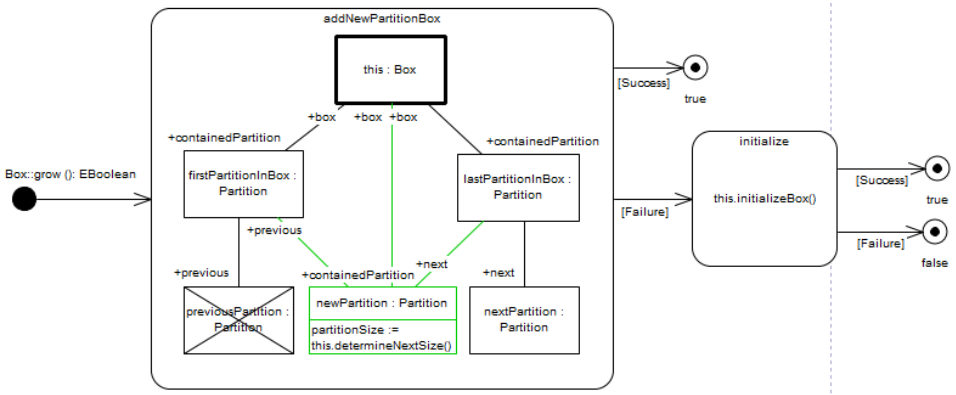


Figure 9.4: Extending `grow` with a *MethodCallExpression*

- To review our work up to this point, we have declared `initializeBox` and invoked it from a statement node. We have yet to actually specify the method however. Double-click the anchor to return to the main diagram and create a new SDM for `initializeBox`.
- Create a normal activity node named `buildPartitions` with the pattern depicted in Figure 9.5.

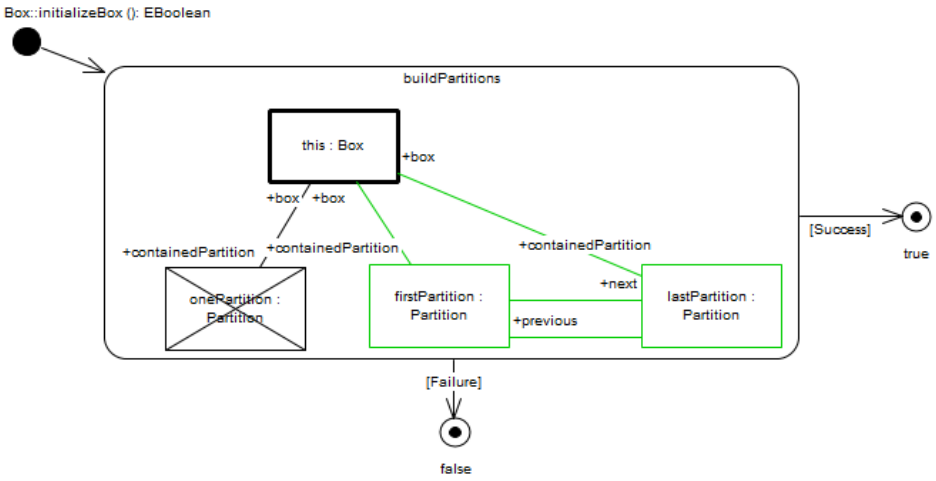


Figure 9.5: Complete SDM

- The NAC used here is only fulfilled if the box has absolutely no partitions, i.e., is in a pristine state and can be initialized. In other words, if `grow` is used for an empty box, it initializes the box for the first time and grows it after that, ensuring that the box is always in a valid state.
- You're finished! Save, validate, and build your metamodel.

10 A string representation of our learning box

In the next SDM we shall create a string representation for all the contents in a single learning box. To accomplish this, we will have to iterate through every card, in every partition. The concept is similar to `Partition`'s `empty` method, except we'll need to create a nested *for each* loop (Figure 10.1). Further still, we'll need to call a helper method to accumulate the contents of each card to a single string.

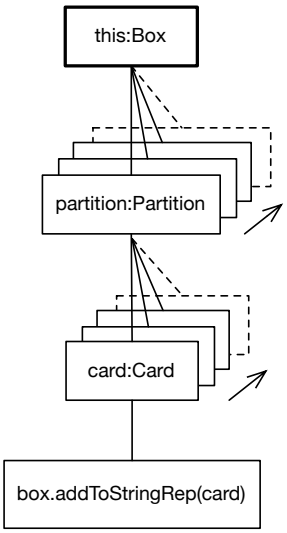


Figure 10.1: Nested *For Each* loops

As you can see, The first loop will match all partitions, while the second matches each card. Finally, a *statement node* is used to invoke the `addToStringRep` method. In contrast to how they were used for conditional branching in `grow`, this statement node will simply invoke a void method.

Unlike `initializeBox` however, this helper method is actually better specified as an injection so, analogously to how you implemented `determineNextSize` for `box.grow()`, quickly edit `BoxImpl.java` by replacing the default code for `addToStringRep` with that in Figure 10.2. You can use Eclipse's built-in auto-completion to speed up this process. Save, create the injection file, and confirm the contents of `BoxImpl.inject`.

```
public void addToStringRep(Card card) {
        // [user code injected with eMoflon]
        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 10.2: Implementation of `addToStringRep`

10.1 Implementing toString for `Box`

▶ Visual SDMs support arbitrary nesting of *for each* story nodes via special guards. In Section 5.1 we used the `[end]` edge guard to terminate a loop. Now we'll use a new guard, the `[each time]` guard, to indicate control flow that is *nested* and executed for each match. Go ahead and create the SDM for `Box::toString` until it closely resembles Figure 10.3. *[each time]*

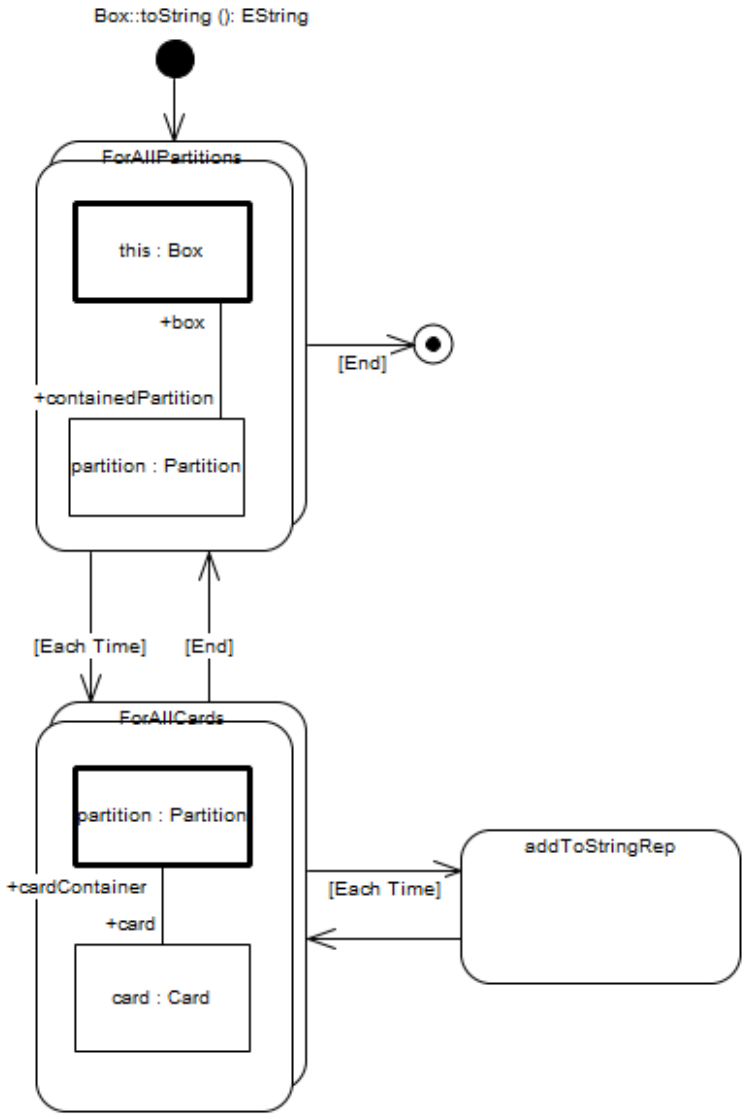


Figure 10.3: Control flow with nested loops

- Knowing `addToStringRep`'s default node type will not allow it to invoke our helper method, change it into a `StatementNode`. Then, analogously to how you established a *MethodCallExpression* for `grow`, have this node invoke `this.addToStringRep(Card)` (Figure 10.4).

- You can see in the `Method` statement that a `Card` parameter is required. For this double click on the field below and enter the values like in Figure 10.5. Now we included `card` so we may pass the object variable to the method.

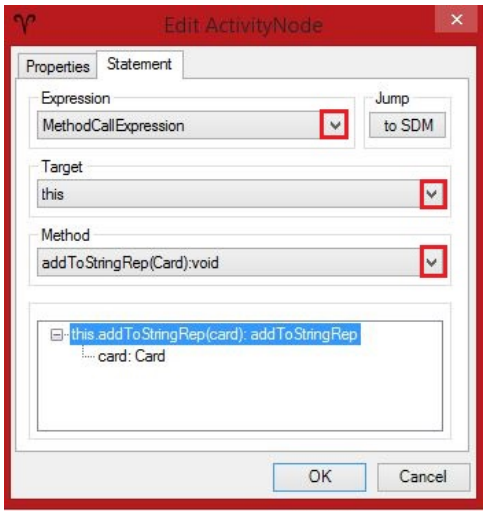


Figure 10.4: Add a *MethodCallExpression*

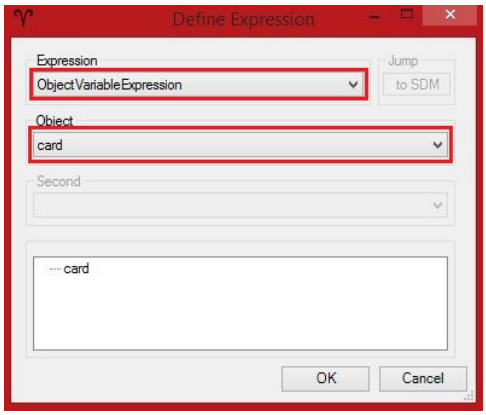


Figure 10.5: Add a parameter to the *MethodCallExpression*

- To complete the SDM, return the final string representation value of the box via an *AttributeValueExpression* in the stop node (Figure 10.6). This is a new expression type we haven't encountered before. It simply binds the `stringRep` attribute of the box (`this`) to the return value in the stop node

AttributeValueExpression

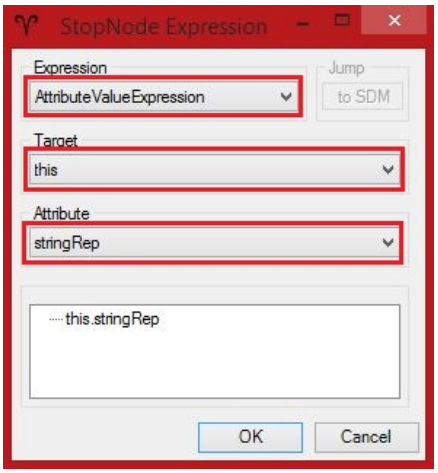


Figure 10.6: Specify a return value as an *AttributeValueExpression*

- Take some time to compare and reflect on the complete SDM as depicted in Figure 10.7. The idea was to abstract from the actual text representation of the box and model the necessary traversal of the data structure. The helper method `addToStringRep` could, for example, build up something totally different.

- While modelling this SDM, we have seen that *for each* story nodes can be nested, and have used a *MethodCallExpression* to invoke a void helper method only for its side effects (building up the string representation of the box).

- As always, save, validate, and build your metamodel in Eclipse.

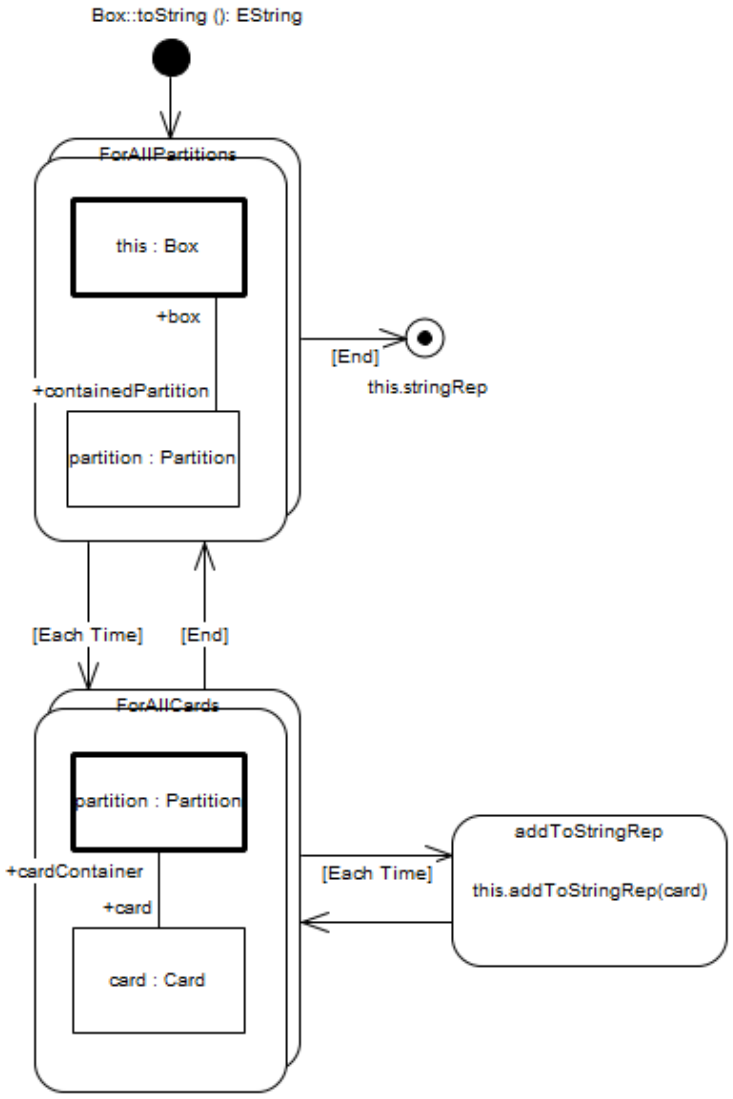


Figure 10.7: The complete SDM for `Box::toString`

11 Fast cards!

Congratulations, you're almost there! This is the last SDM needed before your Leitner's learning box is fully functional.

For very simple cards (i.e., words in a different language that are quite similar), it might be a bit annoying to have to answer these cards again and again in successive partitions. Such *fast* cards should somehow be marked as such and handled differently. If a fast card is correctly answered once, it should be immediately moved to the final partition in the box. This way, the card is practiced once, and only tested once more before finally being ejected from the box.

It makes sense for a `FastCard` to inherit from `Card`, so we'll extend the current object in our metamodel by a new `EClass` for fast cards, depicted below with a marker to show it behaves differently (Figure 11.1).

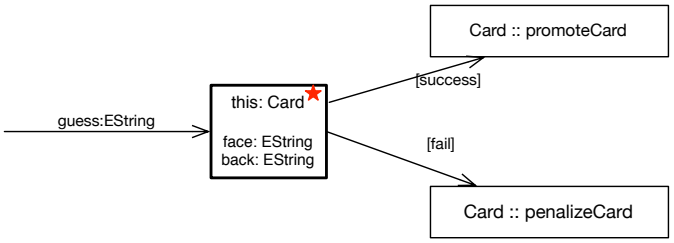


Figure 11.1: Checking a fast card against a guess

In addition to creating a new `EClass`, we also need to extend the existing `check` method to check for this special card type once a guess is determined to be correct. Now `check` needs to decide, based on the dynamic type[21] of `card`, if it needs to handle this special fast card. This can be expressed in SDMs with a *BindingExpression* (or just *Binding*). A binding can be specified for a *bound* object variable and is the final case in which an object variable can be marked as being bound. *Binding*

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.

Finally, this new pattern faces a similar challenge as `grow`. A `FastCard` can't simply progress to the `next` partition. It must skip ahead to the absolute last partition in the box. This means yet another NAC is required to determine the last partition in a `Box`.

[21] 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).

11.1 Implementing FastCards

- To introduce fast cards into your learning box, return to the metamodel diagram and create a new `EClass`, `FastCard`. Quick link to `Card` and choose `Create Inheritance` from the context menu. We only want to check the dynamic type of a tested card at runtime, which means we don't need to override anything. Therefore, when the `Overrides & Implementations` dialogue appears, make sure nothing is selected (Figure 11.2). Your metamodel should then resemble Figure 11.3.

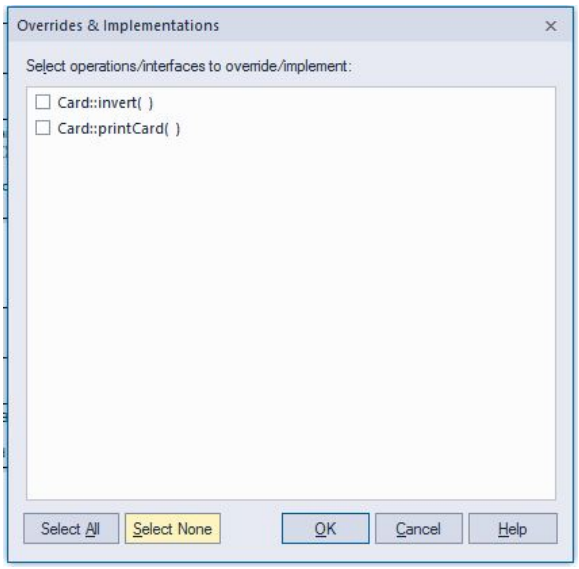


Figure 11.2: Selecting operations to override

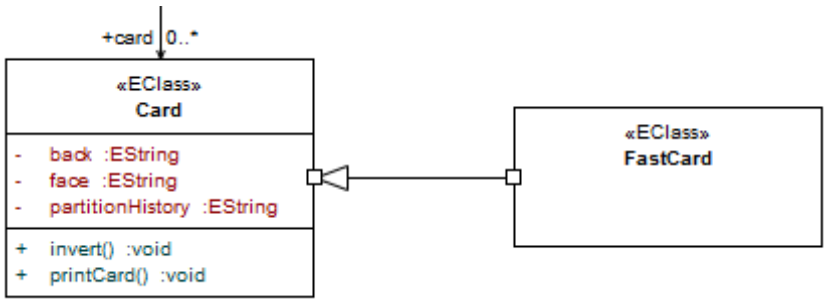


Figure 11.3: Fast cards are a special kind of card

- Now return to the `check` SDM (in `Partition`) and extend the control flow as depicted in Figure 11.4.

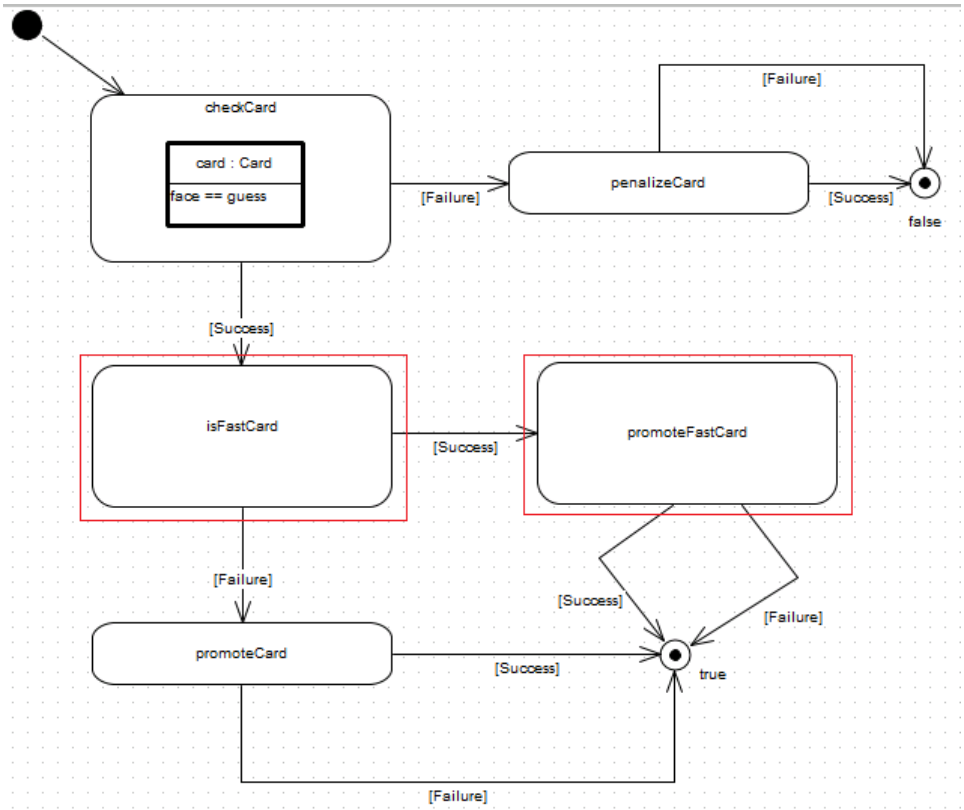


Figure 11.4: Extend check to handle fast cards.

- As you can see, you have created two new story nodes, `isFastCard`, and `promoteFastCard`.
- Next, in order to complete the newest conditional, create a bound `FastCard` object variable, named `fastcard` in `isFastCard` (Figure 11.5).
- To check the dynamic type, we'll need to create a binding of `card` (of type `Card`) to `fastcard` (of type `FastCard`), so edit the `Binding` tab in the `Object Variable Properties` dialogue (Figure 11.5). Please note that this tab will not allow any changes unless the `bound` option in `Object Properties` is selected. As you can see, this set up configures the pattern matcher to check for types, rather than `parameters` and `attributes` as we've previously encountered.

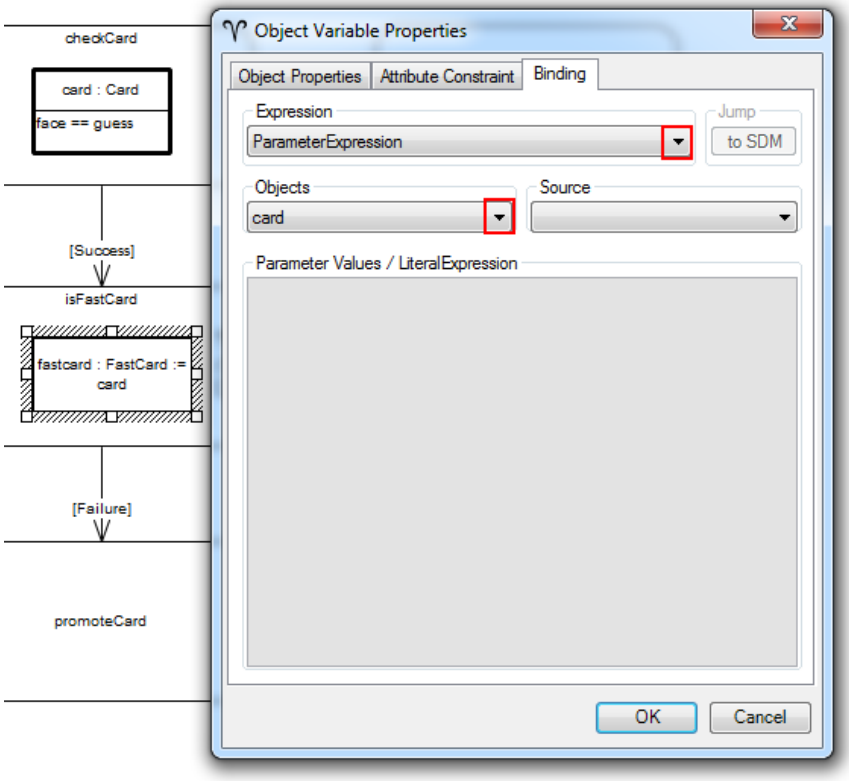


Figure 11.5: Create a binding for `fastcard`

In our case, we could use a *ParameterExpression* or an *ObjectVariableExpression* as `card` is indeed a parameter *and* has already been used in `checkIfGuessIsCorrect`. We haven't tried the latter yet, so let's use *ObjectVariableExpression*.

ObjectVariableExpression

- Update the `fastcard` binding by switching the expression to `ObjectVariableExpression`, with `card` as the target. Note that a binding could also use a *MethodCallExpression* to invoke a method whose return value would be the bound value. This is very useful as it allows invoking helper methods directly in patterns.

- To finalize the SDM, (i) extract the `promoteFastCard` story pattern and build the pattern according to Figure 11.6 and (ii) create the parallel `Success` and `Failure` edges from this activity to the stop node returning `true` for the same reason as in `check` earlier. Compare the pattern in Figure 11.6 to Figure 4.13 and **??**, the original promotion and penalizing card movements. As you can see, they're very similar, except `fastCard` is transferred from the current partition (`this`) immediately to the last partition in `box`, identified as having no next `Partition` with an appropriate NAC. Note that a second NAC is used to handle the case where `this` would be the next `Partition`, which is also not what we want.

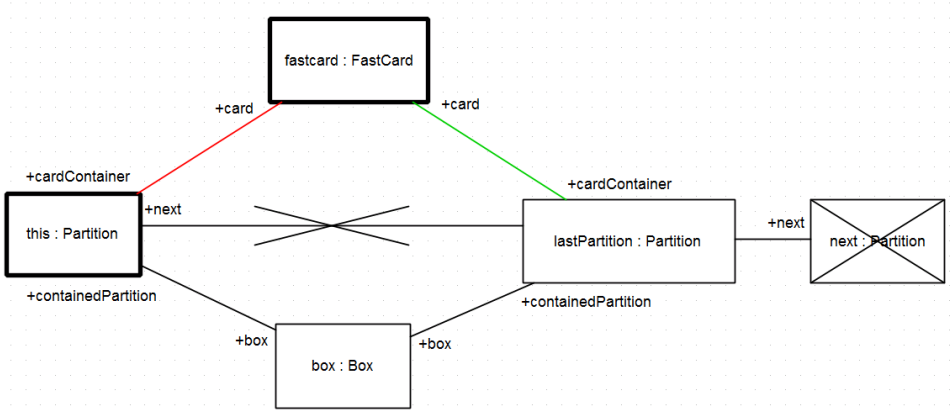


Figure 11.6: Story pattern for handling fast cards.

- You have now implemented every method using SDMs – fantastic work! Save, validate, and build your metamodel to see some new code. Inspect the implementation for `check`. Can you find the generated type casts for `fastcard`?

11.2 FastCards in the GUI

We hope you haven't forgotten about the GUI! Now that we have a new `card` type, let's quickly try editing our `box` instance so we can experiment with them in our application.

- To review the details of creating instances, read Part II, Section 3 but for now, open `Box.xmi`, right-click on your first partition, and create a new `FastCard` child element. Open the properties tab below, and edit the `back` and `face` values for testing (Figure 11.7). As you can see, it has all the same attributes as a standard `card`.

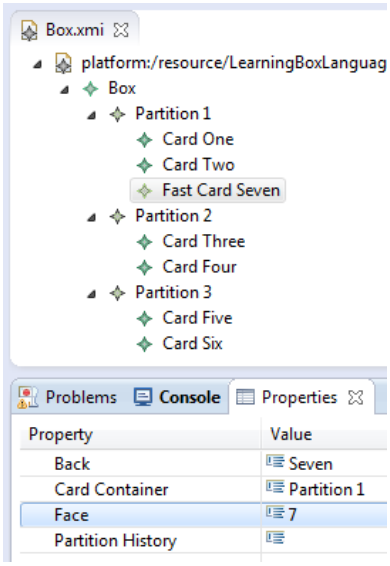


Figure 11.7: Creating and editing a new `FastCard` element

- Save your file, then open the GUI and try your extended `check` method. Experiment with making wrong and correct guesses for both card types, paying attention their behaviour. If you've done everything right until this point, your newest `FastCard` should act differently than its standard `card` counterparts.

12 Reviewing eMoflon's expressions

As you've discovered while making SDMs, eMoflon employs a simple context-sensitive expression language for specifying values. We have intentionally avoided creating a full-blown sub-language, and limit expressions to a few simple types. The philosophy here is to keep things simple and concentrate on what SDMs are good for – expressing structural changes. Our approach is to provide a clear and type-safe interface to a general purpose language (Java) and support a simple *fallback* via calls to injected methods as soon as things get too low-level and difficult to express structurally as a pattern.

The alternative approach to eMoflon would be to support arbitrary expressions, for example, in a script language like JavaScript or in an appropriate DSL[22] designed for this purpose.

We've encountered several different expression types throughout our SDMs so far, and all of them can be used for binding expressions. Since each syntax has used at least three of these once, let's consider what each type would mean:

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

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 (i.e., a subtype), this represents basically a successful typecast if the pattern matches.

[22] 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.

ObjectVariableExpression:

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

13 Complex Attribute Manipulation

Note: Complex attribute manipulation is an optional feature of eMoflon. If you are in a hurry, just skip this section.

Instead of going through the whole of this part, you may simply fetch the final state of the handbook example via *Install Workspace →eMoflon Examples →Handbook Part 3 Final*.

In Section 4.1 we have modeled the `Partition::check()` method. However, as you might realized, our implementation ignores the `partitionSize`, i. e., the maximal number of cards that fit in a partition. To prevent our implementation from "bursting" a partition we will use *complex attribute constraints* to extend the SDM for the `Partition::check()` method. In contrast to simple attribute constrains[23], which provide only binary operations for comparing attribute values (or literals), complex attribute constraints provides a language to specify arbitrary relations on attributes.

13.1 Using Complex Attribute Constrains

We start by extending the `penalizeCard` pattern of the `Partition::check()` method in such a way that it only moves a card to the previous partition if the additional card does not exceed the actual `partitionSize` of the previous partition. First, we extend `Partition` class by an attribute that keeps track of the actual number of card contained in a partition.

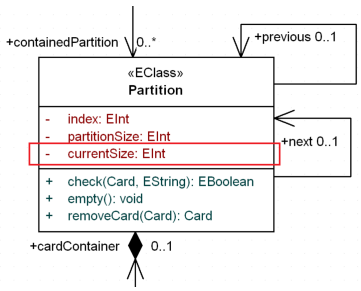


Figure 13.1: Adding the attribute `currentSize` of type `EInt` to `Partition`

▶ Add to the class `Partition` the new attribute `currentSize:EInt` (Figure 13.1).

Now we start defining complex attribute constraints.

[23]First used in Section 4.1 to compare user's guess against the card's face value

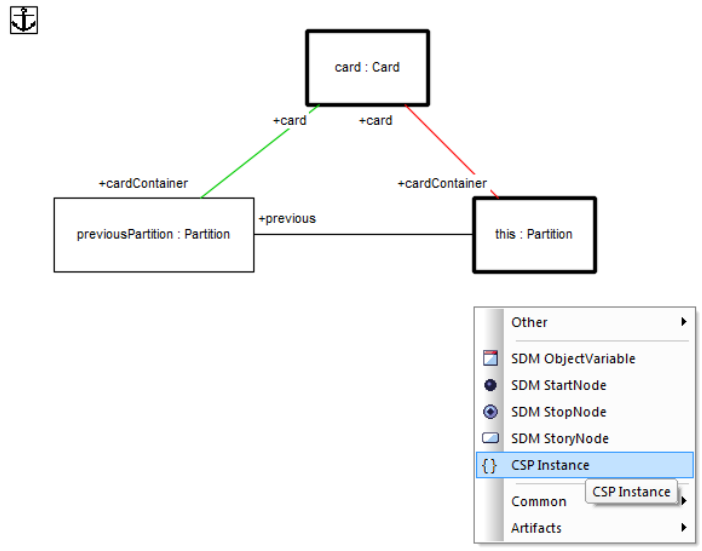


Figure 13.2: Creating a new `CSP instance`

- Navigate to the `penalizeCard` pattern in the `Partition::check(Card,EString)` method and add a `CSP instance`. Following a similar process as creating a new object variable, i. e., either hit space or use the toolbox to create a new *CSP instance* (Figure 13.2).
- You'll notice a box were you can specify your complex attribute constraints. Enter the attribute constraints as shown in Figure 13.3 to define that a card is only moved if the additional card does not exceed the actual `partitionSize` of the previous partition.

Before filling the text area with something meaningful, let's find out more about the general concepts of complex attribute constraints. Basically, an attribute constraint is a predicate of the form

$\texttt{SYMB(param}_1\texttt{,...,param}_n\texttt{)}$

- A *predicate* `SYMB` specifies the relation of the affected parameters $\texttt{param}_1$, ..., $\texttt{param}_n$, e. g., equality `=` or inequality `!=`. A parameter can be either a local variable, a constant or an attribute value, as defined in the following items.
- A *local variables* of the form `name:type`, e. g., `newCurrentSize : EInt` in Figure 13.3 defines an local variable `newCurrentSize` of type `EInt`.

- A *constant* is of the form `value::type` (not the *2* colons), e. g., `1::EInt` represents constant 1 of type `EInt`.
- An *attribute variable* is of the form `objectVariableName.attributeName`, and refers to the current value of the attribute `attributeName` of object variable `objectVariableName`, while an attribute variable of the form `objectVariableName.attribteName'` refers to an attribute value after the transformation. This is necessary, e. g., to increment the attribute `a` of variable `x` as follows: `+(x.a', x.a, 1::EInt)`.
- A *package import* is of the form `importPackage packageName;` and specifies that the package `packageName` should be used to search for types. For instance, `importPackage Ecore;` is required to refer to the type `EInt`.

Enough theory for now: Let's get back to the example.

▶ Insert the following lines (`importPackage Ecore;` should be already there) to reflect Figure 13.3. Especially, make sure that the *NAC index* is set to `-1`. Double-slashes may be used to add single-line comments.

```
// Required to use (e.g.,) datatype EInt
importPackage Ecore;

// Introduces a temporary variable to store the (tentative) new
// size of the previous partition
+(newCurrentSize:EInt, previousPartition.currentSize, 1::EInt);

// Check that the tentative size does not exceed the capacity
// of the previous partition.
<=(newCurrentSize:EInt, previousPartition.partitionSize);

// Decrement size of current partition
-(this.currentSize', this.currentSize, 1::EInt);

// Increment size of previous partition
=(previousPartition.currentSize', newCurrentSize:EInt);
```

▶ Export the project as usual. Before generating code, you have to switch the code generation engine. Open `moflon.properties.xmi` located in your project and change the *SDM Codegenerator Method Body Handler* to `DEMOCLES_ATTRIBUTES` (Figure 13.4).

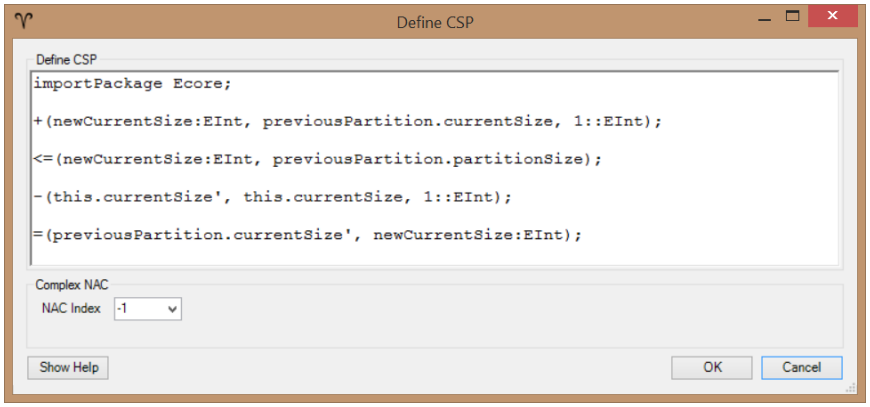


Figure 13.3: Attribute constraints that prevent exceeding the partition size

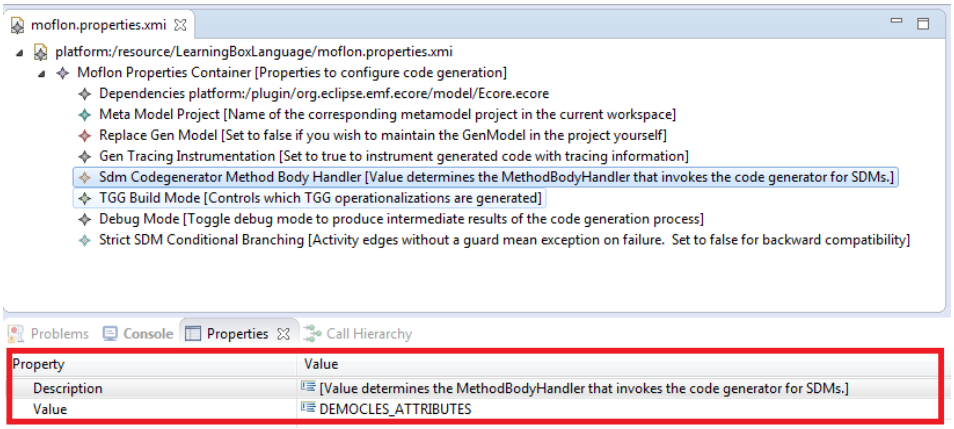


Figure 13.4: Switching SDM code generator to `DEMOCLES_ATTRIBUTES`

The code generator engine is now able to solve the CSP instances, i. e., the constraints are operationalized and put in correct order. The code resulting for the CSP instance is shown in Figures 13.5 and 13.6 for the LHS and RHS, respectively.

```
int this_currentSize = _this.getCurrentSize();

int previousPartition_currentSize =
  previousPartition.getCurrentSize();

int previousPartition_partitionSize =
  previousPartition.getPartitionSize();

int this_currentSize_prime = this_currentSize - 1;

int newCurrentSize = previousPartition_currentSize + 1;

if (newCurrentSize <= previousPartition_partitionSize) {
    int  previousPartition_currentSize_prime = newCurrentSize;
    // return match
}
```

Figure 13.5: LHS pattern code generated for the CSP instance

```
_this.setCurrentSize(Integer.valueOf(this_currentSize_prime));

previousPartition.setCurrentSize(
  Integer.valueOf(previousPartition_currentSize_prime));
```

Figure 13.6: RHS pattern code generated for the CSP instance

Notice that the attribute variables `this.currentSize'` and `previousPartition.currentSize'` (represented in the code by `this_currentSize_prime` and `previousPartition_currentSize_prime`, respectively) are treated as local variables on the LHS and then assigned on the RHS to the corresponding attribute values.

13.2 Defining Your Own Complex Attribute Constraints

In Section 10 we have implemented the `Box::toString()` method to obtain a string representation of the box. Internally, this method uses the method `Box::addToStringRep(Card) : EString` to create a string from a card, that was realized using handwritten code via injections.

In the following, we replace the method `Box::addToStringRep(Card):EString` by a complex attribute constraint.

- Reconsider the SDM shown in Figure 10.7. First, add an attribute constraint for appending the string `"Content: "` to the `stringRep` attribute of `Box`. To this end, add to the `ForAllPartitons` pattern the following complex attribute constraint:

  ```
  +(this.stringRep', this.stringRep, "Content: "::EString)
  ```

Note that the predicate `+` is overloaded: For `EInt`, `+` means integer addition, while for `EInt`, it means concatenation.

In a next step, the pattern `ForAllCards` is extended by an complex attribute constraints such that for each card the string containing the front and back is appended to *this.stringRep*. Instead of using the the concatenation predicate `+` again, we just define our own constraint as follows.

- Add a `CSP instance` with the following content to the `ForAllCards` pattern.

  ```
  myConcat(this.stringRep', this.stringRep, card.face, card.back);
  ```

  The complete SDM should now look as shown in Figure 13.7. Note that `importPackage Ecore;` can be omitted in the constraints of `ForAllCards` because no (Ecore) data types are used there.

- Export as usual. If you are now building the metamodel project in Eclipse, you get an error message (Figure 13.8) that informs you that the attribute constraint with signature
  `myConcat( :EString, :EString, :EString, :EString)`
  is unknown.

- Look inside `LearningBoxLanguage/lib/`. You find a new file called `LearningBoxLanguageAttributeConstraintsLib.xmi`

- Open the file. It contains a constraint specification `myConcat` (Figure 13.9 bottom) that represents the signature (`myConcat( :EString, :EString, :EString, :EString)`), which is derived from the information of the CSP-instance in EA. To define the meaning of `myConcat` we have to define operations for the constraint.

- Right click the operation specification group for `myConcat` (top of Figure 13.9) and add (as a new child) an `operation specification`.

An operation[24] is specified by an *Adornment String*, and a `Specification` that is a template defining the code to be generated.

[24] Double click on the operation specification to open the properties view.

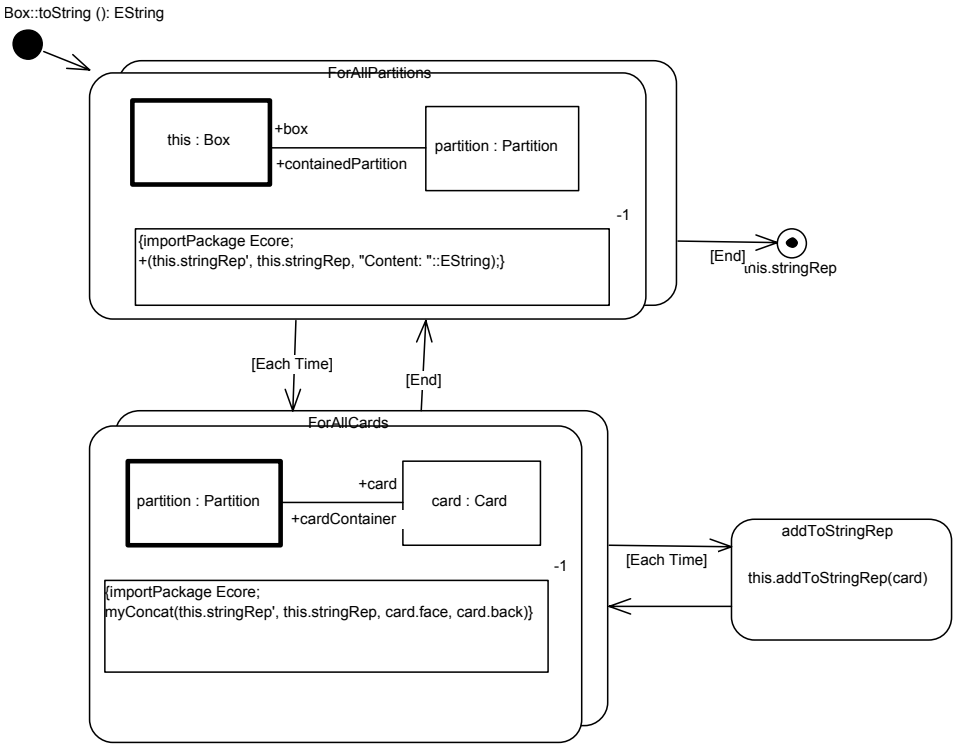


Figure 13.7: Complete SDM for `Box::toString()`

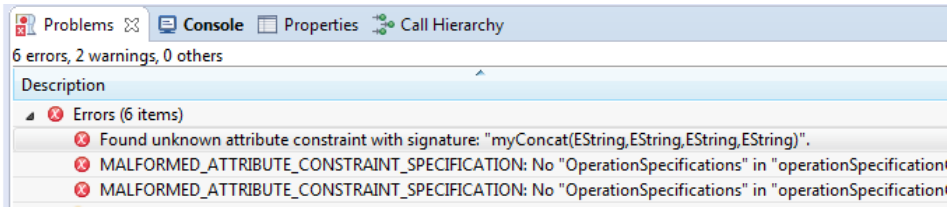


Figure 13.8: Error

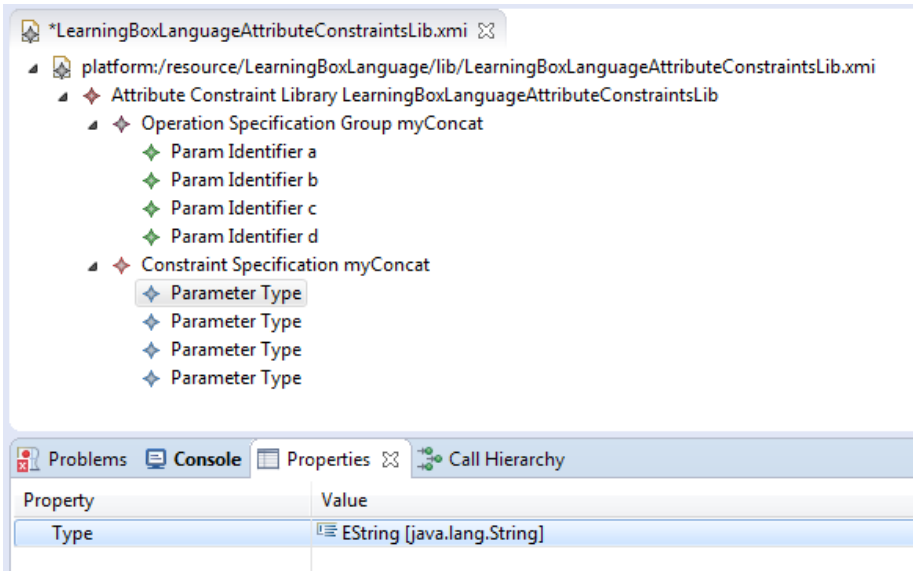


Figure 13.9: `LearningBoxLanguageAttributeConstraintsLib.xmi`

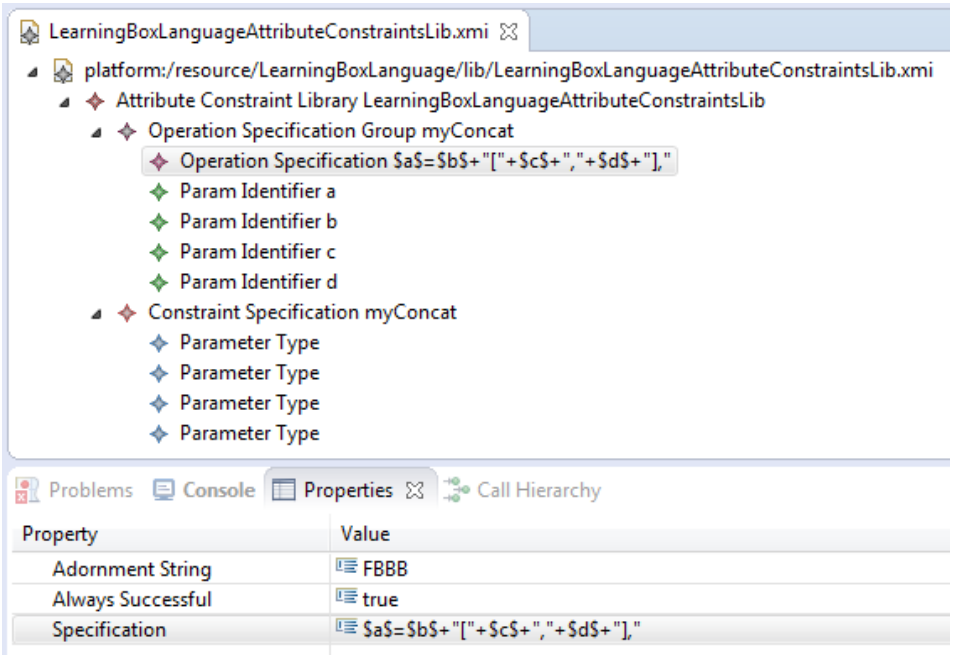


Figure 13.10: Operation specification for `myConcat`

- Complete the operation specification as shown in Figure 13.10. The adornment string `FBBB` means that before the operation can be executed the values for parameter `b`, `c`, and `d` must be known (i. e., they are `b`ound), and that parameter `a` is unknown (i. e., it is a `f`ree parameter). Therefore, in the `Specification` template, we add a formula that describes how `a` can be derived based on `b`, `c`, and `d`. Inside the template code, the variables are refered to using `$`. The surrounding code has to be valid Java code.

13.3 Built-in Attribute Constraint Types

Symbol	**Signatures**	**Semantics**
=	(a:Eint , b:EInt) (a:EDouble , b:EDouble) (a:EFloat , b:EFloat) (a:EShort , b:EShort) (a:ELong , b:ELong) (a:EString , b:EString)	$(a := b)$ for `FB` $(a == b)$ for `BB`
<=	(a:Eint , b:EInt) (a:EDouble , b:EDouble) (a:EFloat , b:EFloat) (a:EShort , b:EShort) (a:ELong , b:ELong)	$(a \leq b)$ for `BB`
<	(a:Eint , b:EInt) (a:EDouble , b:EDouble) (a:EFloat , b:EFloat) (a:EShort , b:EShort) (a:ELong , b:ELong)	$(a < b)$ for `BB`
>=	(a:Eint , b:EInt) (a:EDouble , b:EDouble) (a:EFloat , b:EFloat) (a:EShort , b:EShort) (a:ELong , b:ELong)	$(a \geq b)$ for `BB`
>	(a:Eint , b:EInt) (a:EDouble , b:EDouble) (a:EFloat , b:EFloat) (a:EShort , b:EShort) (a:ELong , b:ELong)	$(a > b)$ `BB`
+	(a:Eint , b:EInt, c:EInt) (a:EDouble, b:EDouble,c:EDouble)	$(a = b + c)$ for `FBB`

Symbol	**Signatures**	**Semantics**
	(a:EFloat , b:EFloat, c:EFloat) (a:EShort , b:EShort, c:EShort) (a:ELong , b:ELong, c:ELong) (a:EString , b:EString, c:EString)	
$-$	(a:Eint , b:EInt, c:EInt) (a:EDouble,b:EDouble,c:EDouble) (a:EFloat , b:EFloat, c:EFloat) (a:EShort , b:EShort, c:EShort) (a:ELong , b:ELong, c:ELong)	$(a = b - c)$ for `FBB`
$/$	(a:Eint , b:EInt, c:EInt) (a:EDouble,b:EDouble,c:EDouble) (a:EFloat , b:EFloat, c:EFloat) (a:EShort , b:EShort, c:EShort) (a:ELong , b:ELong, c:ELong)	$(a = b/c)$ for `FBB`
$*$	(a:Eint , b:EInt, c:EInt) (a:EDouble,b:EDouble,c:EDouble) (a:EFloat , b:EFloat, c:EFloat) (a:EShort , b:EShort, c:EShort) (a:ELong , b:ELong, c:ELong)	$(a = b \cdot c)$ for `FBB`

14 Conclusion and next steps

Congratulations – you've reached the end of eMoflon's introduction to unidirectional model transformations! You've learnt that SDMs are declared as *activities*, which consist of *activity nodes*, which are either *story pattern*s or *statement nodes* (for method calls). *Patterns* are made up of *object* and *link variables* with appropriate attribute constraints. Each of these variables can be given different *binding states*, binding operators, and can be marked as negative (for expressing NACs).

To further test your amazing story driven modelling skills, challenge yourself by:

- Adjusting `check` to eject the card from the box if it is guessed correctly and contained in the last partition (to signal it's been learnt). Do you know how `check` currently handles this?
- Editing `Partition.empty()` to include a method call to `removeCard`, thus reusing this previous SDM
- Modifying the GUI source files to execute all methods

If you have any comments, suggestions, or concerns for this part, feel free to drop us a line at contact@moflon.org. Otherwise, if you enjoyed this section, continue to Part IV to learn about Triple Graph Grammars, or Part V for Model-to-Text Transformations. The final part of this handbook – Part VI: Miscellaneous – contains a full glossary, eMoflon hotkeys, and tips and tricks in EA which you might find useful when creating SDMs in the future.

For more detailed information on each part, please refer to Part 0, which can be downloaded at `https://emoflon.github.io/eclipse-plugin/beta/handbook/part0.pdf`.

Cheers!

Glossary

Activity Top-most element of an SDM.

Activity Edge A directed connection between activity nodes describing the control flow within an activity.

Activity Node Represents atomic steps in the control flow of an SDM. Can be either a story node or statement node.

Assignments Used to set attributes of object variables.

Attribute Constraint A non-structural constraint that must be satisfied for a story pattern to match. Can be either an assertion or assignment.

Binding State Can be either *bound* or *unbound/free*. See *Bound vs Unbound*.

Binding Operator Determines whether a variable is to be *checked*, *created*, or *destroyed* during pattern matching.

Binding Semantics Determines if an object variable *must* exist (*mandatory*), may not exist (*negative*; see *NAC*), or is *optional* during *pattern matching*.

Bound vs Unbound Bound variables are completely determined by the current context, whereas unbound (free) variables have to be determined by the *pattern matcher*. `this` and parameter values are always bound.

Dangling Edges An edge with no target or source. Graphs with dangling edges are invalid, which is why dangling edges are avoided and automatically deleted by the pattern matching engine.

EA Enterprise Architect; The UML visual modeling tool used as our visual frontend.

Edge Guards Refine the control flow in an activity by guarding activity edges with a condition that must be satisfied for the activity edge to be taken.

Link Variable Placeholders for links between matched objects.

Literal Expression Represents literals such as true, false, 7, or "foo." See Section 12.

MethodCallExpression Used to invoke any method. See Section 12.

NAC Negative Application Condition; Used to specify structures that must not be present for a rule to be applied.

Object Variable Place holders for actual objects in the current model to be determined during pattern matching.

ObjectVariableExpression Used to reference other object variables. See Section 12.

Parameter Expression Used to refer to method parameters. See Section 12.

(Graph) Pattern Matching Process of assigning objects and links in a model to the object and link variables in a pattern in a type conform manner. This is also referred to as finding a match for the pattern in the given model.

Statement Node Used to invoke methods as part of the control flow in an activity.

Story Node *Activity node* that contains a *story pattern*.

Story Pattern Specifies a structural change of the model.

Unification An extension of the Object Oriented "Everything is an object" principle, where everything is regarded as a *model*, even the metamodel which defines other models.

An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part IV: Triple Graph Grammars

For eMoflon Version 2.16.0

File built on 21st September, 2016

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

The eMoflon team
Darmstadt, Germany (September 2016)

Contents

Part IV:

Learning Box to Dictionary *and* back again with TGGs

URL of this document: https://emoflon.github.io/eclipse-plugin/beta/handbook/part4.pdf

If you're just joining us in this part and are only interested in bidirectional model transformations with *Triple Graph Grammars* then welcome! In fact to keep things simple, we shall assume for the tool-related parts of the handbook that you're starting (over) with a clean slate.

To briefly review what we have done so far in the previous parts of this handbook: we have developed Leitner's learning box by specifying its *abstract syntax* and *static semantics* as a *metamodel*, and finally implementing its *dynamic semantics* via Story Driven Modeling (programmed graph transformations). If the previous sentence could just as well have been in Chinese[1] for you, then consider work through Parts I—III. If you're only in interested in TGGs, however, and you have the appropriate background (or don't care) then that's also fine.

Even though SDMs are crazily cool (don't you forget that!), it is deeply unsatisfactory implementing an inherently *bidirectional* transformation as two unidirectional transformations. If you critically consider the straighforward solution of specifying forward and backward transformations as separate SDMs, you should be able to realise at least the following problems:

Productivity: We have to implement two transformations that are really quite similar, *separately*. This simply doesn't feel productive. Wouldn't it be nice to implement one direction such as the forward transformation, then get the backward transformation for free? How about deriving forward *and* backward transformations from a common joint specification?

[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?

Maintainability: Another maybe even more important point is that two separate transformations often become a pain to maintain and keep *consistent*. If the forward transformation is ever adjusted to produce a different target structure, the backward transformation must be updated appropriately to accommodate the change, and vice-versa. Again, it would be great if the language offered 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 (i.e., automotive or medical systems). With two separate transformations, *traceability* has to be supported manually (and that seriously sucks)!

Incrementality and other cool stuff: Traceability links can also be used to propagate changes made to an existing pair of models *incrementally*, i.e., without recreating the models from scratch. This is not only more efficient in most cases (small change and humongous models), but is also sometimes necessary to avoid losing information in one model that simply cannot be recreated with the other model. Finally, having a declarative (meaning here direction agnostic, i.e., neither forward nor backward) makes it easier to derive all kind of different transformations including a model generator (we'll actually get to use this later).

Our goal in this part is, therefore, to investigate how Triple Graph Grammars (TGGs), a cool *bidirectional* transformation language, can be used to address these problems. To continue with our running example, we shall transform `LeitnersLearningBox`, a partitioned container populated with unsorted cards that are moved through the box as they are memorized,[2] into a `Dictionary`, a single flat container able to hold an unlimited number of entries classified by difficulty (Figure 0.1).

To briefly explain, each card in the box has a keyword on one side that a user can see, paired with a definition hidden on the opposite side. We will combine each of these to create the keyword and content of a single dictionary entry, perhaps assigning a difficulty level based on the card's current position in the box. We also want to be able to transform in the opposite direction, transforming each entry into a card by splitting up the contents, and inserting the new element into a specific partition in the box. After a short introduction to TGGs and setting up your workspace correctly, we shall see how to develop your first bidirectional transformation!

[2] For a detailed review on Leitner's Learning Box, see Part II, Section 1

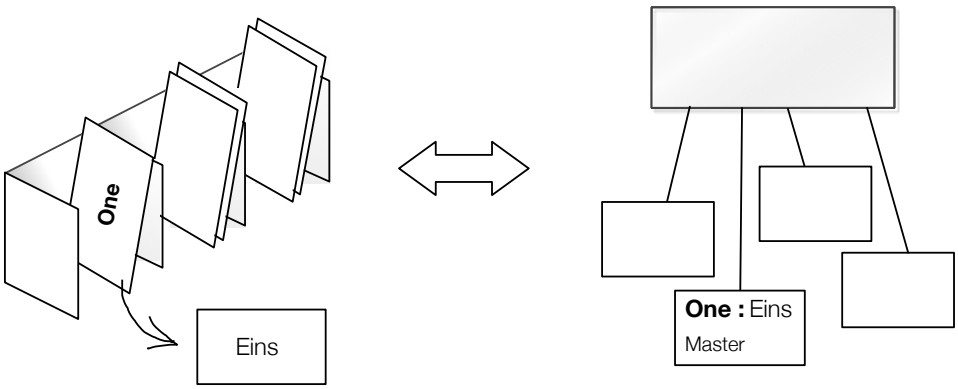


Figure 0.1: Transforming Leitner's learning box into a dictionary

1 Triple Graph Grammars in a nutshell

Triple Graph Grammars [5, 6, 3] are a declarative, rule-based technique used to specify the simultaneous evolution of three connected graphs. Basically, a TGG is just a bunch of rules. Each rule is quite similar to a *story pattern* and describes how a graph structure is to be built-up via a precondition (LHS) and postcondition (RHS). The key difference is that a TGG rule describes how a *graph triple* evolves, where triples consist of a source, correspondence, and target component. This means that executing a sequence of TGG rules will result in source and target graphs connected via nodes in a third (common) correspondence graph.

Graph Triples

Please 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. The important thing to remember is that TGGs are *symmetric* in nature.

So far, so good! Except you may be now be asking yourself the following question: "What on earth does all this have to do with bidirectional model transformation?" There are two main ideas behind TGGs:

(1) A TGG defines a consistency relation: Given a TGG (a set of rules), you can inspect a source graph S and a target graph T, and say if they are *consistent* with respect to the TGG. How? Simply check if a triple $(S \leftarrow C \rightarrow T)$ can be created using the rules of the TGG!

If such a triple can be created, then the graphs are consistent, denoted by: $S \Leftrightarrow_{TGG} T$. This consistency relation can be used to check if a

given bidirectional transformation (i.e., a pair (f, b) of a unidirectional forward transformation f and backward transformation b) is correct. In summary, *a TGG can be viewed as a specification of how the transformations* should *behave ($S \Leftrightarrow_{TGG} f(S)$ and $b(T) \Leftrightarrow_{TGG} T$).*

(2) The consistency relation can be operationalized: This is the surprising (and extremely cool) part of TGGs – correct forward *and* backward transformations (i.e., f and b) can be derived automatically from every TGG [1, 2]!

In other words, the description of the simultaneous evolution of the source, correspondence, and target graphs is *sufficient* to derive a forward and a backward transformation. As these derived rules explicitly state step-by-step how to perform forward and backward transformations, they are called *operational* rules as opposed to the original TGG *declarative* rules specified by the user. This derivation process is therefore also referred to as the *operationalization* of a TGG.

Operationalization

Before getting our hands dirty with a concrete example, here are a few extra points for the interested reader:

- Many more operational rules can be automatically derived from the $S \Leftrightarrow_{TGG} T$ consistency relation including inverse rules to *undo* a step in a forward/backward transformation [4],[3] and rules that check the consistency of an existing graph triple.
- You might be wondering why we need the correspondence graph. The first reason is that the correspondence graph can be viewed as a set of explicit traceability links, which are often nice to have in any transformation. With these you can, e.g., immediately see which elements are related after a forward transformation. There's no guessing, no heuristics, and no interpretation or ambiguity.

  The second reason is more subtle, and difficult to explain without a concrete TGG, but we'll do our best and come back to this at the end. The key idea is that the forward transformation is very often actually *not* injective and cannot be inverted! A function can only be inverted if it is *bijective*, meaning it is both *injective* and *surjective*. So how can we derive the backward transformation?

  eMoflon sort of "cheats" when executing the forward transformation and, if a choice had to be made, remembers which target element was

[3] Note that the TGGs are symmetric and forward/backward can be interchanged freely. As it is cumbersome to always write forward/backward, we shall now simply say forward.

chosen. In this way, eMoflon *bidirectionalizes* the transformation on-the-fly with correspondence links in the correspondence graph. The best part is that if the correspondence graph is somehow lost, there's no reason to worry because the *same* TGG specification that was used to derive your forward transformation can also be used to reconstruct a possible correspondence model between two existing source and target models.[4]

This was a lot of information to absorb all at once, so it might make sense to re-read this section after working through the example. In any case, enough theory! Grab your computer (if you're not hugging it already) and get ready to churn out some wicked TGGs!

[4]We refer to this type of operational rule as *link creation*. This turns out to be harder than it appears and support for link creation in eMoflon is currently still work in progress.

2 Setting up your workspace

To start any TGG transformation, you need to have source and target metamodels. Our example will use the `LeitnersLearningBox` metamodel (as completed in Parts II and III) as the transformation's source, and a new `DictionaryLanguage` metamodel as its target.

Even if you've worked through the previous parts of this handbook, we still recommend that you start with a fresh workspace (and even a fresh Eclipse if possible). If you insist on keeping your stuff from the previous parts then ok—but you're on your own, and anything can go wrong (and the universe might implode, etc).

2.1 Get Eclipse

Navigate to `https://www.eclipse.org/downloads/` and download the latest `Eclipse Modeling Tools`.[5] Do not try to use another Eclipse package as it probably won't work.

2.2 Install eMoflon

Install eMoflon as an Eclipse plugin from this update site.[6] When installing, make sure you choose to install *all* available features.

If you don't already have Graphviz dot installed on your system then install it: `http://www.graphviz.org/Download.php`

2.3 Install your initial workspace

To get started in Eclipse, press the "Install, configure and deploy Moflon" button and navigate to "Install Workspace". Choose "Handbook Example (Part 4)" (Figure 2.1). It contains the `LeitnersLearningBox` and `DictionaryLanguage` metamodels, which we'll be using. By the way, don't freak out if your workspace looks slightly different than our screenshots – things often change faster than we can update this handbook. If the difference is important and confusing, however, please send us an email: `contact@moflon.org`

[5] We have tested this part of the handbook for `Eclipse Mars.2 (4.5.2)` running on Mac OS X Yosemite and Windows 8.

[6] `http://www.emoflon.org/fileadmin/download/moflon-ide/eclipse-plugin/beta/update-site2/`

If everything went well, you should now have two projects in your workspace. Go ahead and explore the metamodels (`/model/*.ecore`, `/model/*.aird`). The generated code (`/gen`) is standard EMF code.

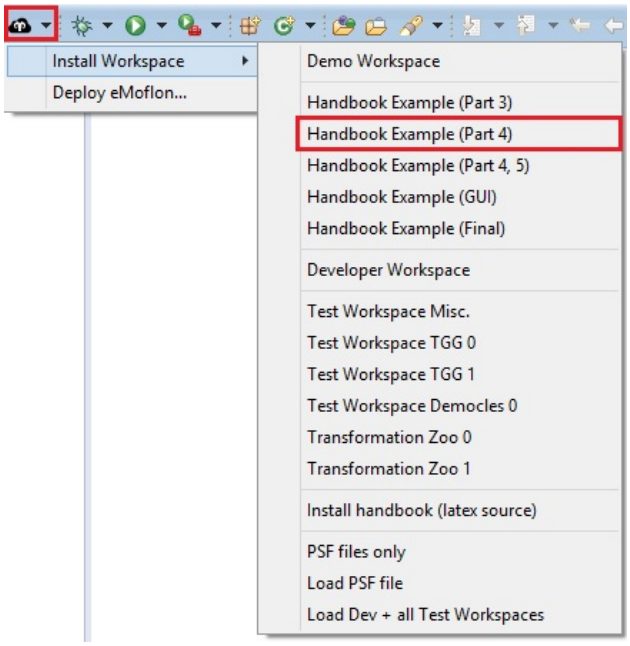


Figure 2.1: Initialize your workspace

If you ever want to create your own "repository" projects, eMoflon provides a wizard ("Create new repository project" in the eMoflon toolbar) for this, which creates the expected project structure with an empty ecore file that you can fill out.

3 Creating your TGG schema

Now that we have our source and target metamodels, we're going to start modeling the correspondence component of our envisioned triple language. This correspondence, or *link metamodel*, specifies *correspondence types*, which will be used to connect specific elements of the source and target metamodels. These correspondence elements can also be thought of as *traceability links*. While the link metamodel is technically just a good old metamodel like any other, eMoflon provides a special wizard and project infrastructure (nature, builder) for so called "integration" projects.

Link Metamodel
Correspondence Types

The overall metamodel triple consisting of the relevant parts of the source, link, and target metamodels is called a *TGG schema*. A TGG schema can be viewed as the (metamodel) triple to which all *new* triples must conform. In less technical lingo, it gives an abstract view on the relationships (correspondence) between two metamodels or domains. A domain expert should be able to understand why certain connected elements are related, irrespective of how the relationship is actually established by TGG rules, just by looking at the TGG schema.

TGG Schema

In our example schema, we will start by creating a link between our source `Box` and target `Dictionary` to express that these two container elements are related.

- Choose the "Create new integration project" wizard from the eMoflon task bar, enter the project name as `LearningBoxToDictionaryIntegration`, and click `Finish` (Figure 3.1).
- Open the schema file `src/org/moflon/tgg/mosl/Schema.tgg`, and import both repository projects by adding import statements as depicted in Figure 3.2. As both projects are eMoflon projects (they don't *have* to be but it makes things simpler) you can use a nifty little template provided by the editor for "eMoflon imports".
- Now state which project is to be source (please choose `LearningBoxLanguage`) and target (choose `DictionaryLanguage`) by stating the root packages in the `#source` and `#target` scopes of the schema (Figure 3.2). As with (almost) everything, the editor will try to help you with a reasonable selection if you hit "Ctrl + Space".
- Finally, create a first correspondence type named `BoxToDictionary` connecting `Boxes` and `Dictionaries` as indicated in Figure 3.2.

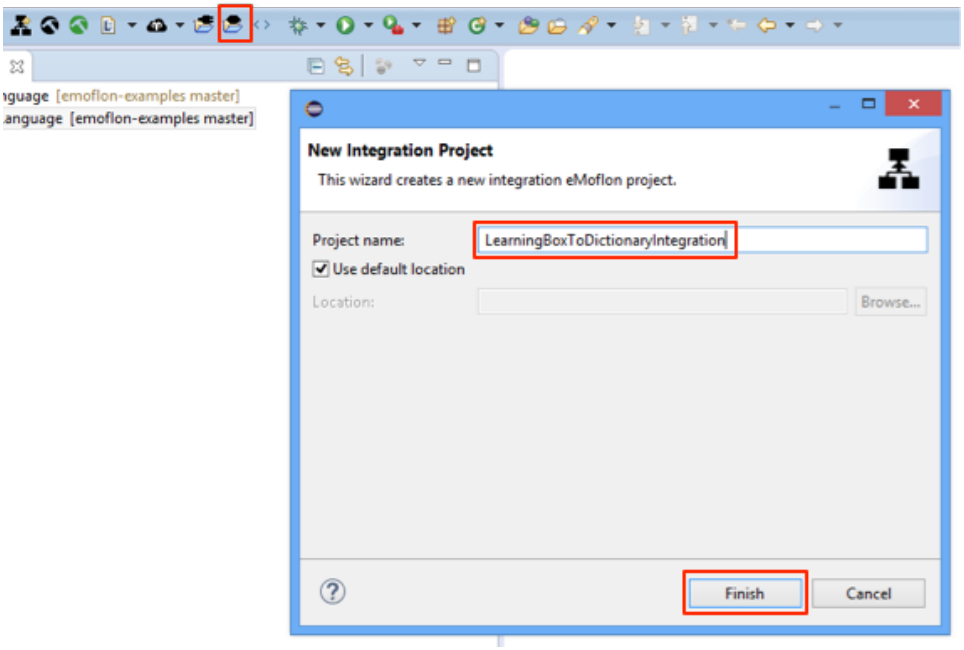


Figure 3.1: Create a new TGG integration project

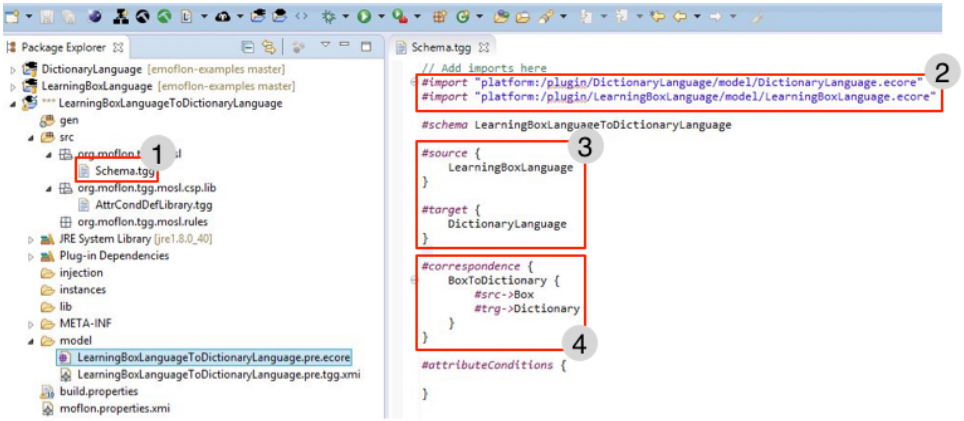


Figure 3.2: Add a first correspondence type

4 Specifying TGG rules

With our correspondence type defined in the TGG schema, we can now specify a set of *TGG rules* to describe a language of graph triples.

As discussed in Section 1, a TGG rule is quite similar to a story pattern, following a *precondition, postcondition* format. This means we'll need to state:

- What must be matched (i.e., under which conditions can a rule be applied; 'black' elements)
- What is to be created when the rule is applied (i.e., which objects and links must exist upon exit; 'green' elements)

Note that the rules of a TGG only describe the simultaneous *build-up* of source, correspondence, and target models. Unlike story diagrams, they do not delete or modify any existing elements. In other words, TGG rules are *monotonic*. This might seem surprising at first, and you might even think this is a terrible restriction. The intention is that a TGG should only specify a consistency relation, and *not* the forward and backward transformations directly, which are derived automatically. In the end, modifications are not necessary on this level but can, of course, be induced in certain operationalizations of the TGG.

Monotonic

Let's quickly think about what rules we need in order to successfully transform a learning box into a dictionary. We need to first take care of the `box` and `dictionary` structures, where `box` will need at least one `partition` to manipulate its `card`s. If more than one is created, those partitions will need to have appropriate `next` and `previous` links. Conversely, given that a `dictionary` is unsorted, there are no counterparts for partitions. A second rule will be needed to transform `card`s into `entries`. More precisely, a one-to-one correspondence must be established (i.e., one `card` implies one `entry`), with suitable concatenation or splitting of the contents (based on the transformation direction), and some mechanism to assign difficulty levels to each `entry` or initial position of each `card`.

4.1 BoxToDictionaryRule

- Select the created subpackage `src/org.moflon.tgg.mosl.rules` (Step 1 in Figure 4.1) and click on the wizard "Create new TGG rule" (Step 2). In the dialogue that pops up, enter `BoxToDictionaryRule` as the name of the TGG rule to be created. Open the newly created file.

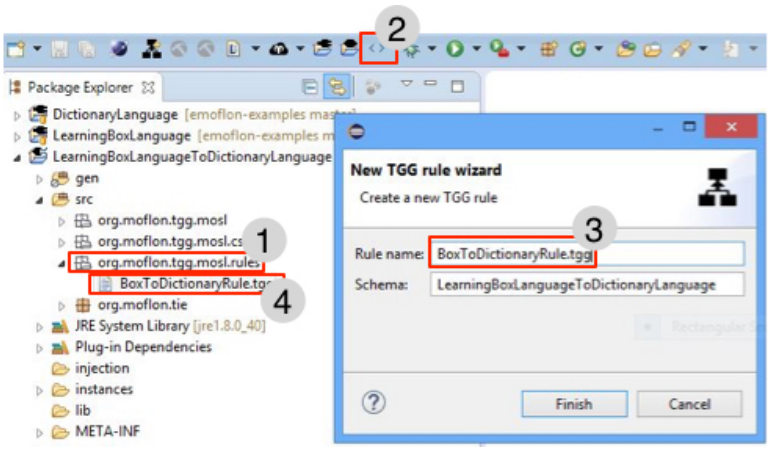


Figure 4.1: Creating a TGG rule

- Ensure that you open the `PlantUML` view (Step 1 in Figure 4.2) as it provides a helpful visualisation of the current TGG rule opened in the editor. To prevent slowing you down, the view is only updated when you save *and* change your cursor position. Elements in the source domain have a yellow background, while target elements have a peach background. Basic UML object diagram syntax is used, extended with colours to represent created elements (green outline) and context elements (black outline). `BoxToDictionaryRule` is a so called"island" rule and only has created elements. For presentation purposes, correspondence links are abstracted from in the visualisation and are depicted as bands (e.g., between `box` and `dictionary`). The rule we want to specify creates a minimal dictionary (just a `Dictionary` node), and a minimal box, which is a bit more interesting as it consists already of three correctly connected partitions. Any structure less than this is not a valid learning box.

- The textual syntax we use is pretty straightforward: every domain has a scope (`#source`, `#target`, and `#correspondence`), and there is a final scope for so called attribute conditions, which express how attribute values relate to each other. The domain scopes contain again scopes for each *object variable* in the rule (e.g., `box`) and these scopes

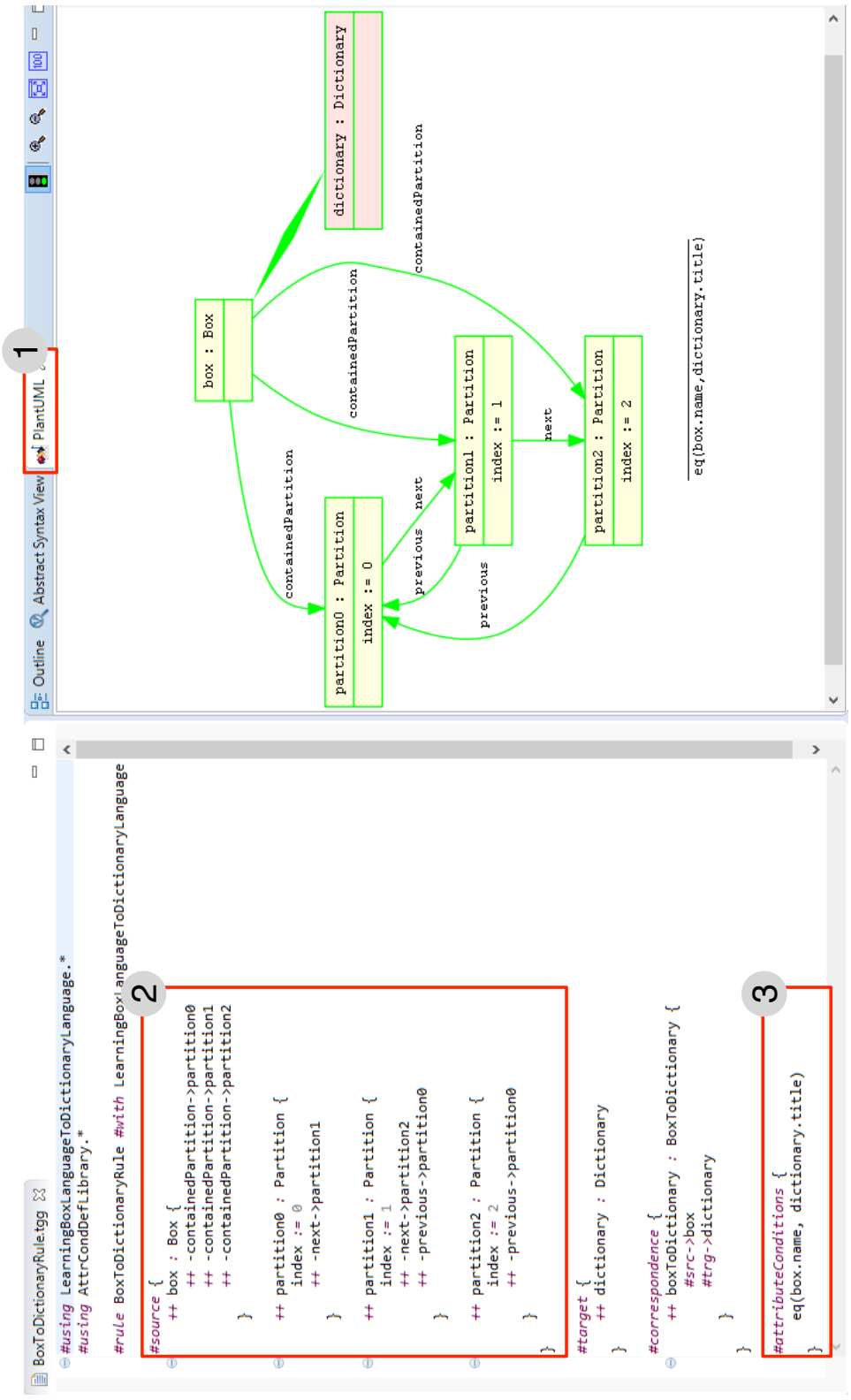


Figure 4.2: Complete TGG rule diagram for `BoxToDictionaryRule`

then contain outgoing *link variables* (e.g., `-containedPartition->`), as well as inline attribute conditions (e.g., `index := 0`). Go ahead and specify the source scope as depicted in Step 2 of Figure 4.2. Notice how the `++` operator is used to mark object and link variables as created or context variables. Try removing and adding it and see how the visualisation changes. As always, let the editor help you by pressing "Ctrl+Space" as often as possible.

- Specify the correspondence and target scopes accordingly and make sure your rule (and its visualisation) closely resembles Figure 4.2.
- To complete our first rule, specify an attribute condition to declare that the name of the box and the title of the dictionary are always to be equal. As depicted in Step 3 Figure 4.2, this can be accomplished using an `eq` condition. We provide a standard library of such attribute conditions (press `F3` on the constraint to jump to the library file), but it is also possible to extend this library with your own attribute conditions. We'll see how to do this in a moment.

Fantastic work! The first rule of our transformation is complete! If you are in hurry, you could jump ahead and proceed directly to Section 5: TGGs in Action. There you can transform a box to a dictionary and vice-versa, but please be aware that your specified TGG (with just one rule) will only be able to cope with completely empty boxes and dictionaries. Handling additional elements (i.e., cards in the learning box and entries in the dictionary) requires a second rule. We intend to specify this next.

4.2 CardToEntryRule

Our next goal is to be able to handle `Card` and `Entry` elements. The new thing here is that it will require a pre-condition – you should not be able to transform these child elements (cards and entries) unless certain structural conditions (there parents exist and are related) are met. In other words, we need a rule that demands an already existing `box` and `dictionary`. It will need to combine 'black' and 'green' variables!

- As we'll be connecting cards and entries, we need a new correspondence type. Open the TGG schema and add a new correspondence type `CardToEntry` connecting a `Card` and an `Entry`.
- Create a new TGG rule with `CardToEntryRule` its name, and specify the rule as depicted in Figure 4.3.

Your diagram should now resemble Figure 4.3. We're not done yet though, we still need to handle attributes! To understand why, consider the current rule and ask yourself *which* partition is meant. Exactly! This is currently not specified, so *any* partition can be taken when applying the rule. eMoflon would actually collect all applicable rules (one for every partition) and consult a configuration component to decide which rules should be taken. The default component just chooses randomly, but you could override this and e.g., pop up a dialogue asking the user which partition is preferred. Although this could be an interesting solution, we'll see how to fully specify things using extra attribute conditions.

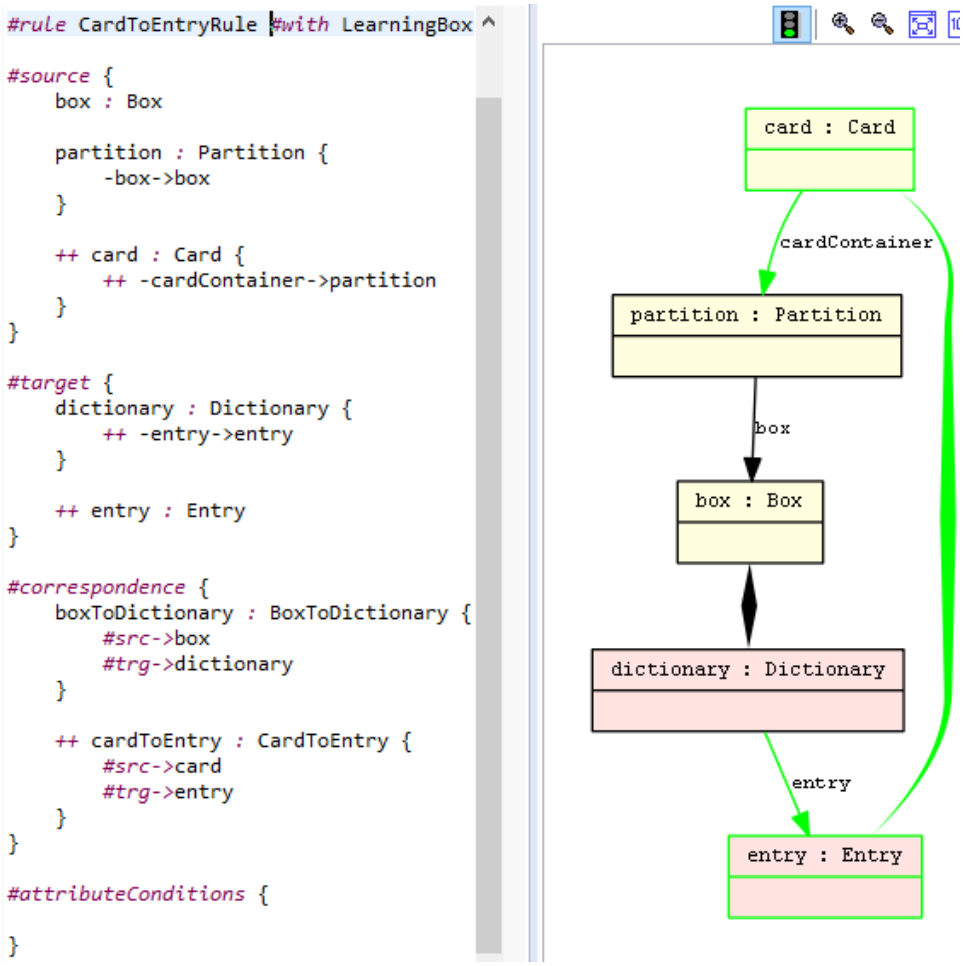


Figure 4.3: `CardToEntryRule` without attribute manipulation

On the way to handling all attributes, let's first start with the relatively easy case of specifying how every `entry.content`, `card.back`, and `card.face` relate to each other. We should probably combine the front and back of

each card as a single content attribute of an entry and, in the opposite direction, split the content into `card.back` and `card.face`.

So let's define `entry.content` as: `<word>:<meaning>`. `card.back` should therefore be `Question:<word>` and similarly, `card.face` should be `Answer:<meaning>`.

- Luckily, we have two library attribute conditions, `addPrefix` and `concat` to help us with this. Add the following to your rule:

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

  "Question " and "Answer " are EString literals, `word` and `meaning` are local variables, and `card.face`, `card.back`, and `entry.content` are attribute expressions (this should be familiar from SDM story patterns).

- Our final task is now to specify where a new `card` (when transformed from an `entry`) should be placed. We purposefully created three partitions to match the three difficulty levels, but if you check the available library constraints, there is nothing that can directly implement this specific kind of mapping. We will therefore need to create our own attribute condition to handle this.

- Open the TGG schema, and define a new attribute condition in the `#attributeConditions` scope. Name it `IndexToLevel`, and enter the values given in Figure 4.4.

  ```
  #attributeConditions {

      #userDefined
      indexToLevel(0:EInt, 1:EString){
          #sync: BB, BF
          #gen:
      }

  }
  ```

  Figure 4.4: Specifying a new attribute condition

- Please note that this is just a *specification* of a custom attribute condition – we still need to actually implement it in Java! As we're so close to finishing this TGG rule, however, let's complete it first before we work out the exact meaning of the mysterious adornments (those

funny `BB`, `BF`, ...) and parameters (`0:EInt` and `1:EString`) of the attribute condition. For now, just make sure you enter the exact values in Figure 4.4.

- We can now use this new attribute condition just like any of the library attribute conditions. Add the following to `CardToEntryRule` to express that the relationship between the index of the partition containing the new card, and the level of the new entry, is defined by our new attribute condition:

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

If everything has been done correctly up to this point, your project should save and build (hit the hammer symbol in the eMolfon task bar). The generated code will have some compilation errors (Step 1 in Fig. 4.5) as Eclipse does not know where to access the generated code for the imported source and target ecore files (these could also be supplied from jars or installed plugins). In our case the generated code is in the respective source and target projects so let's communicate this to Eclipse.

- Open the `MANIFEST.MF` file (Step 2 in Fig. 4.5) and choose the `Dependencies` tab (Step 3).
- Choose both source and target projects (Step 4) as dependencies and click `OK`. All compilations errors should now be resolved.

Great work! All that's left to do is implement the `indexToLevel` constraint, and give your transformation a test run.

4.3 Implementing IndexToLevel

Our TGG still isn't yet complete. While we've declared and actually used our custom `indexTolevel` attribute condition, we haven't actually implemented it yet. Let's quickly review the purpose of attribute conditions.

Just like patterns describing *structural* correspondence, *attribute conditions* can be automatically *operationalized* as required, e.g., for a forward transformations. Even more interesting, a set of attribute conditions might have to be ordered in a specific way depending on the direction of the transformation. Enforcing the conditions might involve checking existing attribute values, or setting these values appropriately.

For built-in *library* attribute conditions such as *eq*, *addPrefix* and *concat*, you do not need to worry about these details and can just focus on expressing what should hold. Everything else is handled automatically.

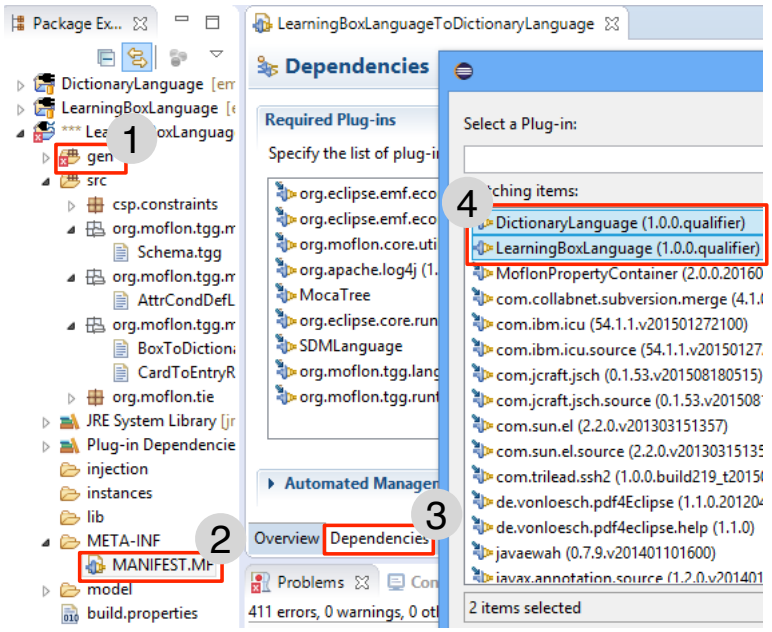


Figure 4.5: Add source and target projects as dependencies

In some cases however, a required attribute condition might be problem-specific, such as our *indexToLevel*. There might not be any fitting combination of library attribute conditions to express the consistency condition, so a new attribute condition type must be declared and implemented.

There is a list of *adornments* in the declaration which specify the cases for which the attribute condition can be operationalized. Each adornment consists of a `B` (bound) or `F` (free) variable setting for each argument of the attribute condition. This might sound a bit complex, but it's really quite simple, especially in the context of our example:

BB indicates that the `partition.index` and `entry.level` are both *bound*, i.e., they already have assigned values. In this case, the *operation* must check if the assigned values are valid and correct.

BF indicates 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 would indicate 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`.

FF would indicate that both `partition.index` and `entry.level` are *free* and we have to somehow generate consistent values out of thin air.

As `partition` is a context element in the rule (the partition is always bound in whatever direction), **FF** and **FB** are irrelevant cases and we do not need to declare or implement what they mean. For the record, note that adornments can be declared as either `#gen` or `#sync`. The reason is that it might make sense to restrict some adornments (typically **FF** cases) to only when generating models. Using **FF** cases for synchronisation might possibly makes sense, but most of the time it would be weird to generate random values during a forward or backward synchronisation.

At compile time, the set of attribute conditions for every TGG rule is "solved" for each case by operationalizing all constraints and determining a feasible sequence in which the operations can be executed, compatible to the declared adornments of each attribute condition. If the set of attribute conditions cannot be solved, an exception is thrown at compile time.

Now that we have a better understanding behind the construction of attribute conditions, let's implement `indexToLevel`.

- Locate and open `IndexToLevel.java` under "src/csp.constraints" in `LearningBoxToDictionaryIntegration`.
- As you can see, some code has been generated in order to handle the current unimplemented state of `IndexToLevel`. Use the code depicted in Figure 4.6 to replace this default implementation.[7]

To briefly explain, the `levels` list contains difficulty level at positions 0, 1, or 2 in the list, which correspond to our three partitions in the learning box. You'll notice that instead of setting "master" to 2, it has rather been set to match the first 0th partition. Unlike an `entry` in `dictionary`, the position of each `card` in `box` is *not* based on difficulty, but simply how it has been moved as a result of the user's correct and incorrect guesses. Easy cards are more likely to be in the final partition (due to moving through the box quickly) while challenging cards are most likely to have been returned to (and currently to be at) the starting position, i.e., the 0th partition.

In the `solve` method, the index of the matched partition in the rule is first of all normalised (negative values do not make sense, and we handle all partitions after partition 2 in the same way). A switch statement is then used, based on whichever adornment is currently the case, to enforce or check the condition.

[7] Depending of course on your pdf viewer, copy and pasting this code should work.

```
package csp.constraints;

import java.util.Arrays;
import java.util.List;
import org.moflon.tgg.language.csp.Variable;
import org.moflon.tgg.language.csp.impl.TGGConstraintImpl;

public class IndexToLevel extends TGGConstraintImpl {

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

    public void solve(Variable var_0, Variable var_1) {
        int index = ((Integer) var_0.getValue()).intValue();
        int normalisedIndex = Math.min(Math.max(0, index), 2);
        String bindingStates = getBindingStates(var_0, var_1);

        switch (bindingStates) {
        case "BB":
            String level = (String) var_1.getValue();
            setSatisfied(levels.get(normalisedIndex).equals(level));
            break;
        case "BF":
            var_1.bindToValue(levels.get(normalisedIndex));
            setSatisfied(true);
            break;
        }}}
```

Figure 4.6: Implementation of our custom `IndexToLevel` constraint

For `BB` we check if the normalised index of the partition corresponds to the difficulty level of the card. For `BF`, the normalised index is used to set the appropriate difficulty level of the card.

5 TGGs in action

Before we can execute our rules, we need to create something for the TGG to transform. In other words, we need to create an instance model[8] of either our target or our source metamodel! Since dictionaries are of a much simpler structure, let's start with the backwards transformation.

- Navigate to `DictionaryLanguage/model/` and open `DictionaryLanguage.ecore`. Expand the tree and create a new dynamic instance of a `Dictionary` named `bwd.src.xmi` (choose the EClass `Dictionary`, right-click, and select `Create dynamic instance`). Make sure you persist your instance as `LearningBoxToDictionaryIntegration/instances/bwd.src.xmi` (as depicted in Figure 5.1).

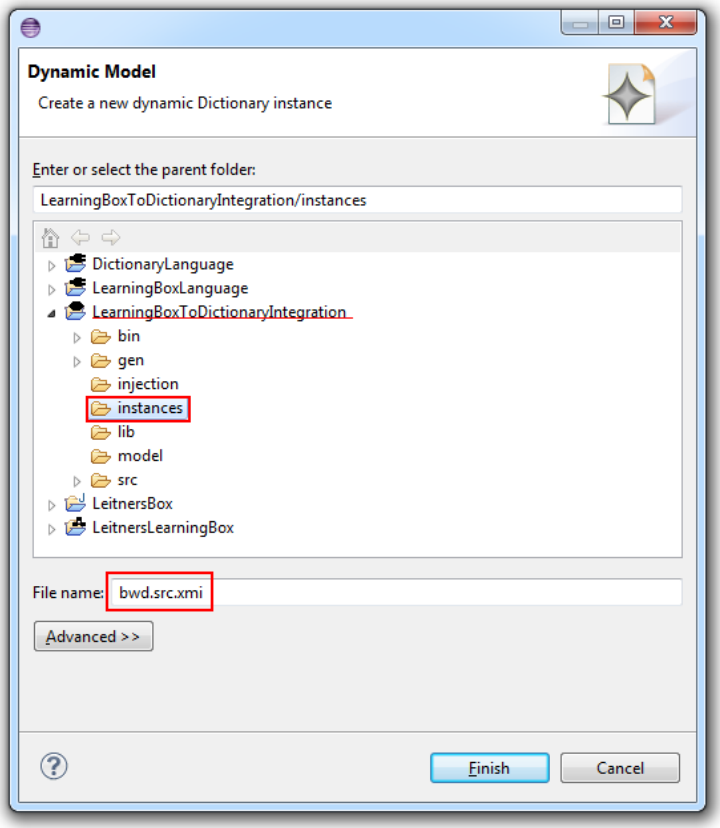


Figure 5.1: Create a dynamic instance of `Dictionary`

[8] For a detailed review how to create instances, refer to Part II, Section 3

- Open the new file and edit the `Dictionary` properties by double-clicking and setting `Title` to `English Numbers` in the `Properties` tab below the window.

- Create three child `Entry` objects. Don't forget the syntax we decided upon for each `entry.content` in the `CardToEntryRule` when setting up the constraints! Be sure to set this property according to the format `<word>:<meaning>`. Give each `entry` a different difficulty `level`, e.g., `beginner` for `One:Eins`, `advanced` for `Two:Zwei`, and `master` for `Three:Drei`. Your instance should resemble Figure 5.2. After this works you should of course play around with the model and add all kind of cards to see what happens.

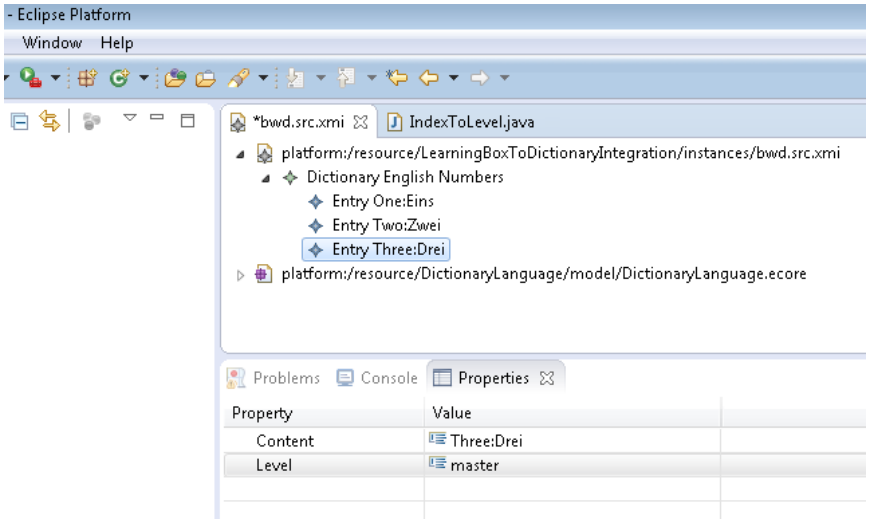


Figure 5.2: Fill a `Dictionary` for the transformation

- eMoflon generates default stubs to execute the transformation. Navigate to "LearningBoxToDictionaryIntegration/src/org.moflon.tie" and open `LearningBoxToDictionaryIntegrationTrafo.java`.

- As you can see, this file is a driver for forward and backward transformations, transforming a source `box` to a target `dictionary`, and backward from a `dictionary` to a `box`. As this is good old Java code, you can adjust everything as you wish. To follow this handbook, however, do not change anything for the moment.

- Right-click the file in the Package Explorer and navigate to "Run as. . . /Java Application" to execute the file.

- Did you get one error message, followed by one success message in the eMoflon console window (Figure 5.3) below the editor? Perfect! Both of these statements make sense – our TGG first attempted the forward transformation but, given that it was missing the source (`box`) instance, it was only able to perform a transformation in the backwards direction. The stub checks for `fwd.src.xmi` and `bwd.src.xmi` files per convention (all this can of course be edited and changed).

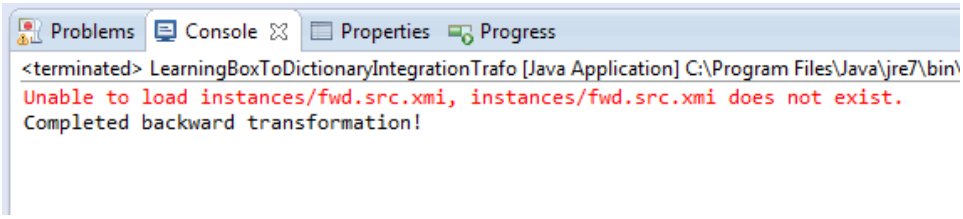


Figure 5.3: Running the backward transformation

- Refresh the integration project's `instances` folder. Three new `.xmi` files should have appeared representing your backward triple. While you created the `bwd.src.xmi` instance, the TGG generated `bwd.corr.xmi`, the correspondence graph between target and source, `bwd.protocol.xmi`, a directed acyclic graph of the rule applications that lead to this result, and `bwd.trg.xmi`, the output of the transformation.

- If you open and inspect `bwd.trg.xmi` you'll find that it's a `Box` of `English Numbers`. Expand the tree and you'll see our `Dictionary` in its corresponding `Box` format containing three `Partitions` (Figure 5.4). Double click each `card` and observe how each `entry.content` was successfully split into two sides. Also note how the cards were placed in the correct (at least according to our `indexToLevel` attribute condition) partition.

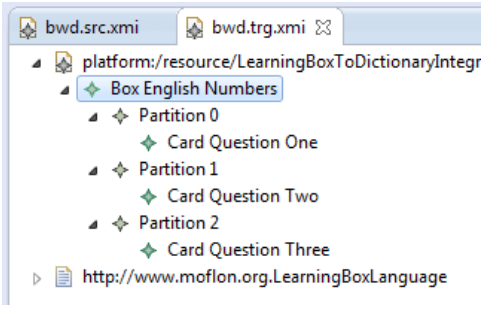


Figure 5.4: Result of the *backwards* transformation

- Congratulations! You have successfully performed your first *backward* transformation using TGGs! To understand better what happened, eMoflon provides two important tools: the first is the correspondence model containing all established links between source and target elements, the second is the so called protocol of the transformation.

- Go ahead and open `bwd.corr.xmi`. This model contains the created correspondence model, and references both source and target models. Using eMoflon's `Abstract Syntax View`, you can drag in a selection of elements and explore (right-click on a node in the view to choose if direct neighbours or all transitive neighbours should be added to the view) how the models are connected. You can remove elements from the view, zoom in and out, and also pick from a choice of standard layout algorithms. Figure 5.5 shows all three complete models in the view with all correspondence links highlighted. You can choose elements in the view to see their attribute values either as a tool tip, or in the standard properties view in Eclipse.

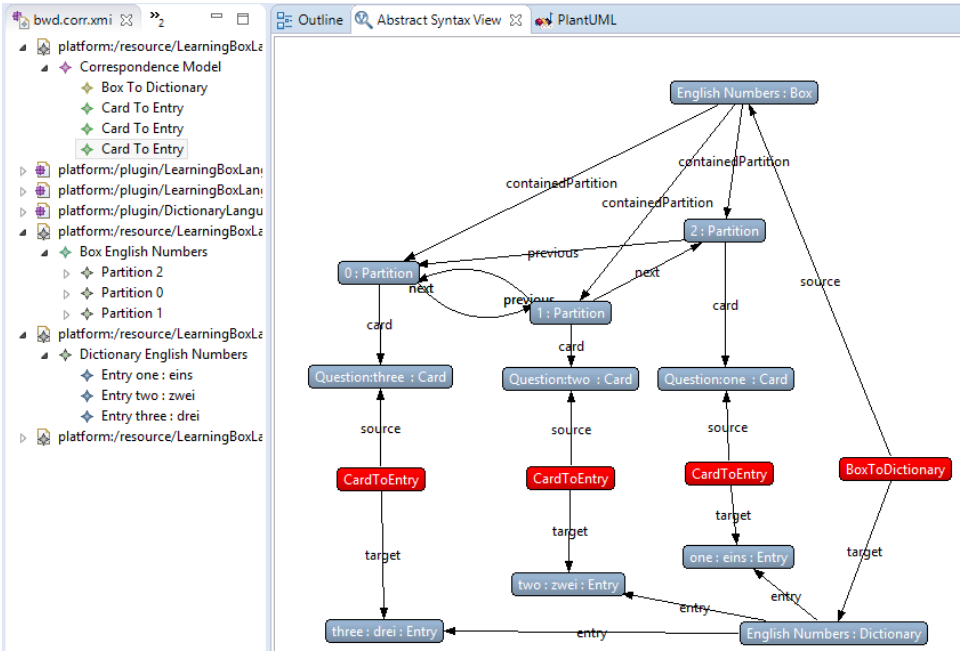


Figure 5.5: Inspecting the correspondence models

- eMoflon provides a special visualisation for protocol models, so go ahead and open `bwd.protocol.xmi`. This time around, use the PlantUML view to inspect the protocol. If you select the root element (`Precedence Structure`), you'll see a tree of *rule applications*, showing you which rules were applied to achieve the result. As `CardTo-`

`EntryRule` can only be applied after `BoxToDictionaryRule`, an application of the latter is a parent for all other rule applications. These are all children on the same level, as the exact order of application is irrelevant in this case. This, in general, directed acyclic graph of rule applications is depicted in Figure 5.6.

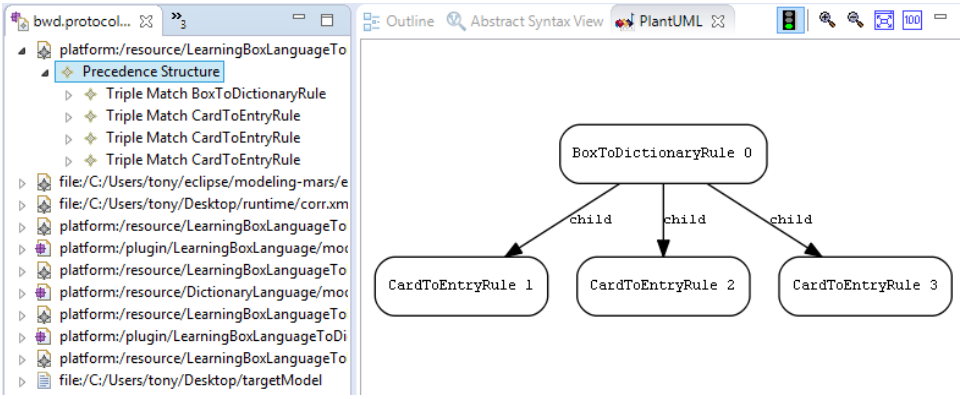


Figure 5.6: Inspecting the protocol

- To inspect a rule application, choose one of the children of the precedence structure in the tree view. If you choose, for example, a `CardToEntryRule` node, a diagram similar to that depicted in Figure 5.7 will be shown as a visualisation of the rule application. To understand this diagram, note that the labels of the nodes are of the form `rule variable ==> model element`, showing how the object and link variables in the rule were "matched" to actual elements in the source and target models. The colours indicate which elements would be created by applying this rule (in the simultaneous evolution of all three models). This protocol is not only for documentation, but is also used for model synchronisation, which we'll discuss and try out in a moment.

- To convince yourself that the transformation is actually bidirectional, create a source model, and run the TGG again to perform a *forwards* transformation of a `Box` into a `Dictionary`. The easiest way to do this is to make a copy of `bwd.trg.xmi` and rename it to `fwd.src.xmi`.

- Run `LearningBoxToDictionaryIntegrationTrafo` again and refresh the "instances" folder. Compare the output `fwd.trg.xmi` against the original `bwd.src.xmi` Dictionary model. If everything went right, they should be isomorphic (identical up to perhaps sorting of children).

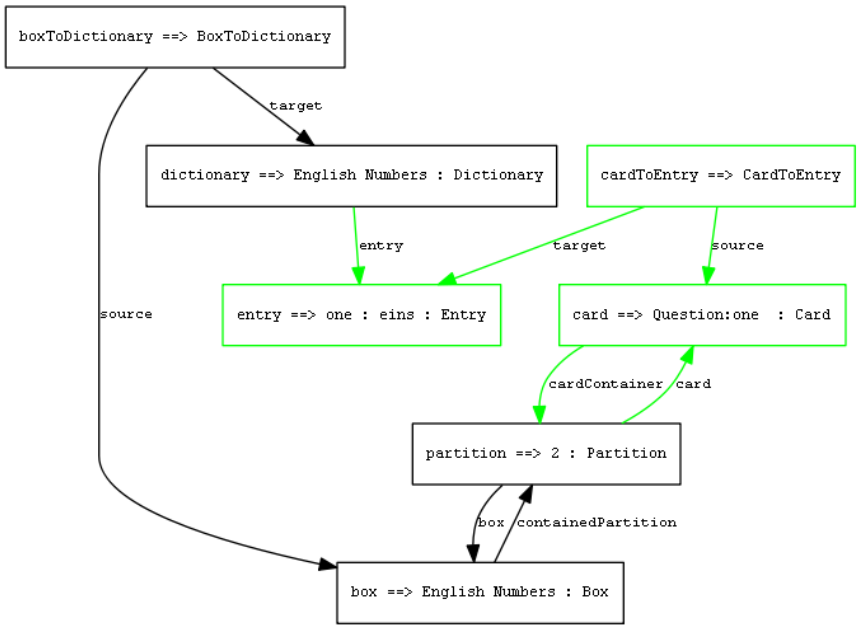


Figure 5.7: Inspecting a rule application

6 Extending your transformation

At this point, we now have a working TGG to transform a `Dictionary` into a `Box` with three `partitions`, and a `Box` with exactly three `Partitions` into a `Dictionary`. The only potential problem is that a learning box with only three partitions may not be the most useful studying tool. After all, the more partitions you have, the more practice you'll have with the cards by being quizzed again and again.

A simple strategy would be to allow additional partitions in the box, but to basically ignore them (treat them all as partitions with index greater than two) when transforming to the dictionary. To accomplish this we need an extra rule that clearly states how such partitions should be ignored, i.e., be translated without affecting the dictionary. We could trivially extend the existing `BoxToDictionaryRule` by connecting a fourth partition, but what if we wanted a fifth one? A sixth? As you can see, this obviously won't work – there will always be the potential for a `n+1`th partition in an `n`-sized box.

With a so-called *ignore rule*, we'll handle some source elements and their connecting link variables without creating any new elements in the target domain. Before specifying this ignore rule, however, let's extend the current

ignore rule

`fwd.src.xmi`[9] by adding a new `Partition` (with `index` = 3) as depicted in Figure 6.1. Connect the new partition `partition3` to `partition0` via a `previous` reference, and connect `partition2` to `partition3` via a `next` reference (this is how to extend a learning box). Create a new `card` in your new `partition3` as well.

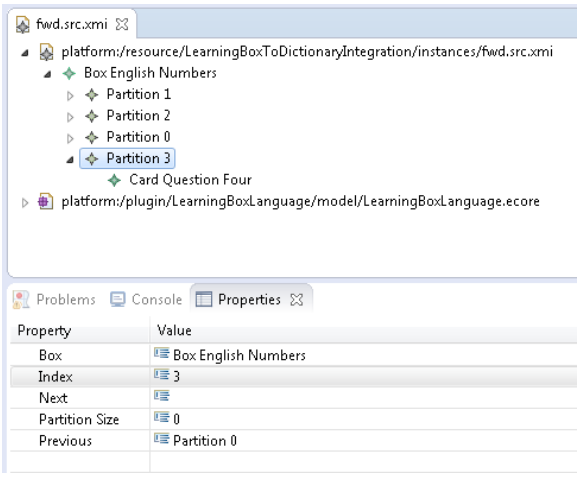


Figure 6.1: Extended `fwd.src.xmi`

If you run your transformation again, you'll get an error message in the console for the forward direction as our `fwd.src.xmi` with four `partitions` cannot be handled with our current TGG. As this might happen quite often when working on a TGG, let's take some time to understand the message and "debug" what went wrong. The first message in the console is:

> *Your TGG LearningBoxLanguageToDictionaryLanguage is not translation locally complete!*

There are basically two things that can go wrong when transforming a source model to a target model with a TGG-based forward transformation: (1) the input model cannot be completely marked (or "parsed", or recognized, etc.) using TGG rules, and (2) some thing went wrong during the translation of the input model, i.e., when trying to extend the output model. The first problem is referred to as *input local completeness*, while the second problem is referred to as *translation local completeness*. The choice of "input" vs. "translation" should be clear from the above explanation. The word "local" indicates that the transformation is always with respect to a certain match, i.e., a restricted fragment of the models. If a TGG is "complete" then these

[9] Remember that you should have created this by copying and renaming `bwd.trg.xmi`.

two problems cannot occur. This is a nice property to have and getting the compiler to check for this statically is ongoing work.

What makes things a bit ugly is that our algorithm does not backtrack while searching for a valid rule application sequence as this would make things extremely (exponentially) slow. The price for this is that we cannot easily differentiate between the case where a TGG rule is simply missing, and the case where a TGG is somehow nasty (requires backtracking) and our algorithm has gotten confused and hit a dead end. This is why the next message in the console is:

> *[...] I was unable to translate [...] without backtracking.*

and not (or slightly more polite isomorphisms thereof):

> *You jerk, I don't have a TGG rule to translate this element!!!*

as you might expect for our concrete example.

After asking you nicely to take a good look at your TGG, the synchroniser tells you exactly where things went wrong:

> *I got stuck while trying to extend the following source matches: [CardToEntryRule]*

That's a bit surprising right? You probably expected everything to work (as it did before), up to the point where the extra partition turns up and no rule can be found for it. Well things aren't that straightforward. To get more information, the synchroniser suggests:

> *Set verbose to true in your synchronization helper and re-run to get the exact list of ignored elements.*

so let's do that and see what happens.

- Open `src/org/moflon/tie/LearningBoxLanguageToDictionaryLanguageTrafo.java` and extend the code invoking the forward transformation as follows:

```
// Forward Transformation
LearningBoxLan... helper = new LearningBoxLan...Trafo();
helper.setVerbose(true); // <--  add this!
helper.performForward("instances/fwd.src.xmi");
```

If you now rerun the transformation, you'll get a printout of exactly which elements could not be translated at all. Curiously, this lists include all partitions *and* the box (`[English Numbers]`). Although we still do not know why the box and the first three partitions could not be translated using `BoxToDicationaryRule`, at least we can now understand why the translation failed at `CardToEntryRule`. The following happened:

1. No match could be found for the box and any partition. Instead of complaining directly, the algorithm assumes that we do not care about these elements (this is very often the case in practice), so it ignores them and tries to continue with the translation.

2. When translating a card, however, although things look ok on the source side, there is no `Dictionary` on the target side and the translation fails. If we would not add the created `Entry` to the `Dictionary`, this rule application would have worked!

OK – let's now understand why the box was already ignored.

- Following the four steps in Figure 6.2, open the generated ecore file in the integration project (Step 1). This file contains not only the correspondence metamodel, but also all operationalized rules. Java code for the transformation is generated directly from this file.

- Under `Rules/BoxToDictionaryRule/` locate the so called "isAppriopriate_FWD..." method for the TGG rule. For every TGG rule, three main types of operational rules are derived in each direction: "isAppropriate" methods to check if a match for a rule can be found, "isApplicable" to check if this match can be extended to cover all domain, and "perform" methods to actually apply the rule in the forward or backward direction. These methods are all generated as unidirectional programmed graph transformations (story diagrams). If you want to see how the generated story diagram looks like, go ahead and select the `Activity` element under the operation (you should get a visualisation as a simple activity diagram).

- The most important story node in the story diagram (Step 2) is `test core match and DEC` (all others are more or less bookkeeping and technical stuff). DEC stands for "Dangling Edge Condition" and represents a lookahead for the algorithm. The basic idea is to check for edges that would be impossible to translate if this rule is applied at this location. This can be checked for statically and the result of this DEC analyis is embedded in the transformation as simple Negative

Application Conditions (NACs). Go ahead and select the story pattern (Step 3). Note that it was derived directly from the TGG rule and, in this sense, *is still a* TGG rule (at least according to the metamodel). You should see an object diagram representing the rule. Note that black means context, while blue means negative (it should not be possible to extend a match to cover any of these elements).

- In our case, there are quite a few edges that would be left (in this sense) dangling. An example is shown as Step 4 in Figure 6.2: if the box has any other contained partition apart from the 3 matched in this rule, then it is clear that this extra edge cannot be translated with any other rule in the current TGG. It would thus be dangling and therefore blocks the application of this rule. Another example would be a next edge going out from `partition2`. Can you locate the NAC for this edge?

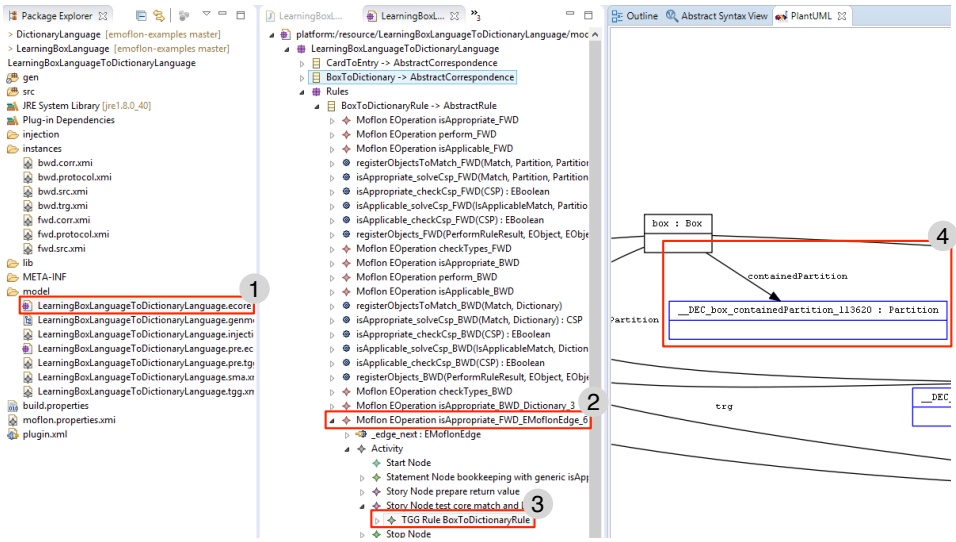


Figure 6.2: Understanding the Dangling Edge Condition

DEC is a great help when it comes to avoiding dead ends without using backtracking, but in our case where the TGG is actually missing a rule, it only postpones the problem and makes it a bit challenging to understand what went wrong. On the bright side we took the chance to dig in a bit right? If you ever have problems understanding why a certain match was not collected, feel free to debug the generated Java code directly if looking at the visualisation does not help (as we did here). Just place breakpoints as usual and run the transformation in debug modus. The generated code is quite readable (at least after a week of practice – haha!).

6.1 AllOtherPartitionsRule

- Create a new rule `AllOtherPartitionsRule`, and complete it according to Figure 6.3.

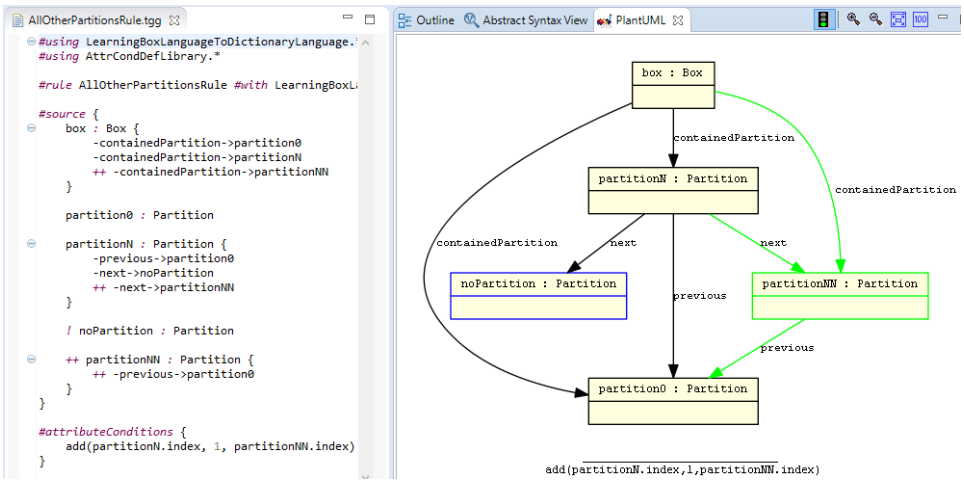


Figure 6.3: The completed `AllOtherPartitionsRule`

- As you can see, this rule doesn't assume to know the final `partition` in the transformation. It matches the `n`th partition as the partition without any next partition, then connects a new `n+1`th partition to `n` and `partition0` (clear as every partitions previous is `partition0`). Note that TGG transformations assume that the models are valid, i.e., have the expected structure (in our case meaning that the learning box is correctly "wired").[10] Remember that "blue" means "negative".

- Generate code for your improved TGG and re-run the transformation. It should work now without any error message. Inspect the protocol to understand what happened.

- Go ahead and add as many `partition`s and `card`s as you like to your model instance. Your TGG is now also able to handle a `box` with any number of `partition`s beautifully. For five partitions all with cards, the protocol gets quite interesting and is no longer a flat tree. Try it out!

[10] This should actually be formalised with a set of metamodel constraints that must be checked before a transformation is run, but we've omitted this here to simplify things.

7 Model Synchronization

At this stage, you have successfully created a trio of rules that can transform a `Box` with any number of `Partitions` and `Cards` into a `Dictionary` with an unlimited number of `Entrys` (or vice versa).

Now suppose you wanted to make a minor change to one of your current instances, such as adding a single new card or entry into one of your instances. Could you modify the instance models and simply run the transformation again to keep the target and sources consistent?

The current `fwd.src.xmi` file (Figure 6.1) has a partition with an index of three which, when transformed, correctly produces a target dictionary with all four entries. What would happen if we attempted to transform this dictionary back into the same learning box, with all four partitions?

- Copy and paste `fwd.trg.xmi`, renaming it as `bwd.src.xmi`.[11]
- Run `LearningBoxToDictionaryIntegrationTrafo.java` and inspect the resulting `bwd.trg.xmi` (Figure 7.1). Unfortunately, the newest `partition3` is missing!

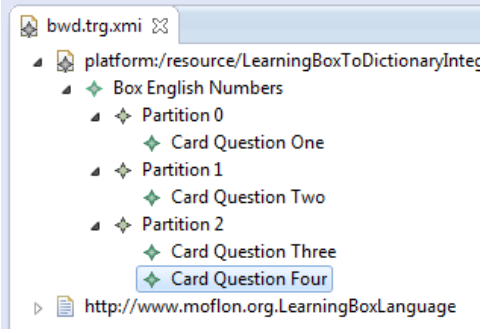


Figure 7.1: We lose information if the original box has more than 3 partitions

If you think about this for a moment it should be clear that our TGG-based backward transformation can only *create* a box with exactly three partitions. Why doesn't it try to apply `AllOtherPartitionsRule` you might be thinking? Well, given only the target model it is simply unclear how often this rule should be applied: Once? Twice? A hundred times? We could of course integrate user interaction to decide, but for a TGG of realistic size this would get pretty messy pretty soon.

[11] Feel free to either delete or rename the original `bwd.src.xmi` for later reference.

When going from the source to the target model, this information (how many partitions exist) is simply discarded and lost. So how can we prevent this information loss when we need to update our models in the future? Luckily, eMoflon can take care of this for you as it provides a *synchronization* mode to support incremental updates.

Let's change our source model by adding a new `Card` to `Partition3` and see if the partition still exists after synchronizing to and from the resulting `Dictionary` model.

- Make sure your `fwd.src.xmi` model is in the "extended" version depicted in Figure 6.1, and that you have a fresh triple of corresponding `fwd.corr.xmi` and `fwd.trg.xmi` created via the forward batch transformation.

- Our synchronisation algorithm requires an explicit "delta" representing the exact changes that are to be propagated. Although this can be extracted from different versions of a model this is not as easy as it sounds. To make matters worse, "delta recognition" is also often not unique, i.e., there are many possible deltas for the same pair of old and new models.

  As we are more interested in the actual synchronisation than in change recognition or "diffing", we require the exact delta which can come from anywhere in actual applications, including being programmed as a hard coded change operation in some kind of synchroniser GUI.

  To play around with a TGG, however, we provide a simple *delta editor*, which behaves very much like the standard EMF model editor, but records all the changes you make and persists them as a delta model, which we understand.

  To open the delta editor, right-click on `fwd.trg.xmi` and choose `Open With/Delta Editor`.

- Let's try a simple attribute change: As `Entry four:vier` was created from a card in the third partition, its default level is beginner (remember we assume easy cards have wondered further into the box). We'll we think this is wrong so let's change the level to `master`. Perform this change directly with the delta editor as if it were the standard EMF model editor (compare with Figure 7.2). Save the editor and close it. Take a look at the created `fwd.trg.delta.xmi` and inspect it. Although we do not (yet) have a fancy visualisation for delta models, it should be pretty easy to see how your change has been represented as a data structure. Finally, note that `fwd.trg.xmi` has *not* been changed yet, i.e., the synchroniser will do this when trying to propagate your

changes. This is important to understand as handling conflicts (work in progress, planned for a release in the far, far away future) might imply a compromise, i.e., not all your changes might be accepted or possible.

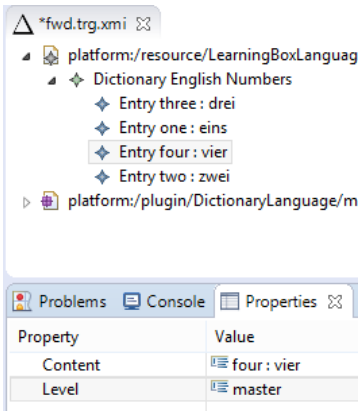


Figure 7.2: Change the level of an entry

- To backward propagate your changes, open `LearningBoxLanguage-ToDictionaryLanguageSync.java`, our generated stub for model synchronisation. Our stubs are optimized for this handbook so you do not have to change a thing. It's all plain and simple Java though, so you should be able to understand how our API is being used. The section where you can adjust and make changes is:

  ```
  // Adjust values as required
  String delta = "instances/fwd.trg.delta.xmi";
  String corr  = "instances/fwd.corr.xmi";
  BiConsumer<...> synchronizer = helper::syncBackward;
  ```

  These values all fit our current change (the created delta file, the relevant correspondence model, and we want to synchronise backwards), so we're ready to go. Hit `Run` and see what happens! If everything went well, `Card question:four` should have been relocated to `partition0` to fit its corrected difficulty level. The great thing is `partition3` is now empty, *and has been retained*. Not that the protocol has been updated, and that `fwd.trg.xmi` not contains these changes.

Remember this is all academic software and especially the delta editor is rather new and very buggy. Feel free to try out all kinds of deltas and if you think you've found a bug, please send us an email at `contact@moflon.org` and we'll be happy to fix it (or reply with a "back in your face!" if it's not one and *you* just got confused).

8 Model Generation with TGGs

In addition to model transformation and model synchronization, TGG specifications can be used directly to generate models. Often there is a need for large and randomly generated models for testing purposes and it's surprisingly hard and awful work to whip up such a generator that *only* creates valid models with respect to the TGG. Once you add or change a rule – puff! Your generator produces rubbish. It makes sense to automate this generation process and eMoflon provides some basic support.

- We assume for this section, that you've completed the previous section and have the source, target, and integration projects in their final versions in your workspace.

  In your project `src` folder locate and open `LearningBoxToDictionaryIntegrationModelGen.java` (Figure 8.1), our default stub for model integration.

```
30  public static void main(String[] args) throws IOException
31  {
32     // Set up logging
33     BasicConfigurator.configure();
34
35     AbstractModelGenerationController controller = new DefaultModelGenController();
36     controller.addContinuationController(new MaxRulePerformCounterController(20));
37     controller.addContinuationController(new TimeoutController(5000));
38     controller.setRuleSelector(new LimitedRandomRuleSelector().addRuleLimit("<enter rule name>", 1));
39
40     ModelGenerator gen = new ModelGenerator(LearningBoxToDictionaryIntegrationPackage.eINSTANCE, controller);
41     gen.generate();
42  }
```

Figure 8.1: Stub for the model generator

The `ModelGenerator` class uses an `AbstractModelGenerationController` to control the generation process (Line 35). In this default template the generation process will be terminated after 20 rules have been applied (`MaxRulePerformCounterController` (Line 36)). Additionally, the `TimeoutController` will terminate the process after 5000ms (Line 37). You can use the `MaxModelSizeController` class to terminate the generation process if a specific model size has been reached. The `RuleSelector` controls which rules are selected as the next to be executed. The built-in `LimitedRandomRuleSelector` always selects a random rule and has the additional feature to limit the number of performs for specific rules (Line 38). As with everything we generate, this is standard Java code, doesn't bite, and can be extended as you wish with your own controller classes and generation strategies.

- To ensure that we only get a model with a single root, change `<enter`

`rule name`> to `BoxToDictionaryRule` so that this island rule (which is always applicable) will only be applied once.

- Save the file and hit `Run`!

  You'll probably get some logging information in the console (Figure 8.2) containing information gathered during the generation process such as model size for each domain, number of performs for each rule, duration of generation process for each rule etc.

```
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - --- Model Generation Log ---
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - performs: 26
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - duration: 105ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - failures: 1
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - 29/11 nodes/edges created for source
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - 20/0 nodes/edges created for target
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - 20/40 nodes/edges created for correspondence
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - BoxToDictionaryRule
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    performs: 1
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    duration: 56ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    failures: 0
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - CardToEntryRule
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    performs: 19
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    duration: 35ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    failures: 0
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - AllOtherCardsRule
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    performs: 6
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    duration: 14ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    failures: 1
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - --- Model Generation Log ---
```

Figure 8.2: Logging output after model generation

In your `instances` folder there should now be a new folder named `generated-Models` with a timestamp suffix. It contains your newly generated source and target models. Have fun viewing. Why not try creating like seriously gigantic models?

Note that to support model generation for custom attribute condition in general, you might have to specify additional `#gen` adornments. No additional adornments were required for our new attribute condition, and all library conditions suppo

9 Conclusion and next steps

Absolutely amazing work – you've mastered Part IV of the eMoflon handbook! You've learnt the key points of Triple Graph Grammars and *bidirectional* transformations, and now know how to set up a TGG consisting of a schema and a set of rules. With these basic skills, you should be able to tackle quite a few bidirectional transformations using TGGs.

For detailed descriptions on the upcoming and previous parts of this handbook, please refer to Part 0, which can be found at `https://emoflon.github.io/eclipse-plugin/beta/handbook/part0.pdf`.

Cheers!

Glossary

Correspondence Types Connect classes of the source and target metamodels.

Graph Triples Consist of connected source, correspondence, and target components.

Link or correspondence Metamodel Comprised of all correspondence types.

Monotonic In this context, non-deleting.

Operationalization The process of deriving step-by-step executable instructions from a declarative specification that just states what the outcome should be but not how to achieve it.

Triple Graph Grammars (TGG) Declarative, rule-based technique of specifying the simultaneous evolution of three connected graphs.

TGG Schema The metamodel triple consisting of the source, correspondence (link), and target metamodels.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part V: Miscellaneous

For eMoflon Version 2.16.0

File built on 21st September, 2016

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

The eMoflon team
Darmstadt, Germany (September 2016)

Contents

Part VI:

And all that (eMoflon) jazz

URL of this document: https://emoflon.github.io/eclipse-plugin/beta/handbook/part5.pdf

Welcome to the miscellaneous part of our eMoflon handbook. You can consider this Part to be the 'bonus' or appendix area of the entire handbook series. Here we have collected and documented a series of advanced topics related to our tool. These include some tips and tricks you may find helpful while using the tool with Enterprise Architect (EA) and information about the protocol file generated with every Triple Graph Grammar (TGG) transformation which we were never able to explain in Parts IV or V. This entire part is kept rather compact, intended to be used mainly as a reference and consulted on demand.

Please note that if you're looking for instructions on how to properly export and import separate metamodels into the same project for work with TGG transformations, please refer to Part IV, Section 2.2, where we included detailed steps in the context of an example.

If you feel anything is missing from this part, or if you have any other comments or suggestions about the handbook series and our tool, feel free to contact us anytime at contact@emoflon.org.

1 Grokking Enterprise Architect

> Grok: "... to understand so thoroughly that the observer becomes a part of the observed."
>
> - Robert A. Heinlein, *Stranger in a Strange Land*

This section is a collection of a few of what we feel are the most important tips and tricks for working productively with Enterprise Architect (EA). We truly believe that spending the time to learn and practice these is necessary for a pleasant modelling experience.

1.1 Positioning elements

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

- To select a group of elements, either drag a selection box around the items or hold `Ctrl` and select each element one-by-one.
- In the top right corner of the last selected element, a small colon-styled symbol will appear (Figure 1.1). Click on this for a context list of different options you can simultaneously apply to all active elements. The same list appears on the toolbar above the diagram.
- Experiment to find out what effect each option has. The last symbol in the list opens a further drop-down menu with standard layout algorithms to organize your diagram automatically.
- Right-clicking any of the selected elements opens a different menu with a further set of layout options and their descriptions (Figure 1.2). `Align Centers` or `Same Height and Width` can be especially useful.

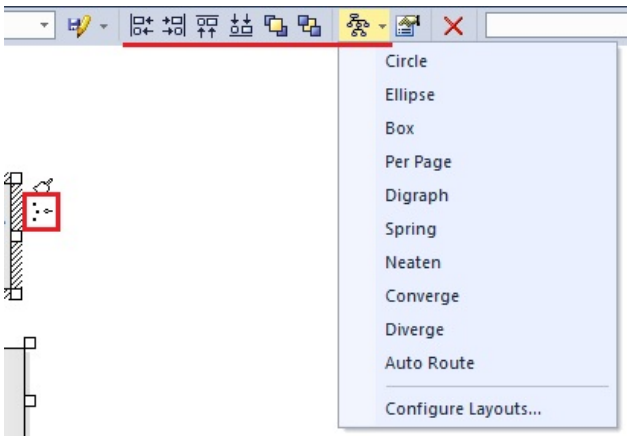


Figure 1.1: Setting the layout of multiple elements

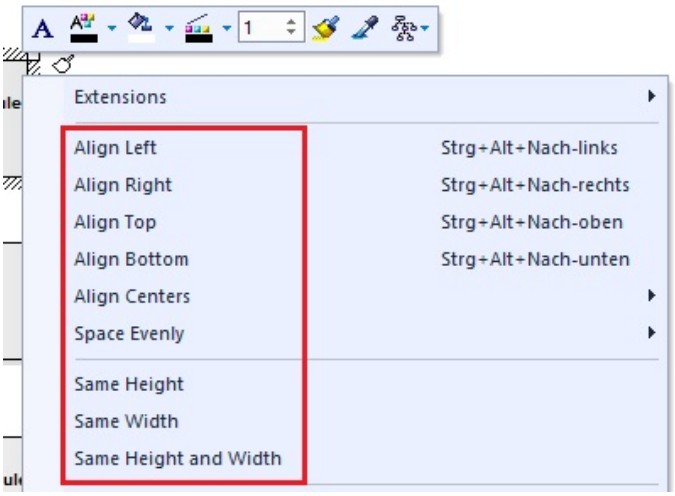


Figure 1.2: Further layout options

1.2 Bending lines to your will

Another important part of 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 (Figure 1.3). You can now pull the bending point and shift the line as you wish.

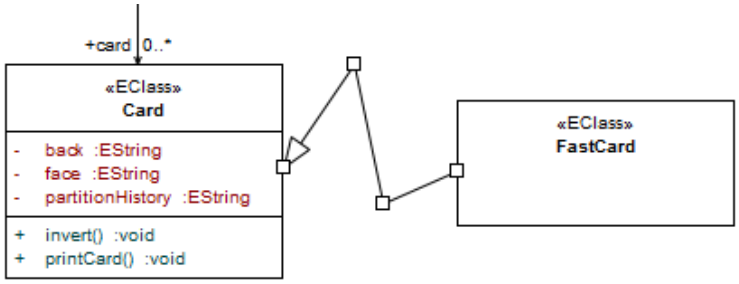


Figure 1.3: Adding bending points to a line

- You can create as many bending points as you wish, and you can *remove* them by holding down `Ctrl` and clicking once on the unwanted point.

1.3 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.* The complete model can always be browsed in its entirety via a tree view in the package browser. This space contains all elements that will be exported. The driving reason behind this setup is that diagrams typically do not contain all elements and one usually uses multiple (possibly redundant) diagrams to show the 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. We have added some support with the validation in the eMoflon add-in control panel, which can prompt a warning when an element cannot be found in any diagram,[1] but there's

[1] Review Part II, Section 2.8 for an example

currently no way to recover a deleted element.

A common mistake new users make is to remove an element by pressing `del`, and expecting the element to be deleted from the model. As you can probably guess, this is not the case as evidenced in the package browser (Figure 1.4).

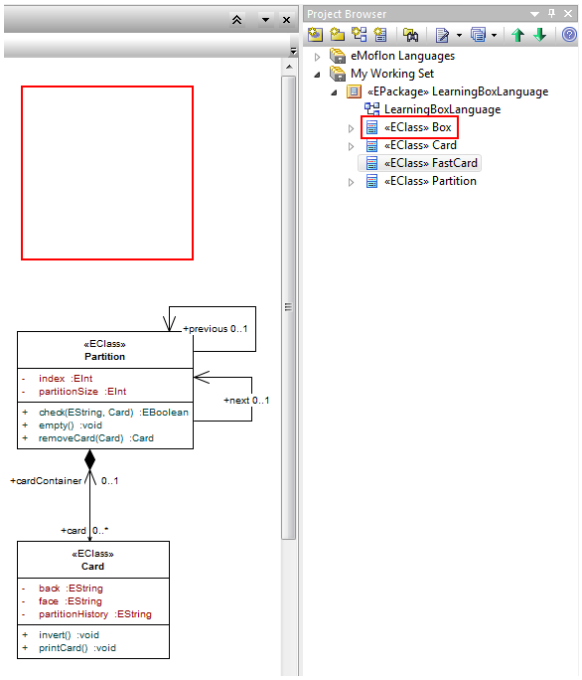


Figure 1.4: Removing an element from a diagram via pressing `Del` does not delete it from the model and it is still present in the package browser

- To fully delete an element from a model (not just a diagram), select it in the diagram and press `Ctrl + Del`. Confirm the action in the warning dialogue (Figure 1.5), and the element should no longer be in the project browser.

- Alternatively, elements can be deleted directly from project browser by right-clicking the item and navigating to the large red 'x' at the bottom of the context menu

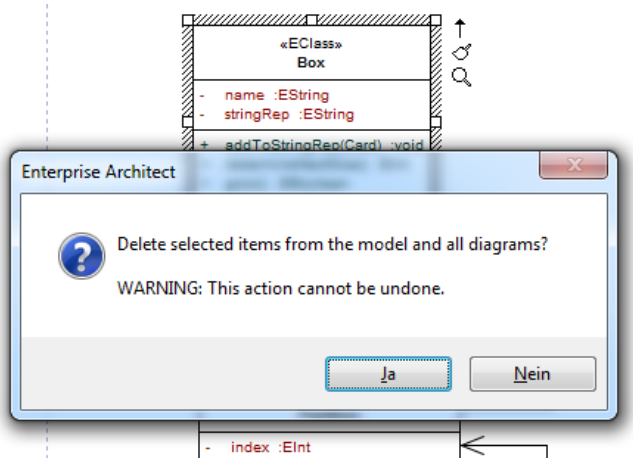


Figure 1.5: Deleting an element from a diagram and the model

1.4 Excluding certain projects from the export

You may find it sometimes necessary to exclude certain projects from your diagram export (such as the `MocaTree` model used in Part V). Some reasons for this could be (i) because the project is still a work in progress and simply not ready to be exported, (ii) because the complete project is present in the Eclipse workspace but has not been modelled completely in EA, and you wish to do this gradually on-demand, (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 stable and you do not want to wait for EA to process a known, unchanging model. Whatever the reason, you can prevent unnecessary exports by setting a certain *tagged value* of the project.

- Open your project in EA, and navigate to "View/Tagged Values" from the menu bar (Figure 1.6).
- The tagged value, *Moflon::Export*, should already be present and be set to a default `true` value (Figure 1.7). If you want the project to be ignored by the eMoflon's validation and/or export functions, change the value to `false` (and conversely back to `true` to export it again).

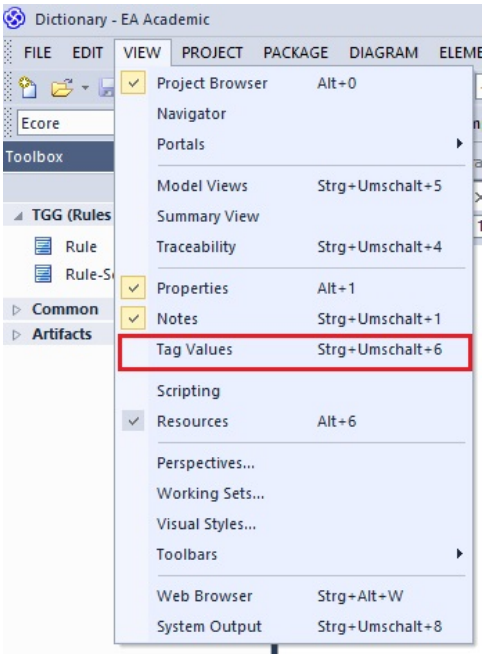


Figure 1.6: Opening the tagged values view

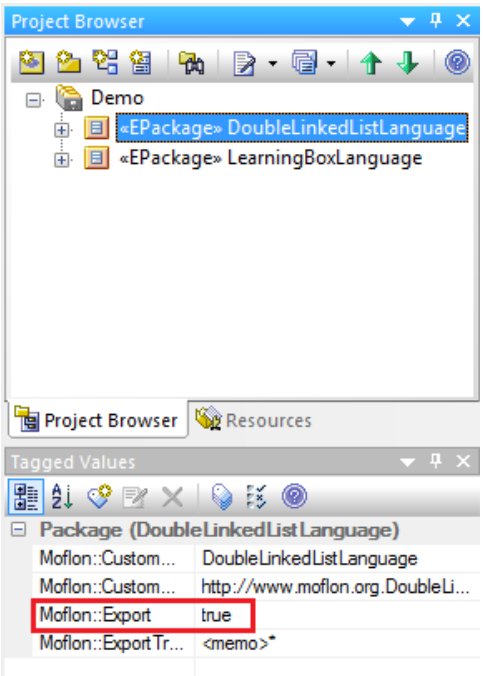


Figure 1.7: The `Moflon::Export` setting determines ignored projects

1.5 Getting verbose!

Although we use colours in SDMs to indicate when an element is to be matched (black), created (green), or destroyed (red), it sometimes makes sense to indicate these binding operators via explicit stereotypes (i.e., for black-and-white printouts of a model).

- Open the relevant diagram in the EA editor window and, depending on what type it is, press the `Verbose` button in either the `eMoflon SDM Functions` or `eMoflon TGG Functions` panel (Figure 1.8).

Figure 1.8: Add extra markup to colored links and objects in the current diagram

- This will add small `++` or `- -` symbols next to deleted and created elements in the current diagram (Figure 1.9). Press the button again to deactivate these indicators.

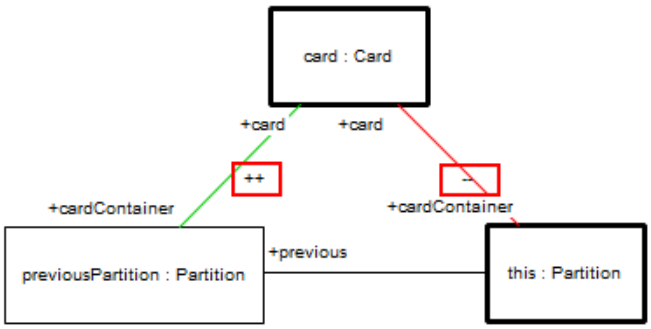


Figure 1.9: Diagram in verbose mode

1.6 Duplicating elements via drag-and-drop

Sometimes you'll have an element (or many) that are nearly identical, and life would be *so* much easier if you could copy and paste an existing one already. Suppose you want a copy of a `this` element, so you press `Ctrl + C`, followed by `Ctrl + V`. An error dialogue preventing the action will immediately raise, stating that the "... diagram already contains an instance of the element you are trying to paste." EA can only support unique objects, so you'll need to use the following process.

- In either a diagram or in the project browser, hold `Ctrl`, then drag the element you wish to duplicate. A confirmation-style dialogue will appear (Figure 1.10), and a properties window will follow. You must assign a unique name to the new element, or else you'll receive an error when you try to export the project later.

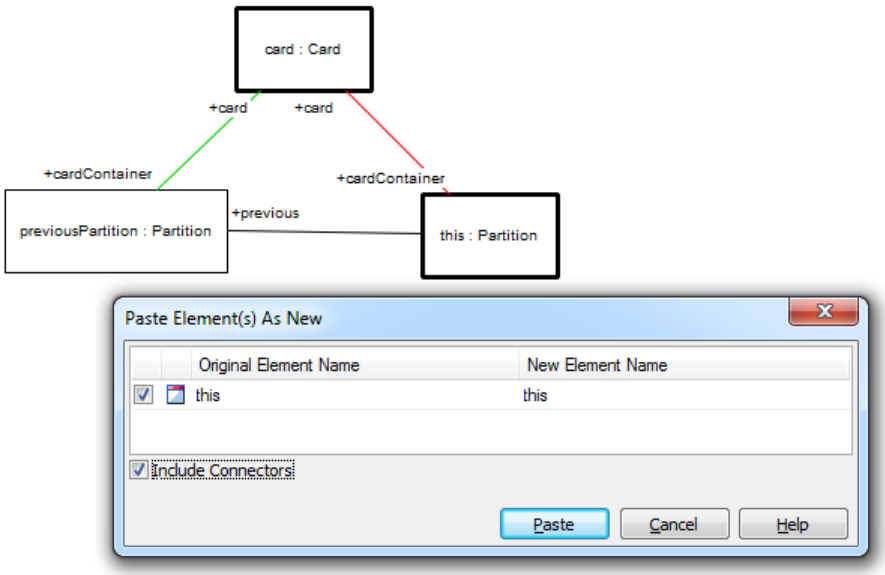


Figure 1.10: Copying elements

1.7 Seek, and ye shall find ...

EA has a model search function that can be quite handy for large models with thousands of elements and a brain that can't *quite* remember where something is.

- Select `Model Search Window` in the toolbar and enter the name of an element you wish to find (Figure 1.11).[2]

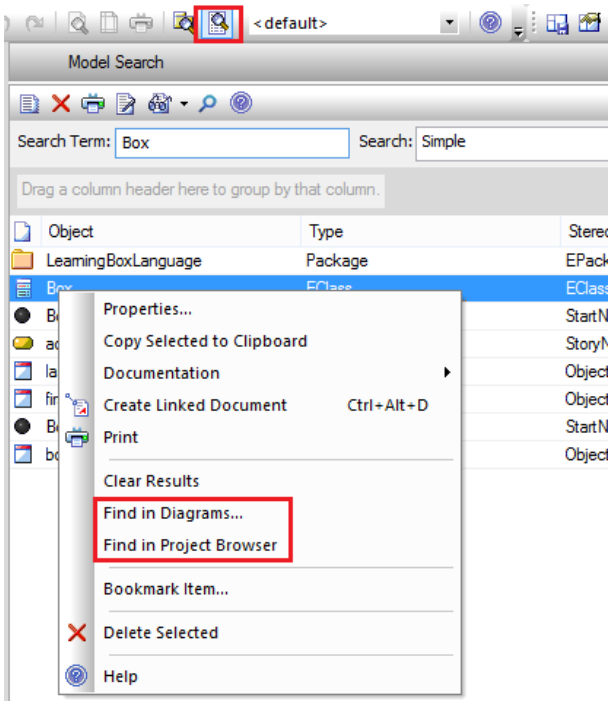


Figure 1.11: Model Search Window

- All elements that meet the search criteria are listed and you can right-click on each of the items and select one of the options above to locate the element.
- In a similar way, you can locate the corresponding class of an object by right clicking and selecting "Find/Locate Classifier in Project Browser."

[2] You can also access this window by pressing `Ctrl+Alt+A`

1.8 Advanced search

EA offers an even more advanced search capability using SQL.[3]

- To use this, first open the model search window via either the menu bar or by pressing `Ctrl + Alt + A`.
- Click the "Builder" button, and switch to the `SQL` tab (Figure 1.12).

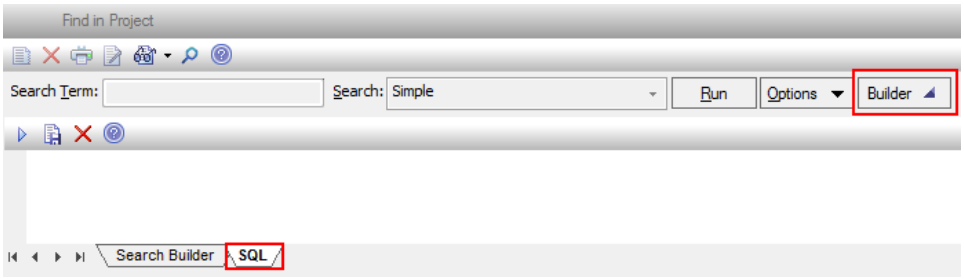


Figure 1.12: Advanced project search window

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 get you started:

- 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';
```

- To find all connectors attaching a note to an element

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

[3]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 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 SQL` button in the upper left corner of the editor toolbar (it shows a triangular shaped “play” icon), or press `F5` on your keyboard.

2 Using existing EMF projects in eMoflon

This chapter contains stepwise instructions on how to use existing EMF/Ecore projects with an eMoflon project. We will present an example of an existing metamodel which must be integrated with eMoflon before, for example, its transformation using SDMs can be specified. The basic workflow for using an existing EMF project in eMoflon is described in the following.

We will begin by implementing a small subset of the `Ecore -> GenModel` transformation, where `GenModel` is part of the EMF/Ecore standard. The *GenModel* for a given Ecore model can be viewed as a *wrapper* that contains additional generation-specific Java code details. These details are separated from the Ecore model to keep it free of such "low-level" information and settings.

2.1 Modelling relevant aspects in EA

The first step is to load an 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. Although this might sound frightening (especially for large, complex metamodels), the emphasis here on *relevant* indicates that only elements that are needed for the transformation have to be present in EA, where more can be added iteratively as the transformation grows.

If you find this section challenging or unclear, refer to Part II: Ecore for a detailed review of metamodel construction.

- Open Eclipse and create a new metamodel project named `EcoreToGenModel`, do not select the `Add Demo Specification` option in the project wizard window.
- A new specifications folder with the project name should have been loaded into the workspace.
- Double-click the generated `EcoreToGenModel.eap` file to open your project in EA. Explore the project browser and make note of the packages already present in EA under `eMoflon Languages`, especially `Ecore` which we shall use in this transformation.
- Select the root note `My Working Set` and create a new package named `GenModelLanguage`.
- Add a new Ecore diagram and model the elements as depicted in Figure 2.1. You'll need to create the three EClasses on the left, but

`Ecore::EPackage` and `Ecore::EClass` are to be drag-and-dropped and pasted as links from the project browser.

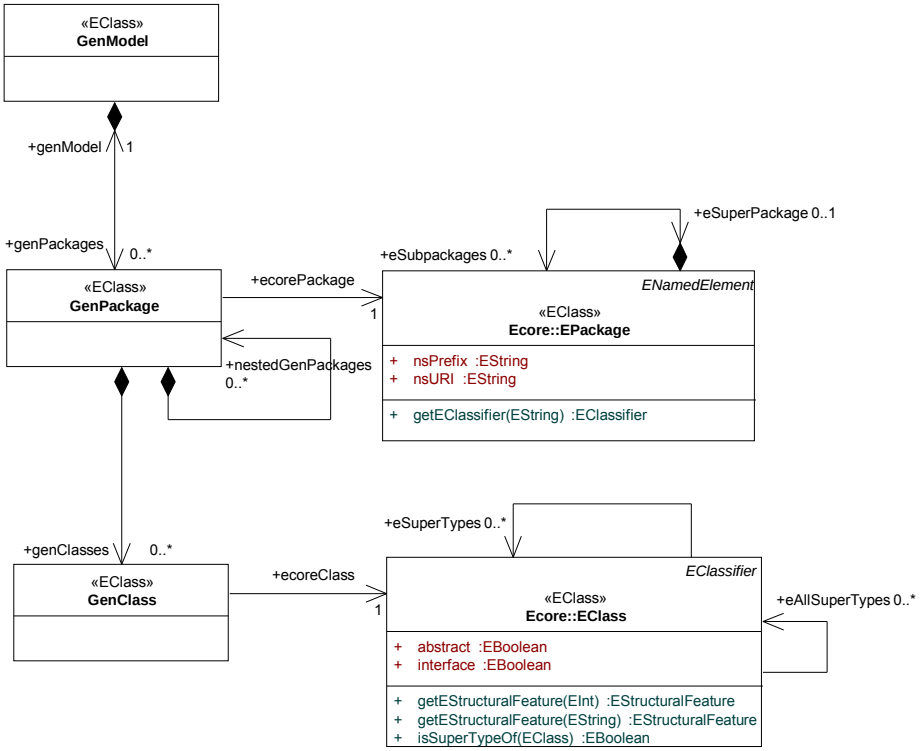


Figure 2.1: Metamodel of `GenModel`

- Please note that the actual `GenModel` metamodel contains many more elements, but this subset is sufficient for our task. Although this subset can be incomplete, it must be correct and not contradict the actual `GenModel` metamodel in any way!
- Navigate to the project browser again and create another package named `Ecore2GenModel`. This will contain the `Transformer` class; Create and complete its Ecore diagram as depicted in Figure 2.2.
- Carefully double-click each method to create and implement their SDMs as depicted in Figures 2.3 and 2.4.

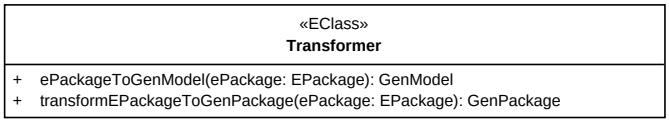


Figure 2.2: Methods in `Transformer`

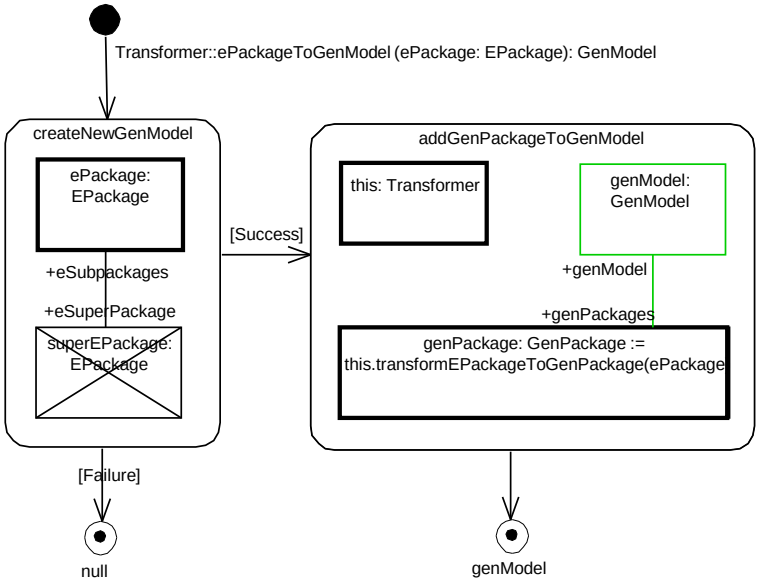


Figure 2.3: Main method for `EPackage` to `GenModel` transformation

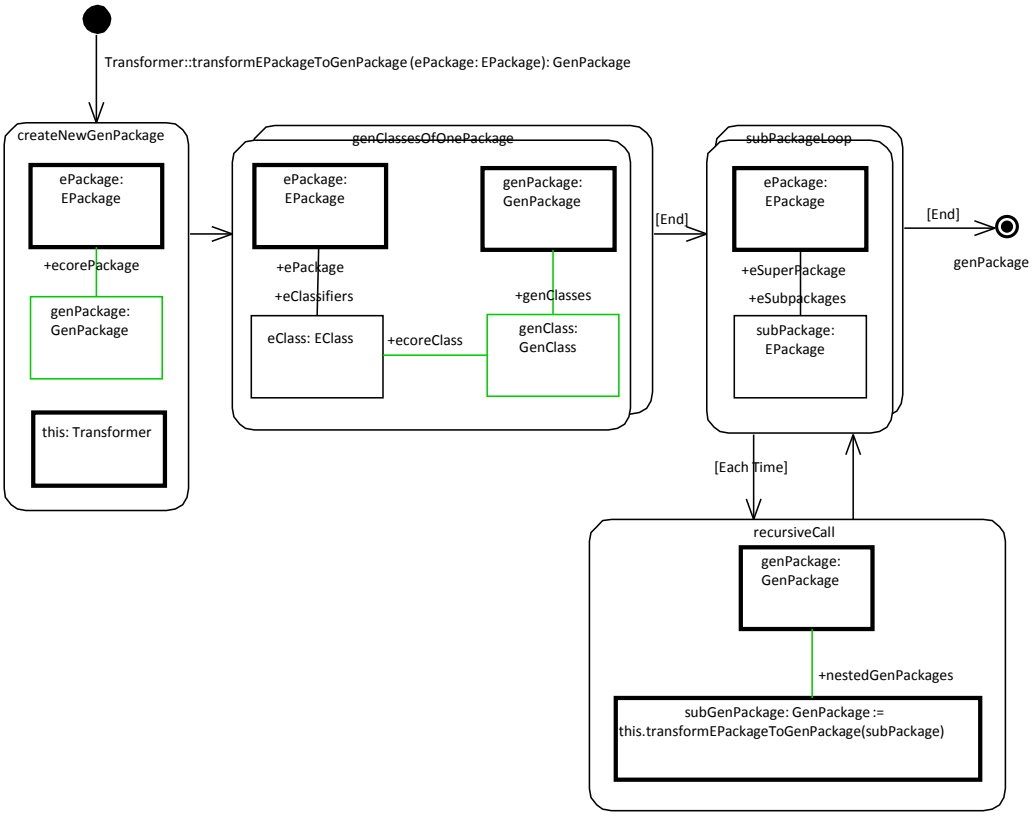


Figure 2.4: Helper function to transform all `EPackages` to `GenPackages`

2.2 Configuration for code generation in Eclipse

Since there is already generated code for the existing `GenModel` metamodel (provided via the Eclipse plugin), we do *not* want to export our incomplete subset of `GenModel` from EA. Instead, we need to configure Eclipse to access the elements specified in our partial metamodel from the complete metamodel.

- In EA, right-click your `GenModelLanguage` package and select "Properties..."
- Navigate to "Properties/Moflon" in the dialogue window and update the tagged `Moflon::Export` value to `false` (Figure 2.5).

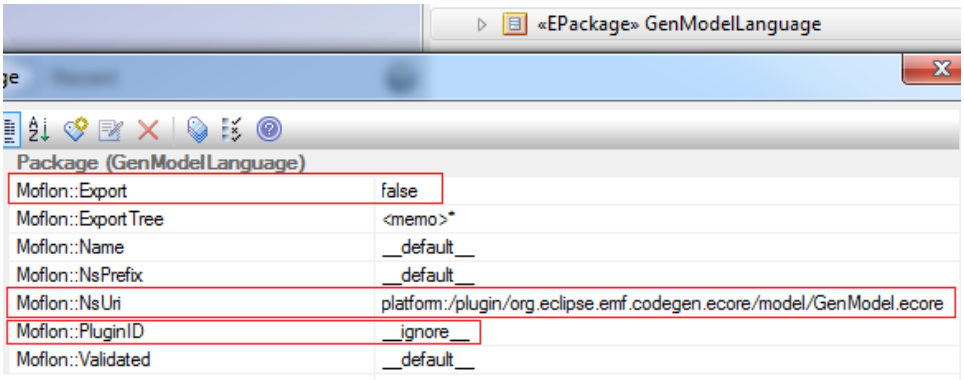


Figure 2.5: Update the `GenModel` export option and other tagged values

- Next we have to set the "real" URI of the project to be used in Eclipse so that the relevant references are exported properly. Set the value of `Moflon::NsUri` to `platform:/plugin/org.eclipse.emf.codegen.ecore/model/GenModel.ecore`. As the default plugin ID generation provided by eMoflon is also not valid here, set the value of `Moflon::PluginID` to `__ignore__` (two underscores before and after!). The three relevant values to be set are shown in Figure 2.5.
- Validate and export all projects as usual to your Eclipse workspace, and update the metamodel project by pressing `F5` in the package explorer.
- Right-click `Ecore2GenModel` once more and navigate to "Plug-in Tools/Open Manifest." The plug-in manager should have opened in the editor with a series of tabs at the bottom.

- Switch to the `Dependencies` tab. Press `Add` and enter `org.eclipse.emf.codegen.ecore`. This plug-in includes both the `Ecore` and `GenModel` libraries we require for successful compilation of the transformation code.

Although we have already specified the URI of the existing project (in this example, `GenModel`) as tagged project values, we still have to configure a few things for code generation.

- Expand the `Ecore2GenModel` project folder and open the `moflon.properties.xmi` file tree. Right-click the properties container, and create a new `Additional Dependencies` child. Double click the element to open its properties tab below the editor, and as shown in Figure 2.6, update its `Value` to:

```
platform:/plugin/org.eclipse.emf.codegen.ecore/model/GenModel.ecore
```

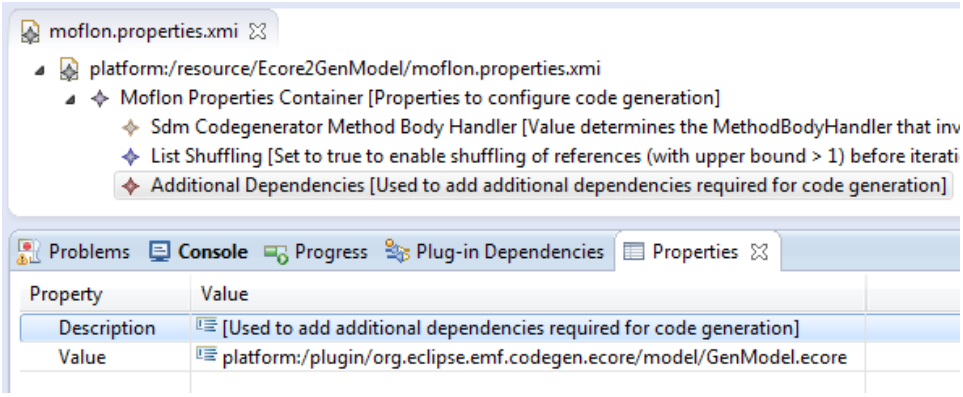


Figure 2.6: Setting properties for code generation

- Similarly, add a second `Additional Used Gen Packages` child and set its value to:

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

Finally, to compenstate for some cases where our naming conventions were violated, analogously add the following mapping as corrections:

- Add an *import mapping* child for correct generation of imports, setting the key as `genmodel` (depicted in Figure 2.7) and value to:
  `org.eclipse.emf.codegen.ecore.genmodel`

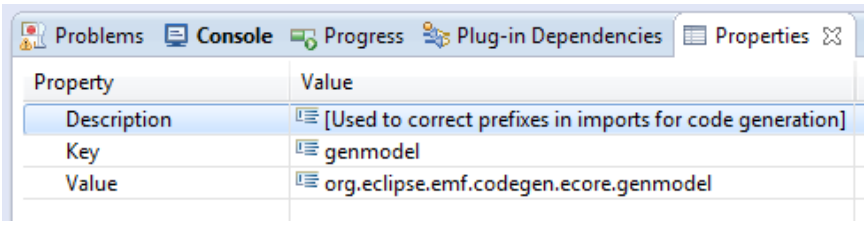


Figure 2.7: Correcting default conventions for generating imports

- Finally, add 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. Set its key as `genmodel`, and its value to: `GenModelFactory`.
- Its now time to generate code for the project. If everything worked out and the generated code compiles, you can ensure that the transformation behaves as expected by invoking the methods and transforming an ecore file to a corresponding genmodel.

As a final remark, note that import and factory mappings are not always necessary, `GenModel` is in this sense a particularly nasty example as it violates all our default conventions.

3 eMoflon in a Jar

This section describes how to package code generated with eMoflon into runnable Jar files, which is useful if you want to build applications for end-users.

We distinguish between repository, i.e., SDM-based, and integration, i.e., TGG-based, projects.

3.1 Packaging SDM projects into a Jar file

The following explanations use the demo specification that is shipped with eMoflon to explain the workflow of building a runnable Jar file.

- Open a fresh workspace and add to it the eMoflon Demo specification by selecting the "Install, configure and deploy Moflon" button and open the "Install Workspace" menu bar. Select the "Demo Workspace".
- Generate code for the demo and verify the result by running the test cases in *DemoTestSuite*.
- Add a suitable main method to `NodeTest`, for instance:

```
public static void main(String[] args) {
    System.out.println("Begin of test runs");
    new NodeTest().testDeleteNode();
    new NodeTest().testInsertNodeAfter();
    new NodeTest().testInsertNodeBefore();
    System.out.println("End of test runs");
}
```

- Run `NodeTest` as "Java Application" (*not* as "JUnit Test"). Now you have a new launch configuration named "NodeTest".
- Now, select the repository project (containing the generated code) and the project *DemoTestSuite*. You do not need to add the project containing the EA project. Right-click and select "Export...". Choose "Runnable JAR file".
- On the next page, select the launch configuration you just created by running `NodeTest` and an appropriate target location for your Jar file. The libraries should be packaged or extracted into the generated Jar file.
- Afterwards, open up a console in the folder containing the generated Jar file and execute it as follows:

```
java -jar [GeneratedJarFile.jar]
```

3.2 Packaging TGG projects into a Jar file

In the following, you will create a runnable Jar from a TGG specification. We assume that you have some existing TGG implementation and that you want to execute the `main` method in class `org.moflon.tie.MyIntegrationTrafo`.

Note: The following instructions show how to use Eclipse's built-in facility for generating runnable Jars. There are other build tools such as ant, Maven or Gradle that facilitate this process.

- Ensure that your TGG rules from within Eclipse. For simplicity, we assume that your main method currently resembles the following snippet:

```
public static void main(String[] args) throws IOException {
    // Set up logging
    BasicConfigurator.configure();

    // Forward Transformation
    MyIntegrationTrafo helper =
        new MyIntegrationTrafo();
    helper.performForward("instances/fwd.src.xmi");
}
```

  The default transformation helper should print a short success message when the forward transformation has finished.

- Before packaging your project, you have to change the ways how the TGG rules are being loaded (in the constructor of `MyIntegrationTrafo`). Replace this method call

```
loadRulesFromProject("..");
```

  with

```
File jarFile = new File(MyIntegrationTrafo.class
    .getProtectionDomain().getCodeSource()
    .getLocation().toURI().getPath());
loadRulesFromJarArchive(
    jarFile,
    "/MyIntegration.sma.xmi");
```

  This is a tiny trick to find out the name of the Jar file that you are about to build. If you already know the name of your Jar file (e.g., "tggInAJar.jar"), you could simply use the following code:

```
loadRulesFromJarArchive(
    "tggInAJar.jar",
    "/MyIntegrationTrafo.sma.xmi");
```

- Next, make the "model" directory a source folder by right-clicking it and selecting "Build Path/Use as Source Folder". This will make the contents of "model" available in the Jar file to be built.

- Now, your projects are ready to be packaged. Select all projects that are involved in your TGG, that is, the project of the source and target metamodel as well as the actual integration project.

  Right-click the projects and select "Export..." and then "Java/Runnable JAR File".

- Select the appropriate launch configuration (named "MyIntegrationTraf"), choose the export destination, and make sure that the library handling is set to "Extract required libraries".

- After a successful export, locate the generated Jar file. The program expects to find the source model of the transformation at the following path, relative to the folder containing your Jar file: "instances/fwd.src.xmi".

  Now, let's take the transformation for a spin:

```
java -jar [GeneratedJarFile.jar]
```

4 Useful shortcuts

This page is a simple list of special hotkeys you might find useful while working with eMoflon in either EA or Eclipse. Please note standard shortcuts, such as `Ctrl + S` and `Ctrl + Z`, are still applicable in most cases.

4.1 In Eclipse (general)

Note: `I` indicates *in Integrator window*, and `GK` indicates *German keyboards only*

`Ctrl + Space`	Auto-type completion
`Ctrl + 1` (problems tab)	Quick-fix menu
`Alt + arrow` (I)	Proceed to next step
`Shift + Alt + arrow` (I)	Fast navigation
`Shift + Ctrl + Alt + arrow` (I)	Proceed to next breakpoint
`Shift + Ctrl + AltGr + arrow` (I, GK)	Proceed to next breakpoint

4.2 In Eclipse (eMoflon Plugin)

`Alt + Shift + E`	Open list of all available eMoflon commands
`Alt + Shift + E, B`	Trigger a build without clean
`Alt + Shift + E, C`	Trigger a clean and build
`Alt + Shift + E, D`	Convert file to visual representation (dot)
`Alt + Shift + E, G`	Start integrator (on correspondence file)
`Alt + Shift + E, I`	Create/update injections
`Alt + Shift + E, M`	Convert project to textual syntax
`Alt + Shift + E, P`	Add ANTLR parser and/or unparser
`Alt + Shift + E, V`	Valide Ecore file
`Alt + Shift + E, X`	Export and build EAP file

4.3 In EA

Note: `D` indicates *in Diagram*, and `PB` indicates *in Project Browser*

`Alt + Enter`	Selected element `Properties` dialogue
`F9 + EClass`	Class `Attribute` editor
`F10 + EClass`	Class `Operations` editor
`Space` (D)	Current toolbar menu
`Del + Ctrl + element` (D/PB)	Delete element from model
`Ctrl + element` (D/PB)	Duplicate and create new element
`Ctrl + Alt + A`	Open Model Search Window
`Alt + G + element` (D)	Highlight Element (PB)

5 Legacy support for CodeGen2

Since eMoflon 1.8, the default code generator is *Democles*. The previous code generator *CodeGen2* is available, but no longer officially supported.

This sections describes how to configure your project to use CodeGen2.

- Install the eMoflon CodeGen2 feature: Select "Help/Install new software...", choose the eMoflon update site and tick "eMoflon CodeGen2" (Figure 5.1). Proceed with "Next" and follow the instructions to install the feature.

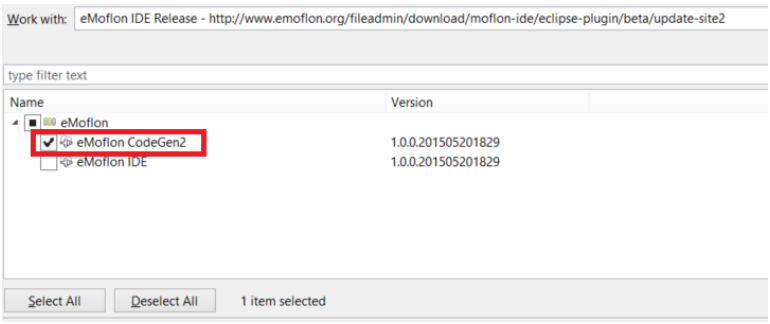


Figure 5.1: Installing the eMoflon CodeGen2 feature

- Open the file "moflon.properties.xmi" in your project(s) and set the code generation strategy to CODEGEN2 (Figure 5.2).

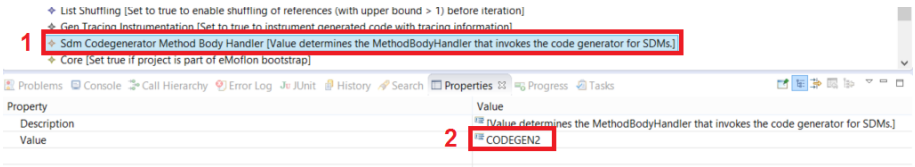


Figure 5.2: Code generation strategy selection in "moflon.properties.xmi"

- Open the file "META-INF/MANIFEST.MF" in your project(s) and add the following dependency: *org.moflon.sdm.codegen2.runtime* (Figure 5.3).
- Clean and build your project using the eMoflon context menu (Alt+Shit+E, B).

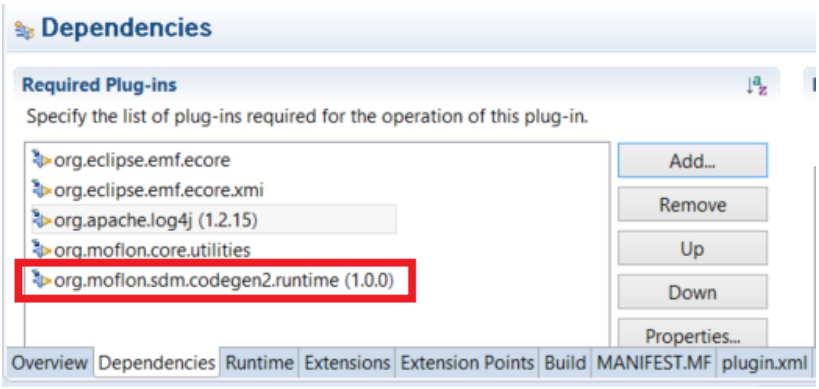


Figure 5.3: Dependency to CodeGen2 runtime in “MANIFEST.MF”

6 Creating and Using Enumerations in Enterprise Architect

This section describes how to create enumeration types in your metamodel. For illustration purposes, we use the linked-list demonstration project from Part I.

Suppose we want to assign one of several predefined colors (e.g., red, green, blue) to each `Node` in a `List`. To represent the colors, we create an enumeration called `Color` with three elements: `Color.RED`, `Color.GREEN` and `Color.BLUE`.

- Let's start with the double-linked list demonstration specification, which is readily provided with eMoflon: Create a new meta-model project ("File/New/Other..."), then "eMoflon/New Metamodel Wizard"), call your project "Demo", and tick "Add Demo Specification".
- Open the EAP file "Demo.eap" and navigate to the diagram "org.moflon.demo.doublelinkedlist".
- Now, add a new "EEnum" type called color. This can be done via the Toolbox ("Diagram/Toolbox") or by pressing space while the cursor is inside the digram area. Your diagram should resemble Figure 6.1.

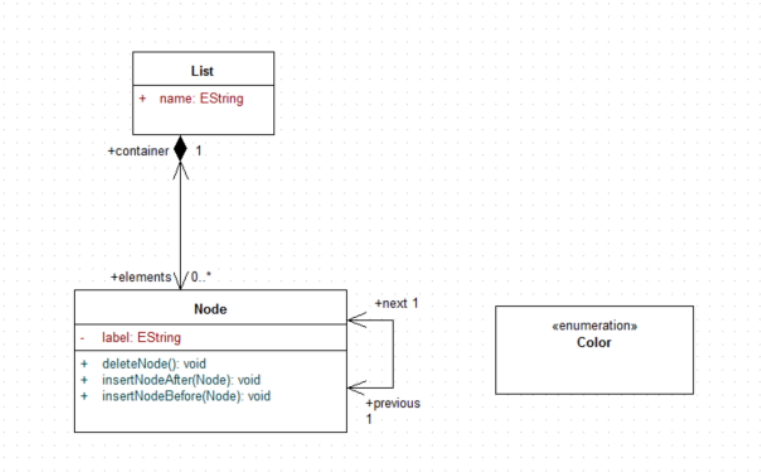


Figure 6.1: Creating a new EEnum

- We now create the three colors that our list nodes may have. An enum constant is a special attribute. Therefore, open the attriutes view of `Color` ("Right-click/Features & Properties/Attributes...") and add

three attributes as shown in Figure 6.2. Make sure that the type of the attributes is `Color` and that each attribute has an "initial value".

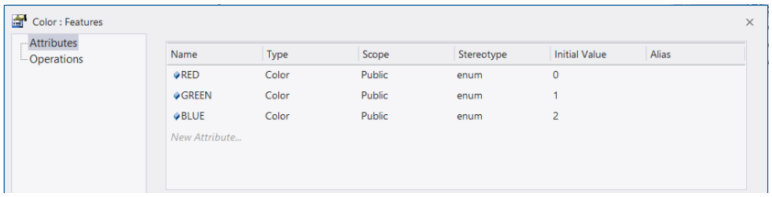


Figure 6.2: Creating the three color attributes `RED`, `GREEN`, and `BLUE`

- Finally, create a `color` attribute of type `Color` in EClass `Node` (Figure 6.3).

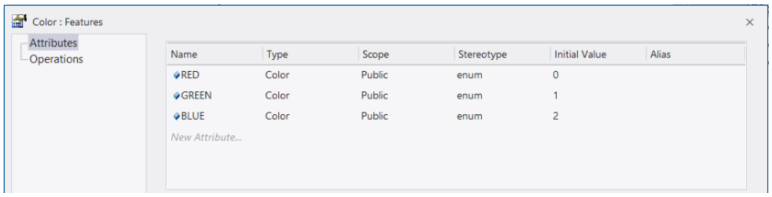


Figure 6.3: `color` attribute of EClass `Node`

- Validate and export and build your metamodel.
- A minimal test for the new feature could be implemented in the class `NodeTest` as follows:

Listing 6.1: Test for coloring nodes

```
@Test
public void testAddColor() throws Exception {
        Node node =
            DoublelinkedlistFactory.eINSTANCE.createNode();
        node.setColor(Color.RED);
}
```

7 Convert your TGG to textual syntax

Your "old" TGG specified with EA (visual syntax) can easily be converted to our new textual syntax:

- In Eclipse, navigate to the `model` folder of your TGG project and right click on the `pre.tgg.xmi` file (`<your TGG>.pre.tgg.xmi`). Choose `Convert to MOSL` from the eMoflon context menu (Figure 7.1).
- In `src/org.moflon.tgg.mosl`, check the created `.tgg` file which represents your TGG in textual syntax. (Figure 7.2).

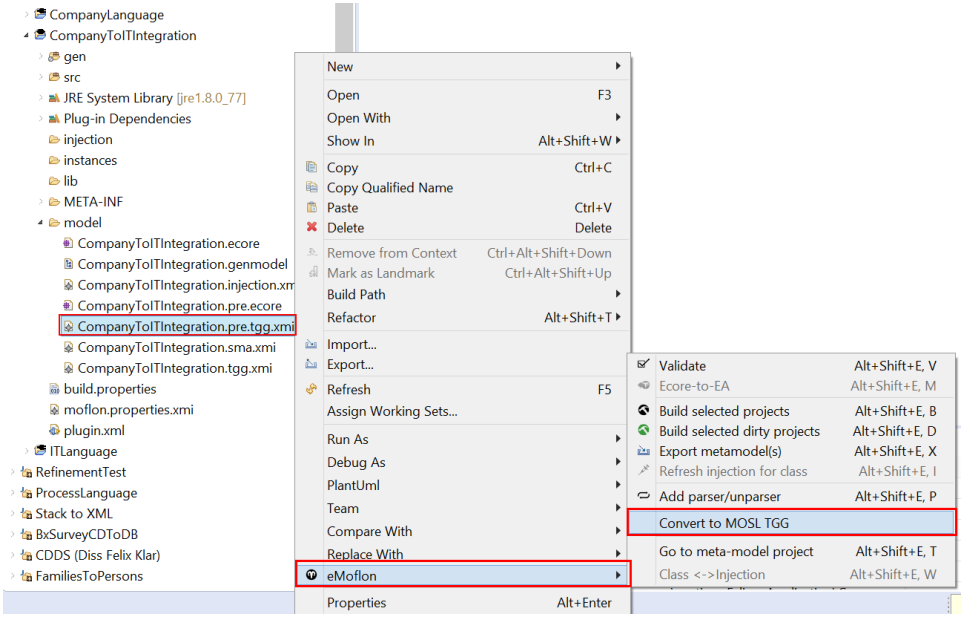


Figure 7.1: Convert your TGG to textual syntax

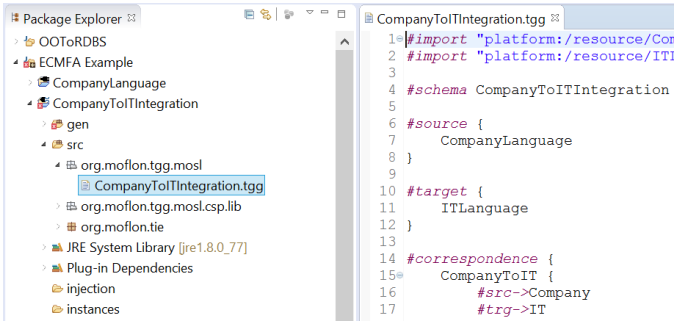


Figure 7.2: The created `.tgg` file representing your TGG in textual syntax

8 Glossary

Abstract Syntax Defines the valid static structure of members of a language.

Activity Top-most element of an SDM.

Activity Edge A directed connection between activity nodes describing the control flow within an activity.

Activity Node Represents atomic steps in the control flow of an SDM. Can be either a story node or statement node.

Assignments Used to set attributes of object variables.

Attribute Constraint A non-structural constraint that must be satisfied for a story pattern to match. Can be either an assertion or assignment.

Bidirectional Model Transformation Consists of two unidirectional model transformations, which are consistent to each other. This requirement of consistency can be defined in many ways, including using a TGG.

Binding State Can be either *bound* or *unbound/free*. See *Bound vs Unbound*.

Binding operator Determine whether a variable is to be *checked*, *created*, or *destroyed* during pattern matching.

Binding Semantics Determines if an object variable *must* exist (*mandatory*), may not exist (*negative*; see *NAC*), or is *optional* during *pattern matching*.

Bound vs Unbound Bound variables are completely determined by the current context, whereas unbound (free) variables have to be determined by the *pattern matcher*. `this` and parameter values are always bound.

Concrete Syntax How members of a language are represented. This is often done textually or visually.

Constraint Language Typically used to specify complex constraints (as part of the static semantics of a language) that cannot be expressed in a metamodel.

Correspondence Types Connect classes of the source and target metamodels.

Dangling Edges An edge with no target or source. Graphs with dangling edges are invalid, which is why dangling edges are avoided and automatically deleted by the pattern matching engine.

Dynamic Semantics Defines the dynamic behaviour for members of a language.

EA Enterprise Architect; The UML visual modeling tool used as our visual frontend.

EBNF Extended Backus-Naur Form; Concrete syntax for specifying context-free string grammars, used to describe the context-free syntax of a string language.

Edge Guards Refine the control flow in an activity by guarding activity edges with a condition that must be satisfied for the activity edge to be taken.

Endogenous Transformations between models in the same language (i.e., same input/output metamodel).

Exogenous Transformations between models in different languages (i.e., unique metamodel instances).

Grammar A set of rules that can be used to generate a language.

Graph Grammar A grammar that describes a graph language. This can be used instead of a metamodel or type graph to define the abstract syntax of a language.

Graph Triples Consist of connected source, correspondence, and target components.

In-place Transformation Performs destructive changes directly to the input model, thus transforming it into the output model. Typically *endogenous*.

Link or correspondence Metamodel Comprised of all correspondence types.

Link Variable Placeholders for links between matched objects.

Literal Expression Represents literals such as true, false, 7, or "foo."

Meta-Language A language that can be used to define another language.

Meta-metamodel A *modeling language* for specifying metamodels.

Metamodel Defines the abstract syntax of a language including some aspects of the static semantics such as multiplicities.

MethodCallExpression Used to invoke any method.

Model Graphs which conform to some metamodel.

Modelling Language Used to specify languages. Typically contains concepts such as classes and connections between classes.

Monotonic In the context of TGGs, a non-deleting characteristic.

NAC Negative Application Condition; Used to specify structures that must not be present for a transformation rule to be applied.

Object Variable Place holders for actual objects in the current model to be determined during pattern matching.

ObjectVariableExpression Used to reference other object variables.

Operationalization The process of deriving step-by-step executable instructions from a declarative specification that just states what the outcome should be but not how to achieve it.

Out-place Transformation Source model is left intact by the transformation which creates the output model. Can be *endogenous* or *exogenous*.

Parameter Expression Used to refer to method parameters.

(Graph) Pattern Matching Process of assigning objects and links in a model to the object and link variables in a pattern in a type conform manner. This is also referred to as finding a match for the pattern in the given model.

Statement Nodes Used to invoke methods as part of the control flow in an activity.

Static Semantics Constraints members of a language must obey in addition to being conform to the abstract syntax of the language.

Story Node *Activity nodes* that contain *story patterns*.

Story Pattern Specifies a structural change of the model.

Triple Graph Grammars (TGG) Declarative, rule-based technique of specifying the simultaneous evolution of three connected graphs.

Type Graph The graph that defines all types and relations that form a language. Equivalent to a metamodel but without any static semantics.

TGG Schema The metamodel triple consisting of the source, correspondence (link), and target metamodels.

Unification An extension of the object oriented "Everything is an object" principle, where everything is regarded as a model, even the metamodel which defines other models.

9 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; how-

ever, 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 DAMAGES 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`.