An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part II: Ecore

For eMoflon Version 1.7.0

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

The eMoflon team
Darmstadt, Germany (Febuary 2015)

Contents

Part II:

Leitner's Learning Box

Approximate time to complete: 1 hour

URL of this document: http://tiny.cc/emoflon-rel-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 Fig. 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 Fig. 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.

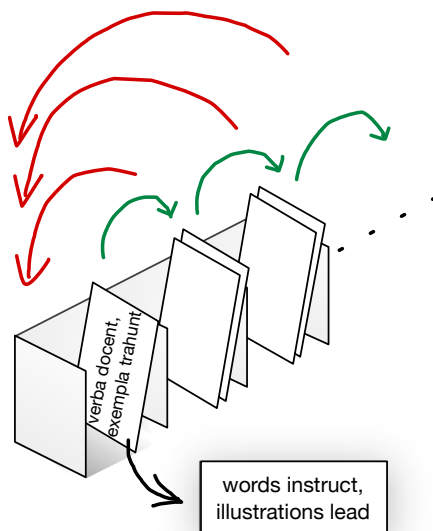[1] http://en.wikipedia.org/wiki/Leitner_system

Figure 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, Fig. 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 lan-* *Meta-*
guage or *meta-language*. For metamodelling with eMoflon, we support *Ecore* *metamodel*
as a modelling language and it defines types like `EClass` and `EReference`, *Meta-*
which we will be using to specify our metamodels. Alternate modelling *Language*
languages include MOF, UML and Kermeta.

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 usu-
ally specified using an extra *constraint language* such as the Object Con- *Constraint*
straint Language (OCL). This idea, however, is beyond the scope of this *Language*
handbook. We'll stick to metamodels without an extra 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* *Dynamic*
of the language. Although these rules can be defined as a set of separate *Semantics*
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.

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 *Model*
according to the abstract syntax and must adhere to the static semantics).

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 visual demo, you can begin modeling 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," and start a new visual project (this time without the demo specifications) named `Leit-nersLearningBox` (Fig. 2). Open the empty `.eap` file in EA.
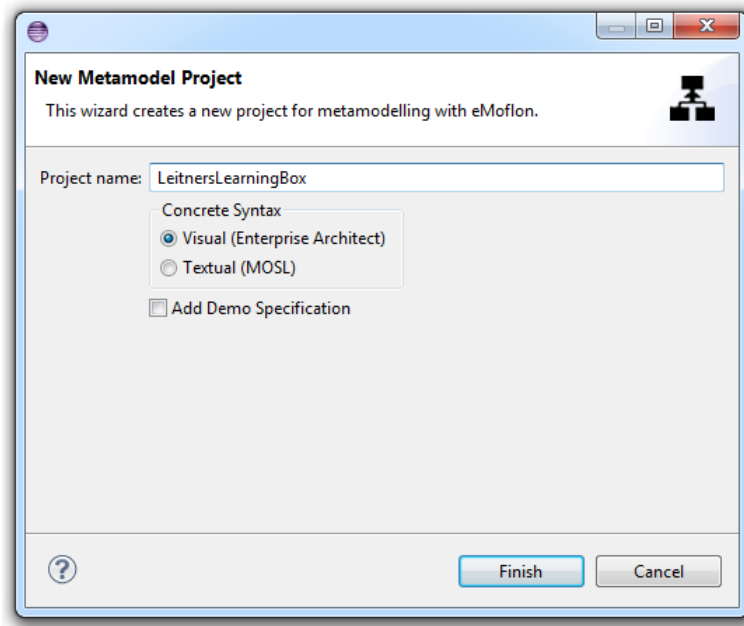

Figure 2: Starting a new visual project

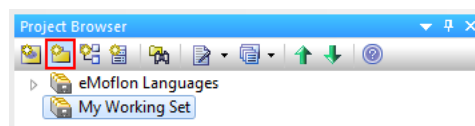▶ In EA, select your working set and press the "Add a Package" button (Fig. 3).

Figure 3: Add a new package to `MyWorkingSet`

▶ In the dialogue that pops up (Fig. 4), enter `LearningBoxLanguage` as the name of the new package. Make sure `Class View` is selected, and click `OK`.
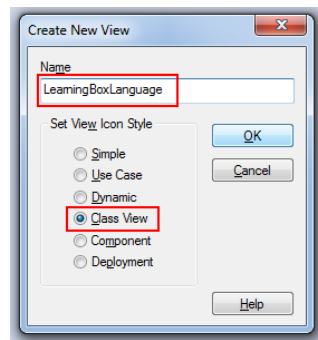
Figure 4: Enter the name of the new package

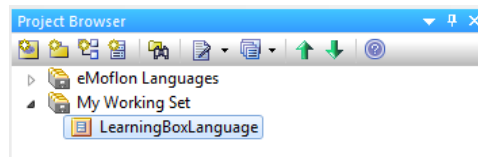▶ Your `Project Browser` should now resemble Fig. 5.

Figure 5: State after creating the new package

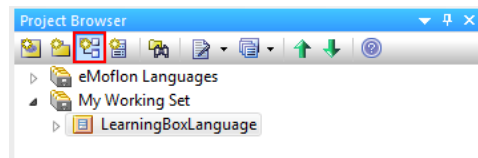▶ Now select your new package and create a "New Diagram" (Fig. 6).

Figure 6: Add a diagram

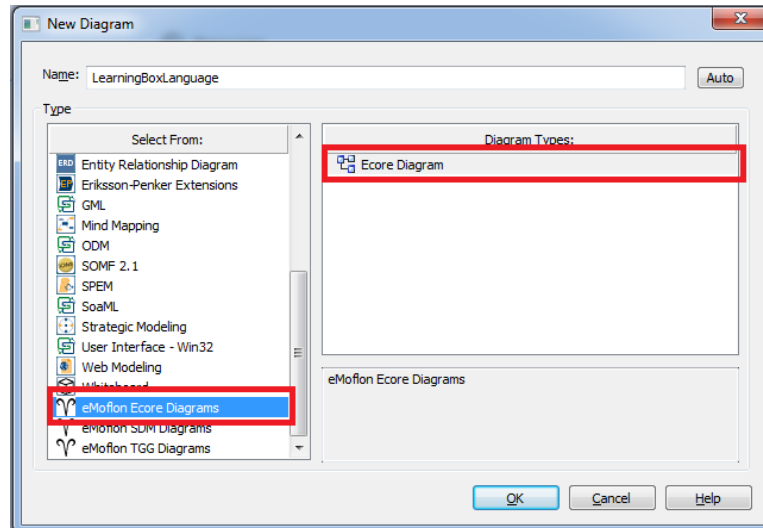▶ In the dialogue that appears (Fig. 7), choose `eMoflon Ecore Diagrams` and press `OK`.

Figure 7: Select the ecore diagram type

▶ After creating the new diagram, your `Project Browser` should now resemble Fig. 8. You'll notice that your `LearningBoxLanguage` package has been transformed into an EPackage.
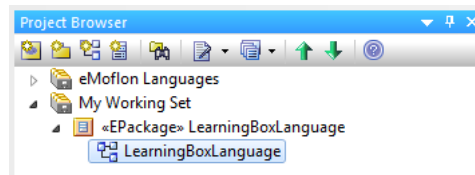

Figure 8: State after creating diagram

▶ 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 (Fig. 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`
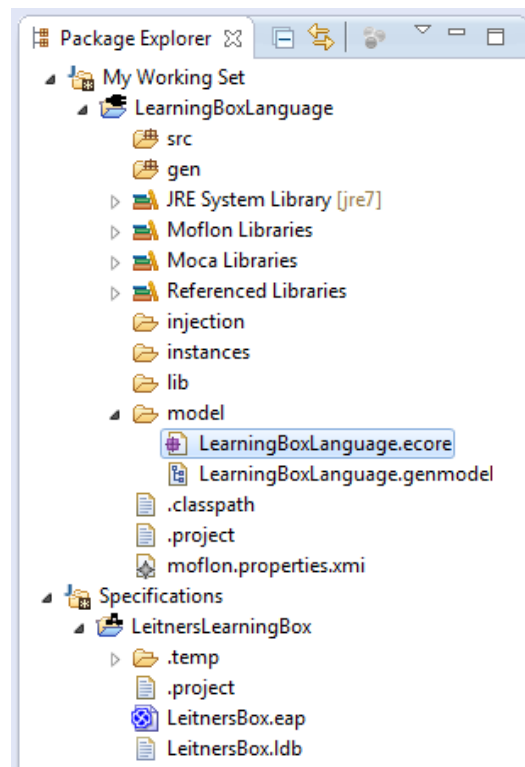
Figure 9: Initial export to Eclipse

▶ 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 Getting started with MOSL

> Please note that the textual syntax is not as thoroughly tested as the visual syntax because most of our projects are built with the visual syntax. This means: Whenever something goes wrong even though you are sure to have followed the instructions precisely, do not hesitate to contact us via contact@emoflon.org.

▶ Create a new metamodel project in Eclipse by navigating to the `New Metamodel` button in the toolbar. In the dialogue that appears, enter `LeitnersLearningBox` as the project name, and select `Textual (MOSL)` (Fig. 10).
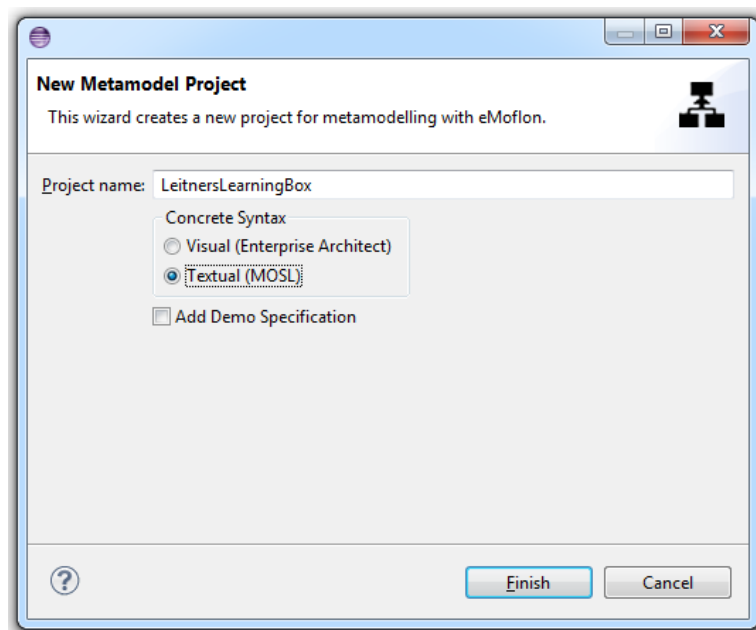

Figure 10: Creating a new metamodel project

▶ You'll see your new project appear under the "Specifications" node.[4] If you're interested in the details of eMoflon's project structure, review Section 4.2 from Part I. Otherwise, expand the project as deep as it goes (Fig. 11).

---

[4]If no nodes appear in your package explorer, ensure your "Top Level Elements" are set to "WorkingSets"
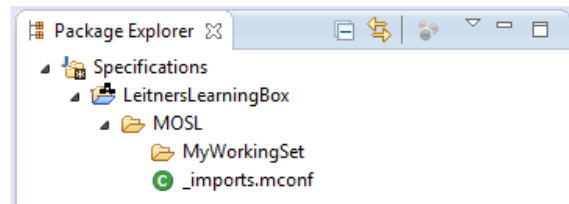
Figure 11: Expanded project files

▶ We're most interested in `MOSL/MyWorkingSet`, which represents the project scope. Right click this folder, and create a new `EPackage` (Fig. 12), naming it `LearningBoxLanguage`. Using this wizard to generate your EPackage will automatically create the necessary `_patterns` folder and `_constraints` file.
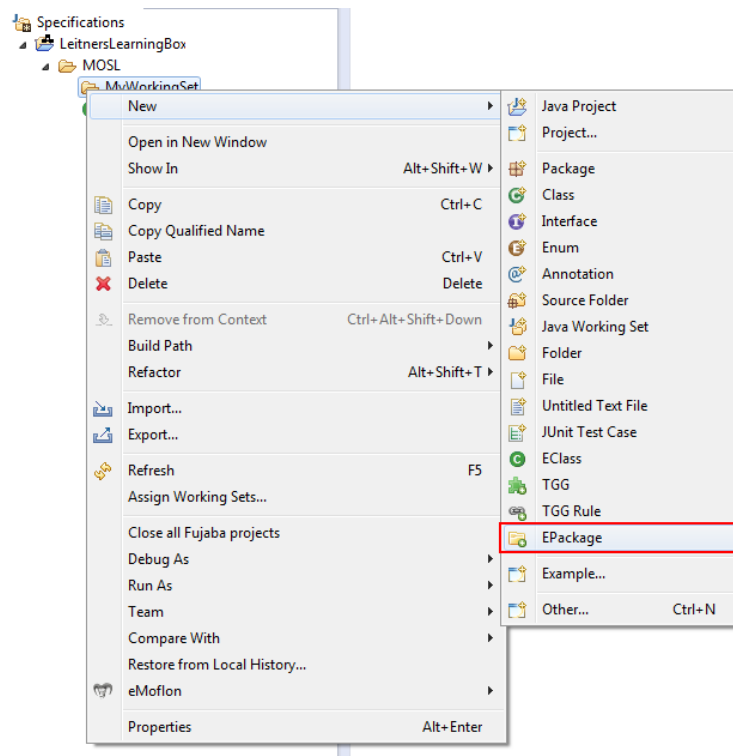

Figure 12: Create an EPackage in your working set

▶ Your package explorer should now resemble Fig. 13.

▶ To finish initalising your metamodel, navigate to "Build (Without cleaning)," found beside "New Metamodel" in the toolbar (Fig. 13).
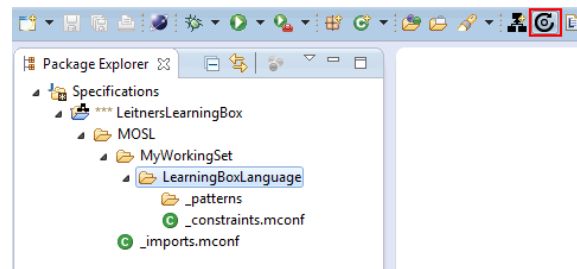

Figure 13: Inital structure of Leitner's Learning Box

▶ A new, "MyWorkingSet" node, named after your project container, should have been created (Fig. 14). The folder "gen" is where all Java files generated from your metamodel will be placed.
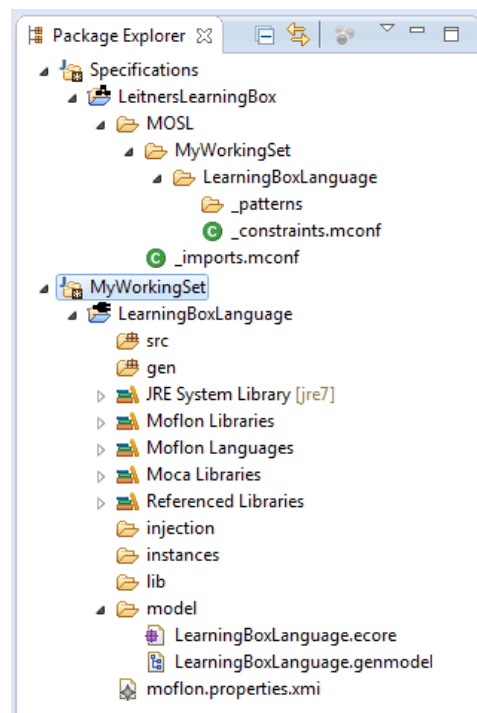

Figure 14: The project fully initialised

▶ Navigate to "LearningBoxLanguage/model." This folder contains `LearningBoxLanguage.ecore`, your metamodel. This will contain all types you define in "LeitnersLearningBox/MOSL/MyWorkingSet/LearningBoxLanguage."

▶ Your project structure is now complete! In the next section, we'll start creating classes and attributes.

2.3   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 (Fig. 15).[5] 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`.
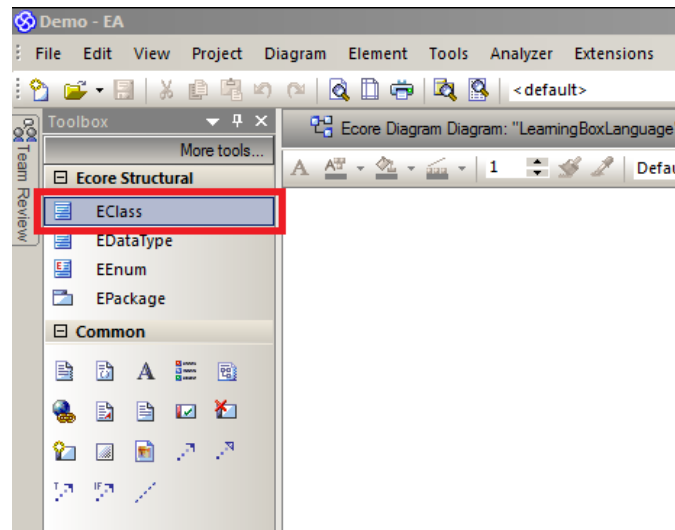

Figure 15: Create an EClass

▶ In the dialogue that pops-up, set `Box` as the name and click `OK` (Fig. 16). 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.

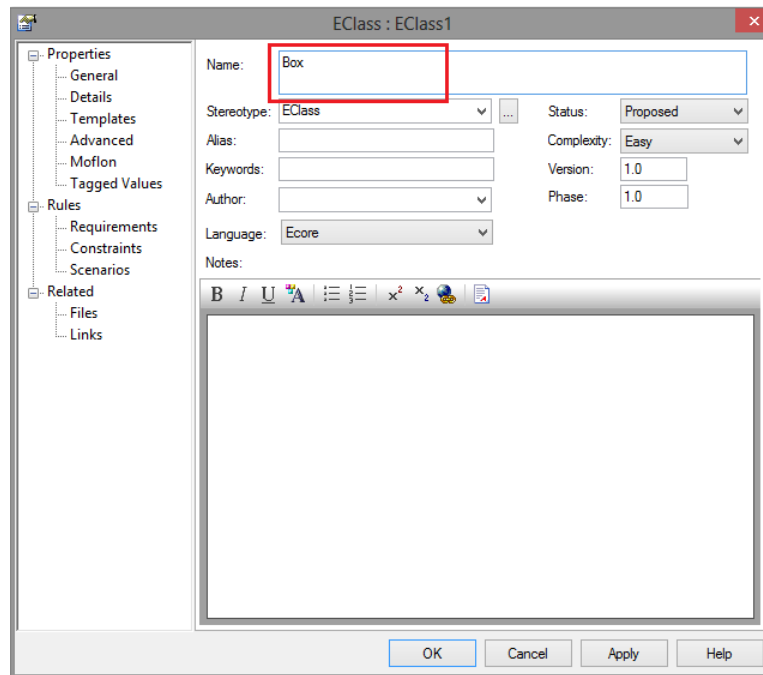[5]If not, choose "Diagram/Diagram Toolbox" to show the current toolbox (Alt+ 5)

Figure 16: Edit the properties of an EClass

▶ After creating `Box`, your EA workspace should resemble Fig. 17.
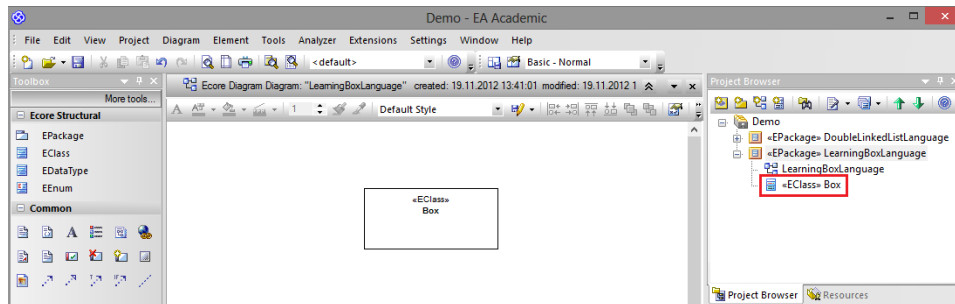

Figure 17: State after creating `Box`

▶ Now create the `Partition` and `Card` EClasses the same way, until your workspace resembles Fig. 18. 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.." (Fig. 19), or press `F9` to open the editing dialogue.
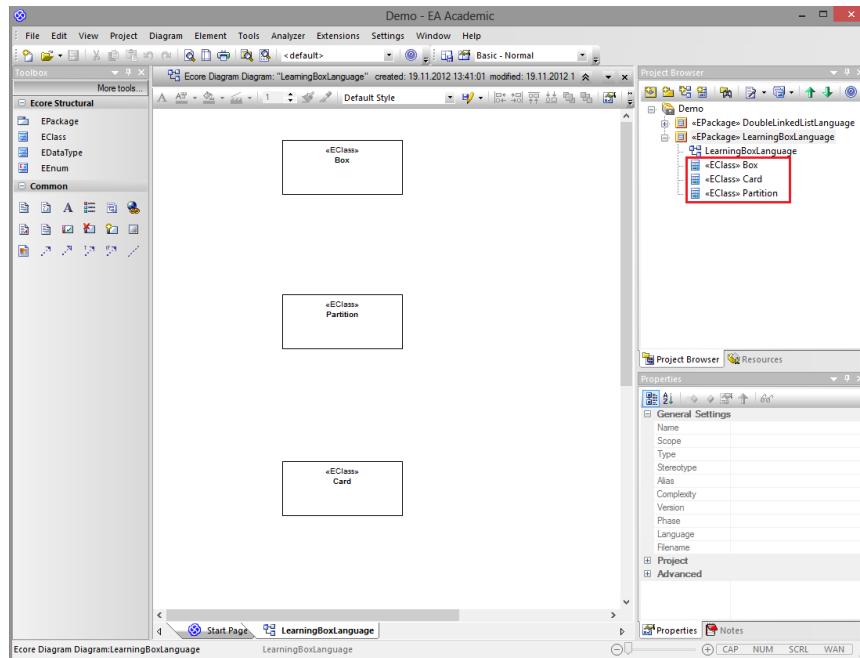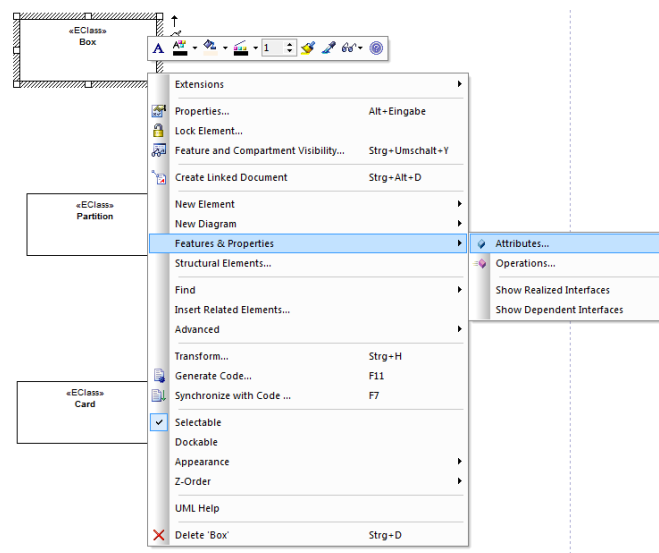
Figure 18: All EClasses for the metamodel

Figure 19: 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 `Save` (Fig. 20). New attributes for the same EClass can be added by pressing `New`.
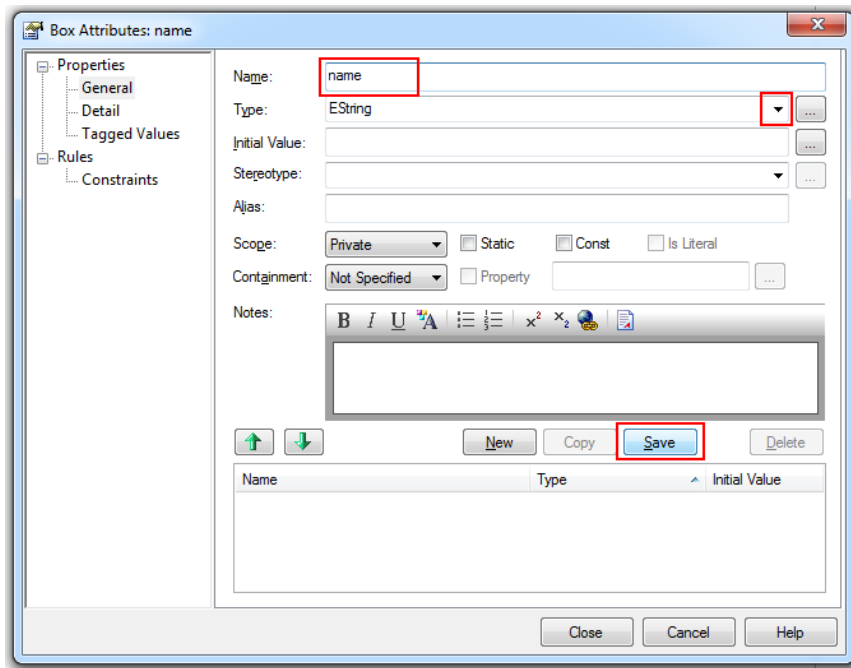

Figure 20: Adding attributes to an EClass

▶ Add the remaining attributes analogously to each EClass until your workspace resembles Fig. 21.

▶ 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!

«EClass»
Box

- name :EString
- stringRep :EString

«EClass»
Partition

- index :EInt
- partitionSize :EInt

«EClass»
Card

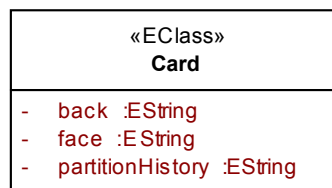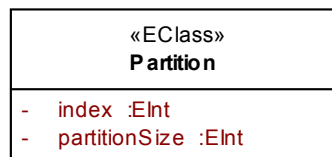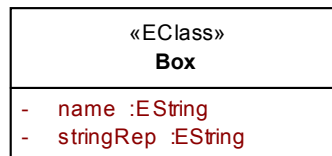| |
|---|
| - back :EString |
| - face :EString |
| - partitionHistory :EString |

Figure 21: Main EClasses declared with their attributes

2.4 Declaring classes and attributes

▶ Right click your `LearningBoxLanguage` model and create your first EClass by navigating to "New/EClass." Name it `Box`.

▶ The class editor should automatically open. Let's add the first two EAttributes of our `Box`, `name` and `stringRep`. eMoflon offers auto-completion templates to help you with this task. Go to an empty line and press `Ctrl + Space`. You'll be provided with a short list of suggestions (Fig. 22). The first four items are related to method control flow, so select `attribute` near the bottom to create `name` of type `EString`.

Figure 22: eMoflon's auto-completion

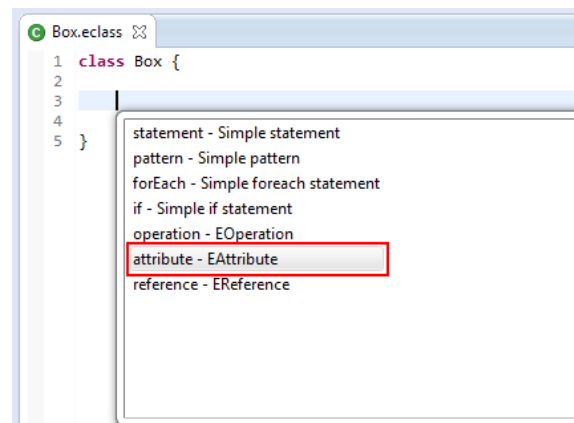▶ Auto-completion also supports you by suggesting a list of types. Start to create a second attribute, `stringRep`, but stop after typing the ':' operator and press the hotkeys. The pop-up list provides a list of all types currently available (Fig. 23) in both your metamodel, and eMoflon's standard metamodels that are included in every new project.

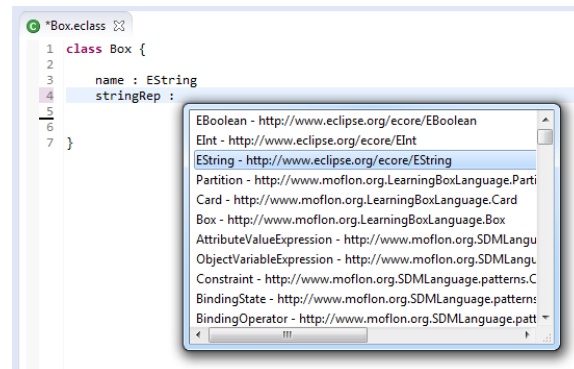▶ Your workspace should now resemble (Fig. 24).

Figure 23: Type suggestions for `stringRep`
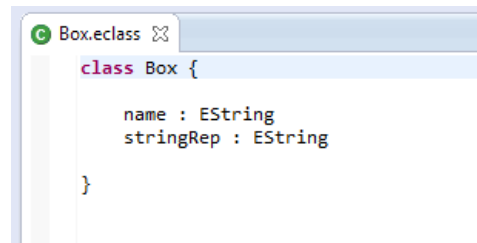
Figure 24: Newly created `Box` EClass

▶ Now create two empty EClasses in your model, `Partition` and `Card`.

▶ In `Partition`, add two `EInt` attributes, `index` and `partitionSize`.

▶ In `Card`, create three `EString` attributes, `back`, `face` , and `partitionHistory`.

▶ If you've done everything correctly, your workspace should now resemble Fig. 25.

▶ That's it for declaring class attributes! Feel free to build your project again and view the changes in the `.ecore` mode, and the generated files in "gen" and "src." On a final note, while some languages (such as Java) allow the declaration of several small classes (such as these three) in the same file, when tooling with eMoflon, we keep them separated. Don't worry – we'll explain this later in the handbook. As for now, continue to the next section to start creating references between these EClasses.

Figure 25: Declaration of all EClasses and attributes
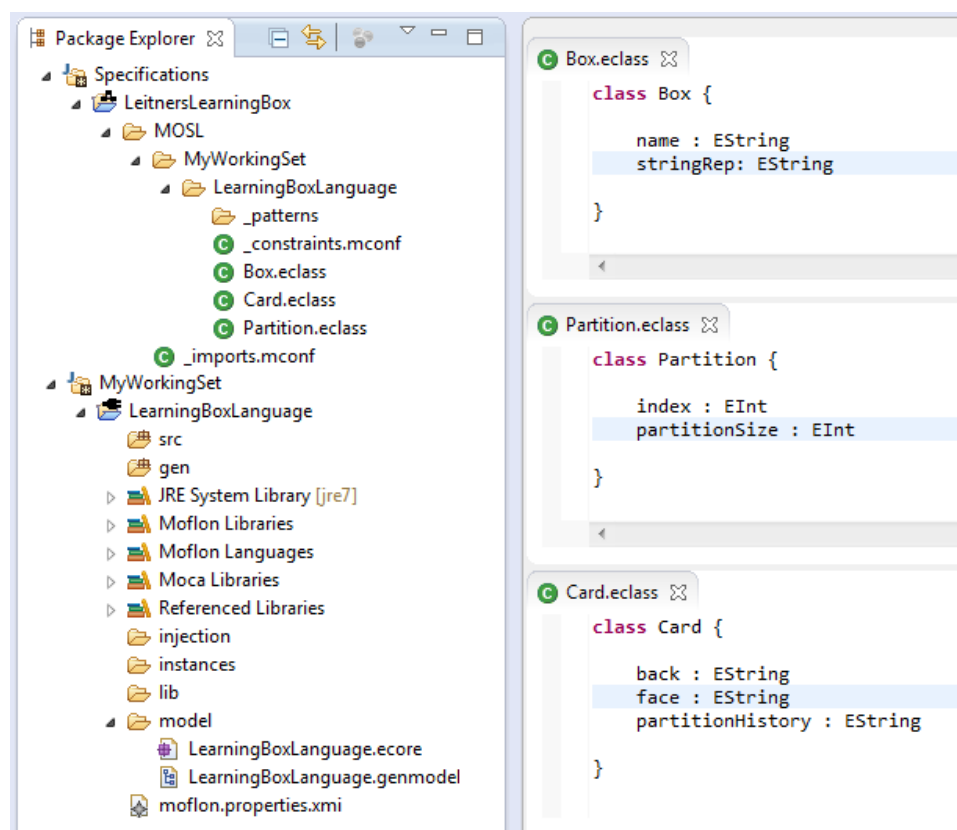
2.5   Connecting your classes

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   *Role*
source and *target* roles determine the direction of the connection.

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

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

The *Aggregration* values of a reference can be either none or composite.   *Aggregation*
Composite means that the current role is that of a *container* for the op-   *Container*
posite role. You'll see in our example that Box is a container for several Partitions. This has a series of consequences: (1) every element must have a container, (2) an element cannot be in more than one container at the same time, and (3) a container's contents are deleted together with the container. 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').

Creating EReferences in EA

▶ A fundamental gesture in EA is *Quick Link*. Quick Link is used to   *Quick Link*
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 (Fig. 26).
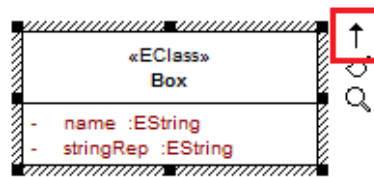
Figure 26: Quick Link is a central gesture in EA

▶ 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" (Fig. 27).

Figure 27: 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.

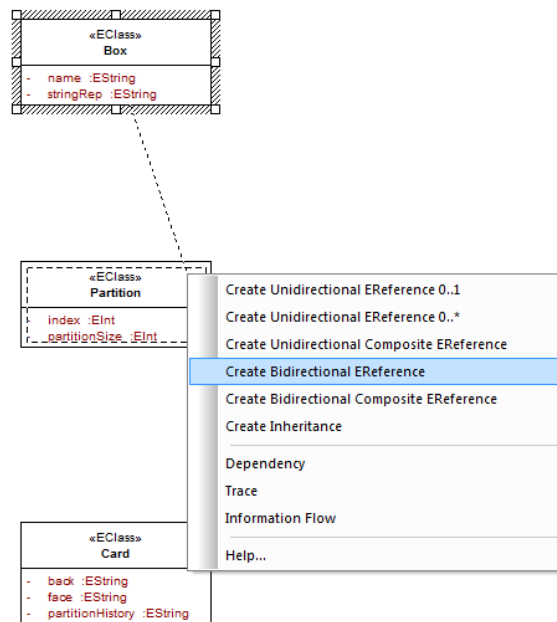▶ Within this dialogue, go to "Source Role," and compare the relevant values in Fig. 28 for the *source* end of the EReference (the `Box` role). As you can see, the default source is set to the EClass you linked from, while the default target is the EClass you linked to. In this window, do not forget to confirm and modify the `Role`, `Navigability`, `Multiplicity`, and `Aggregation` settings for the source as required. Repeat the process for the `Target Role` (Fig. 29).
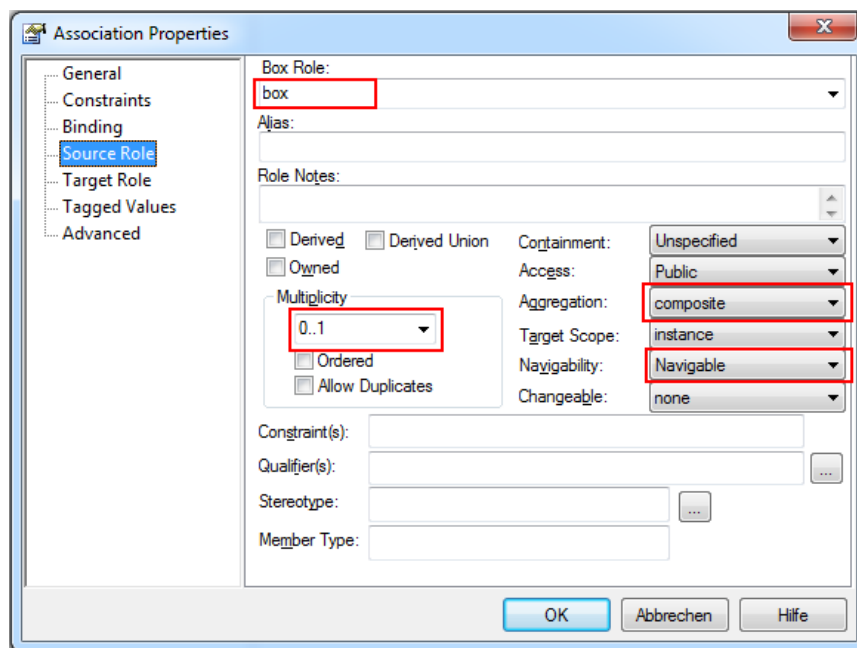

Figure 28: Properties for the source role of an EReference

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 30 sketches this schematically.
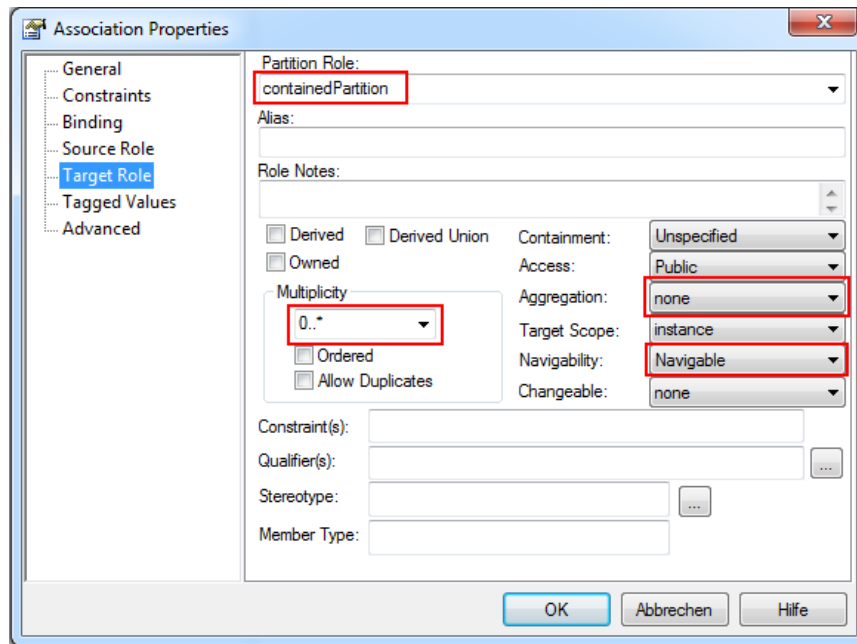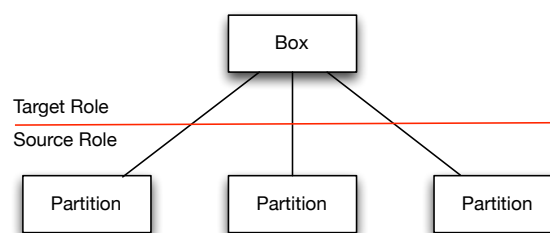
Figure 29: Properties for the target role of an EReference

Figure 30: 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 `Aggregation` settings extend the `Multiplicity` rules. If you've done everything right, your metamodel should now resemble Figure 31, with a single *bidirectional EReference* between `Box` and `Partition`.
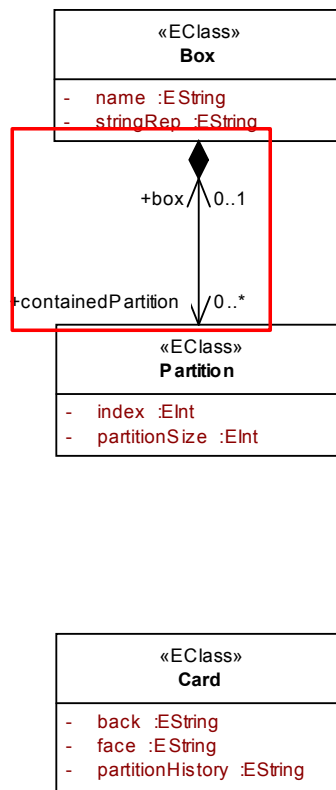
Figure 31: `Box` contains `Partitions`

▶ Following the same process, create two unidirectional self-EReferences for `Partition` (for next and previous `Partitions`), and a second bidirectional composite EReference[6] between `Partition` and `Card` (Fig. 32).

---

[6]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.
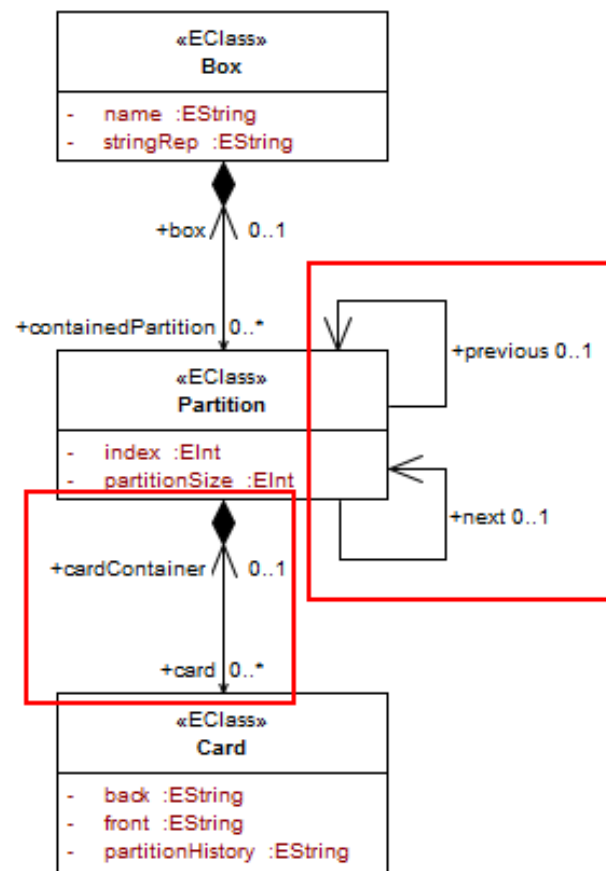
Figure 32: All relations in our metamodel

▶ 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 33.
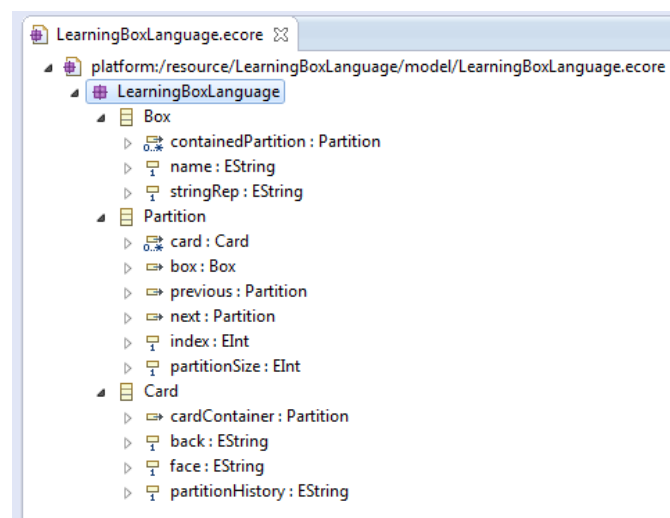
Figure 33: Refreshed Ecore file with all EReferences

▶ All the required attributes and references for your learning box have now been set up. We encourage you to see how these are declared in the textual syntax, starting on the immediate next page. In particular, check out Fig. 37, where each EClass is fully declared, and Fig. 38, where bidirectionality is explicitly specified as a constraint.

Creating EReferences with MOSL

In MOSL, the declaration of an EReference is simple - you set each property according to the following syntax (specified in simple EBNF, if you know what that is):

```
[ '<>' ] '->' role_name '(' multiplicity ')' ':'  target_type

With:
role_name := STRING
multiplicity := '0..1' | '0..*' | '1' | ...
target_type := STRING
```

The source type is determined by the EClass in which the EReference is placed. You can signal an aggregation EReference by including the sideways diamond before the arrow symbol. Don't worry - you don't have to remember this syntax. Our type completion provides a `reference` template when you activate the hot keys. Try it out!

▶ Open `Box.eclass` in the editor and add a *container reference* named `containedPartition` with a multiplicity of zero to infinity, from `Box` to `Partition` (Fig. 34). This EReference means a `Box` can hold an infinite number of partitions.
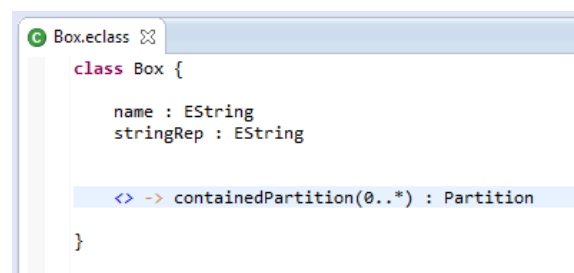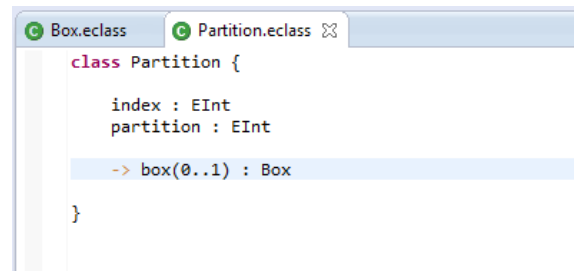
Figure 34: Creating a *container reference* in `Box`
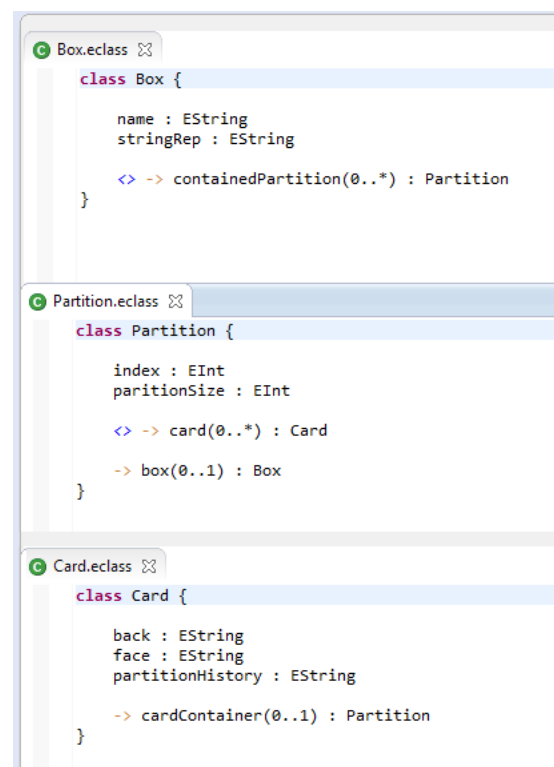
▶ Now add a *simple reference* to `Partition`. Name it `box`, and allow it to hold up to one `Box` (Fig. 35). This means a single partition can belong to either zero, or one `Box`, and that's it. It can't belong to two different boxes at the same time.

▶ Congratulations, you have just built your first pair of EReferences! To see how this is depicted visually, check out Fig. 31 from the previous subsection.

Figure 35: Creating a *simple reference* in `Partition`

▶ Now, lets create another pair of EReferences between `Partition` and `Card`. If you think about it, it's really not all that different from the relation between `Box` and `Partition`. A `Partition` should be able to hold an unlimited amount of `Card`s, but a `Card` should only be allowed to belong to zero or one `Partition`s. Name the two new EReferences `card`, and `cardContainer`. Your EClasses should now closely resemble Fig. 36.

Figure 36: Completed EReference pairs

▶ The next step is to construct two connections between `Partitions` so cards can be moved between their previous and next partitions in the box. Create two new simple references, named `previous`, and `next`, each with a `0..1` multiplicity.

▶ If you have done everything correctly, your EClasses should now resemble Fig. 37.

```
Box.eclass ⊠
    class Box {

        name : EString
        stringRep : EString

        <> -> containedPartition(0..*) : Partition

    }
```

```
Partition.eclass ⊠
    class Partition {

        index : EInt
        partition : EInt

        <> -> card(0..*) : Card

        -> box(0..1) : Box
        -> next(0..1) : Partition
        -> previous(0..1) : Partition

    }
```

```
Card.eclass ⊠
    class Card {

        back : EString
        face : EString
        partitionHistory : EString

        -> cardContainer(0..1) : Partition

    }
```
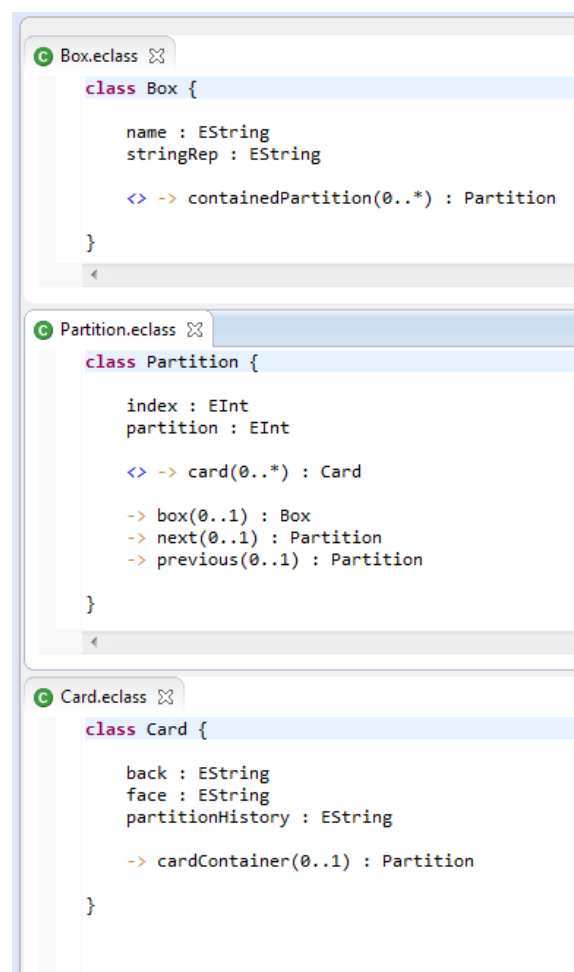
Figure 37: All EReferences for Leitner's learning box

At this point, all of your EReferences have been created! The problem is, suppose you set the `containedPartition` EReference in a particular `Box`. That's great, you would now have the box containing one partition. However, if you went and examined that partition independently, its `box` EReference would still be null. We still need to set up the link between these EReferences so that when one is updated, the other will be too.

► Navigate to the `_constraints.mconf` file. You can see it has a single `opposites` scope that's currently empty. Constraints follow the syntax below:

```
reference '<->' reference

With:
reference := reference_name ':'  source_type
reference_name := STRING
source_type := STRING
```
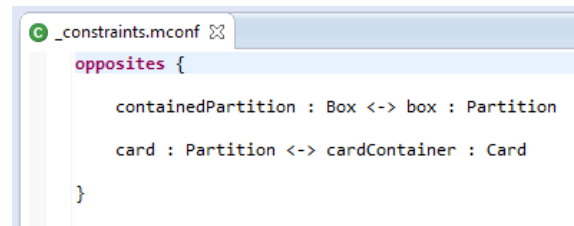
This statement sets the two EReferences to be opposites of one another, i.e., the connection between EClasses will be bidirectional. As you can see, syntax here is slightly different than that of a standard EReference. Instead of the reference type trailing the colon operator, it has switched to become the source type.

► To begin, press `Ctrl + Space` and complete the template with the following:

```
containedPartition :  Box <-> box :  Partition
```

► Reviewing the `Partition` EClass, its easy to see that `previous` and `next` are certainly not opposites,[7] but we do need to establish an opposing link between `card` and its `cardContainer`. Follow the same steps until your constraint file resembles Fig. 38.

---

[7]Review the rules depicted in Fig. 1

Figure 38: The completed constraints file

▶ Now the EReferences for your learning box are complete! To see how
each of the classes, attributes, and references are depicted in the visual
syntax, check out Fig. 32 from section 2.1. Otherwise, build your
project to make sure there are no errors, and continue to the next
section to finalize the declaration of your EClasses.

2.6   Method Signatures

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

▶ Select `Partition` and either right-click to invoke the context-menu (Fig. 39) and choose "Features & Properties/Operations.." or simply press `F10`.
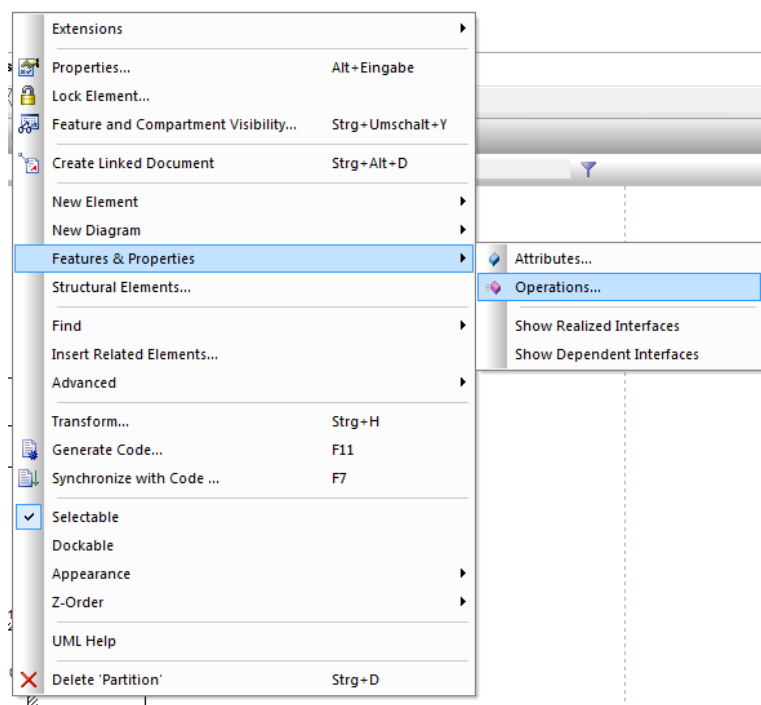
Figure 39: Add an operation

▶ In the dialogue that pops-up (Fig. 40), enter `empty` as the `Name` of the operation, leave the `Return Type` as `void`, and press `Save`.

▶ In the same dialogue, press `New` to add a second operation, `removeCard`, and edit the values as seen in Figure 41. Notice that the `Return Type` can be chosen by either the drop-down menu for primitives (e.g. `EBoolean`), or via the '`...`' button (highlighted in Figure 40) for types you've established in the metamodel (e.g. `Card`).

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

Figure 40: EClass properties editor

▶ Parameters can be added by pressing `Edit`[8] and completing the dialouge. Please remember that you must also use either the drop-down menu, or the '...' button to select the type or else validation will fail.

▶ Repeat this process for the `check` operation (with the two parameters `card:Card, guessed:EString`) that returns an `EBoolean` (Fig. 41).
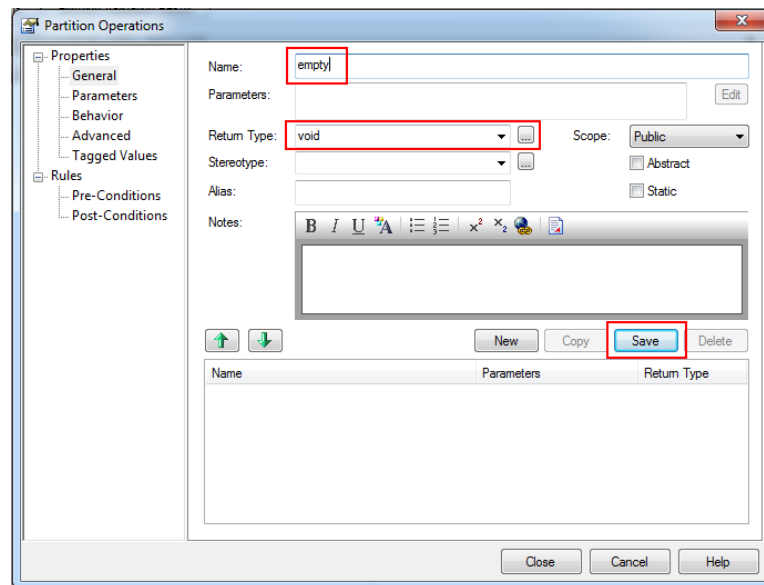
▶ If you've done everything right, your dialogue should now contain three methods - `check`, `empty`, and `removeCard` - each with the corresponding parameters and return types in Fig. 42.

▶ Add all operations analogously for `Box` and `Card` until your metamodel closely resembles Figure 43.[9]

▶ 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.

▶ To see how this complete metamodel is represented in the textual syntax, examine Fig. 45 in the following section.

---

[8]You must save the operation before this option will become active
[9]Please note that names of parameters may not be displayed by default in EA
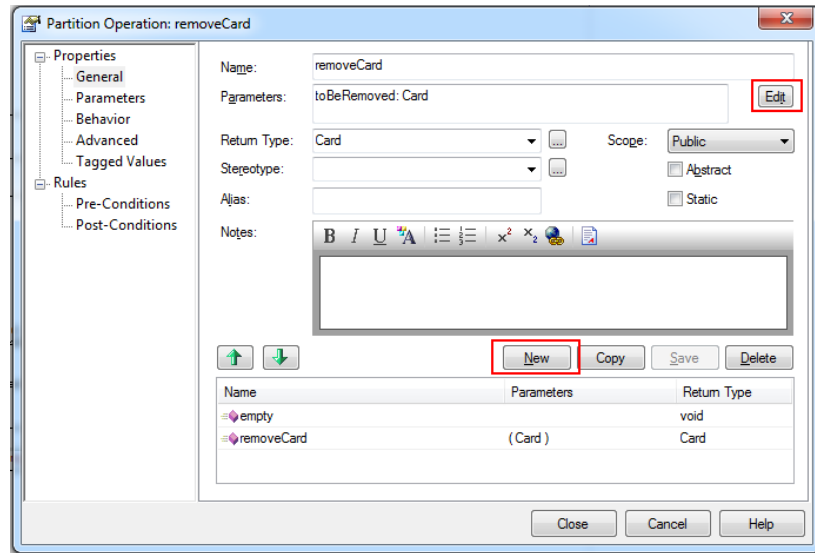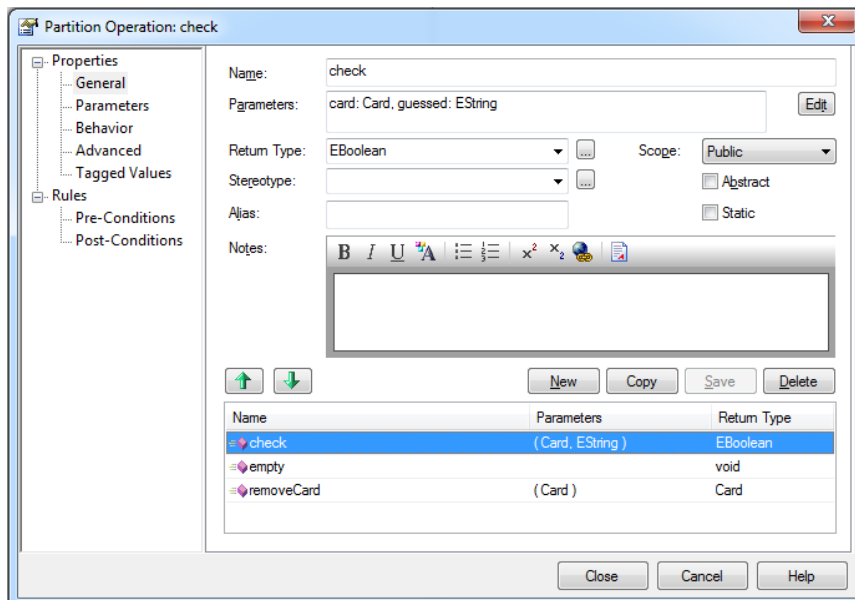
Figure 41: Parameters and return type

Figure 42: All operations in `Partition`
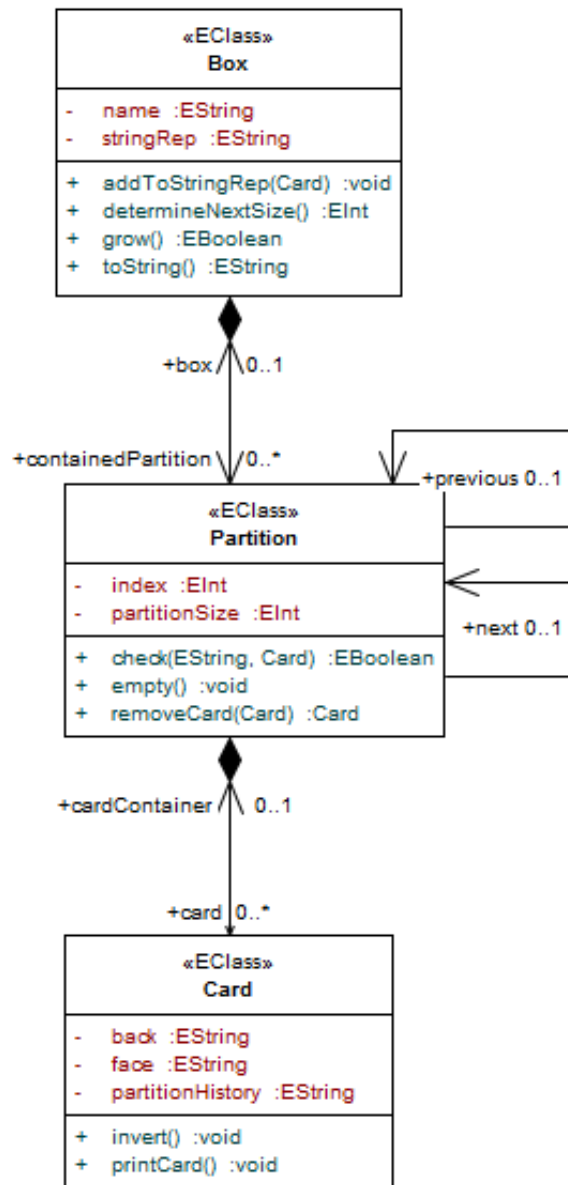
Figure 43: Complete metamodel for our learning box

2.7   Method Signatures

▶ We're nearing the end of our model creation! One of the last things we need to do is to make the model *do* something. After all, a model that only stores attributes and references is a bit boring, right?

▶ Let's set up the operations we want each EClass to do by declaring their *signatures* which follow the syntax below:

```
name '(' argument* ')' ':'  return_type

With:
argument := param_name :  param_type
name, return_type, param_name, param_type := STRING
```
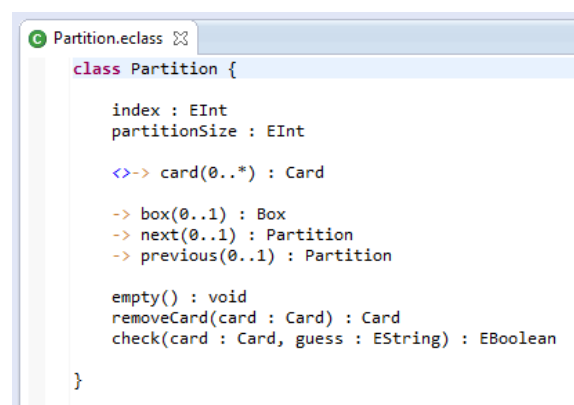
▶ Starting with the `Partition` EClass, we want a partition to be able to do three things: compare the answer on a `Card` with a guess and return a true/false response, remove a specific card from the partition, or empty itself of all cards.

▶ Start with the `empty` method. It won't need any parameters, and it doesn't return anything. Declare this via:

```
empty() :  void
```

▶ Create two more functions for `Partition` the same way. We'll need a `removeCard` method that accepts and returns a `Card`, as well as a EBoolean `check` method that accepts a `Card` and an `EString` guess.

▶ Your `Partition` EClass should now resemble Fig. 44.

Figure 44: The completed `Partition` EClass

▶ What needs to be done in the `Card` EClass? Well, in order to check the card, we'll need to be able to look at the flip side. We'll also need to print whatever is on the current side. Create two paramater-less `void` functions, `invert` and `printCard`.

▶ Finally, what do we want to do with `Box`? In summary, we want a `Box` to:

`determineNextSize():EInt` Calculate how large a new partition in the box should be

`grow():EBoolean` Increase the box by adding a new partition

`toString():EString` Produce a string representation of the box with all its contents

`addToStringRep(card:Card):void` Update the current string representation to include `card`

▶ Implement the above signatures, and your entire workspace should now resemble Fig. 45.

▶ Congratulations! You have now created a metamodel for our Learning Box using eMoflon's textual syntax! To see how this looks in the visual syntax, check out Fig. 43 from the previous section. As a final step, make sure you build the project and wait for the package explorer to refresh.
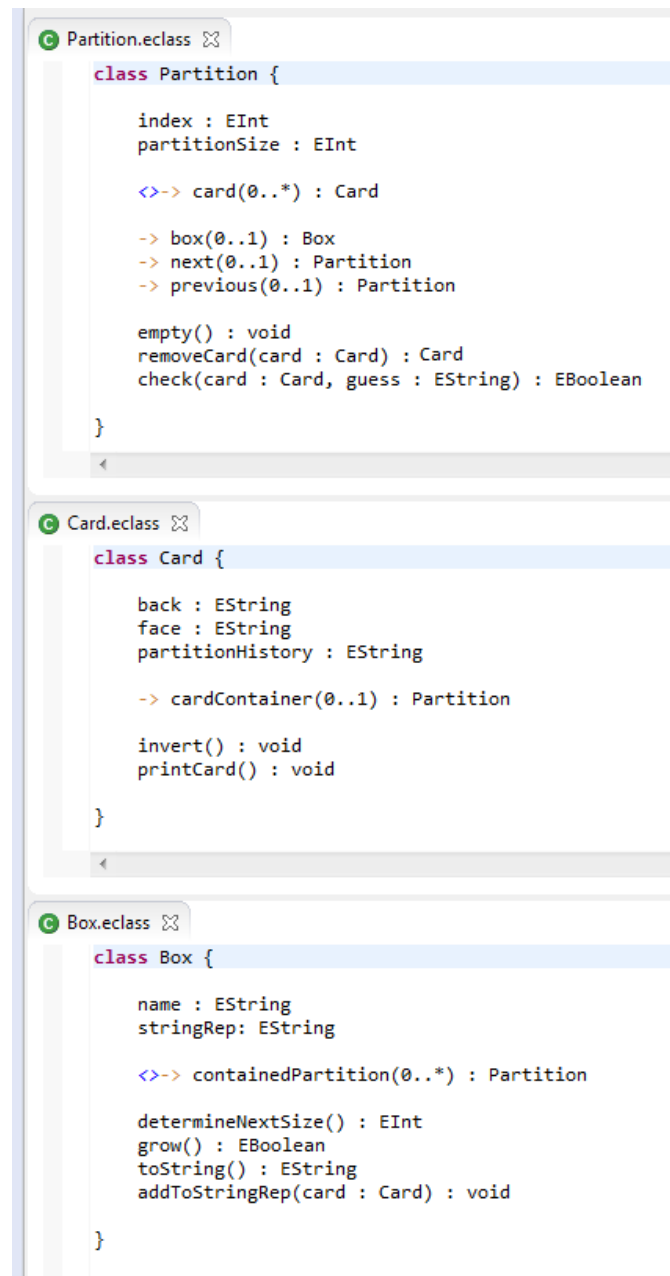
Figure 45: Completed method signatures

2.8   eMoflon validation support in EA

Our EA extension provides rudimentary support for validating your meta-model. 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 46. 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 46: Activating the validation output window

▶ To start the validation, choose "Validate all" in the "Validate" section of the control panel (Fig. 47). 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.
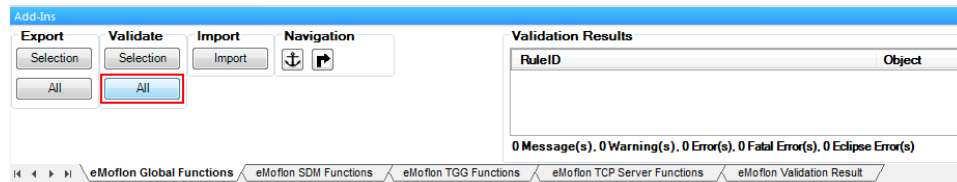
Figure 47: Starting the validation

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`.

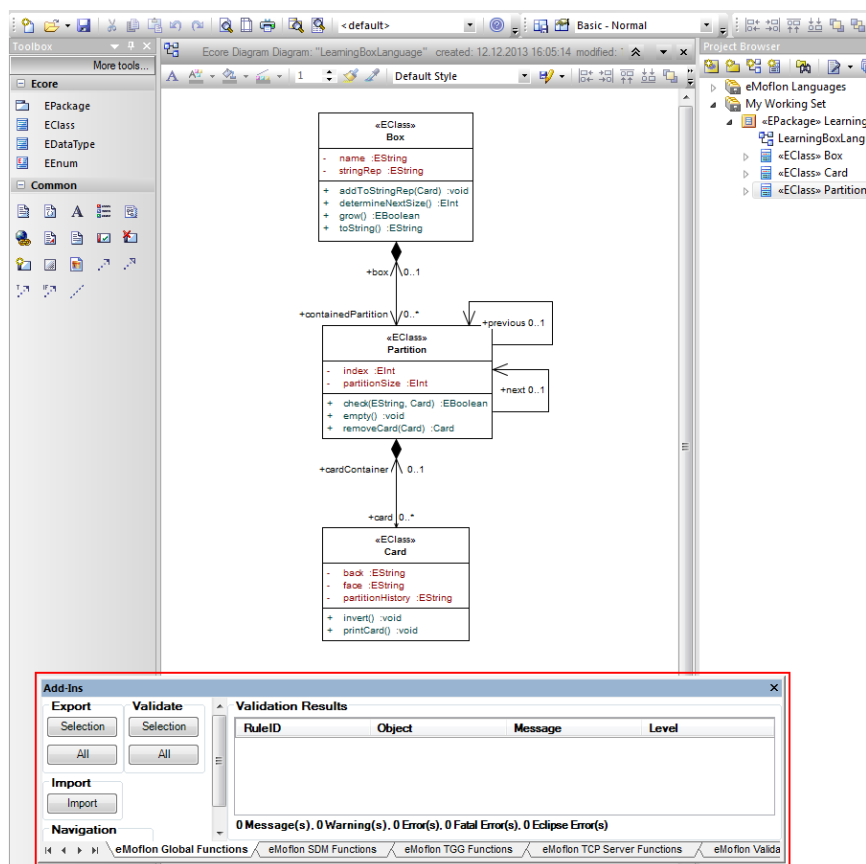▶ Create a new EClass in the `LearningBoxLanguage` diagram. You can retain the default name `EClass1`. Let's assume you wish to delete this class from your metamodel.

▶ 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 (Fig. 48).

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![10]

▶ 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.

---

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

Figure 48: Validation information error: element still exists

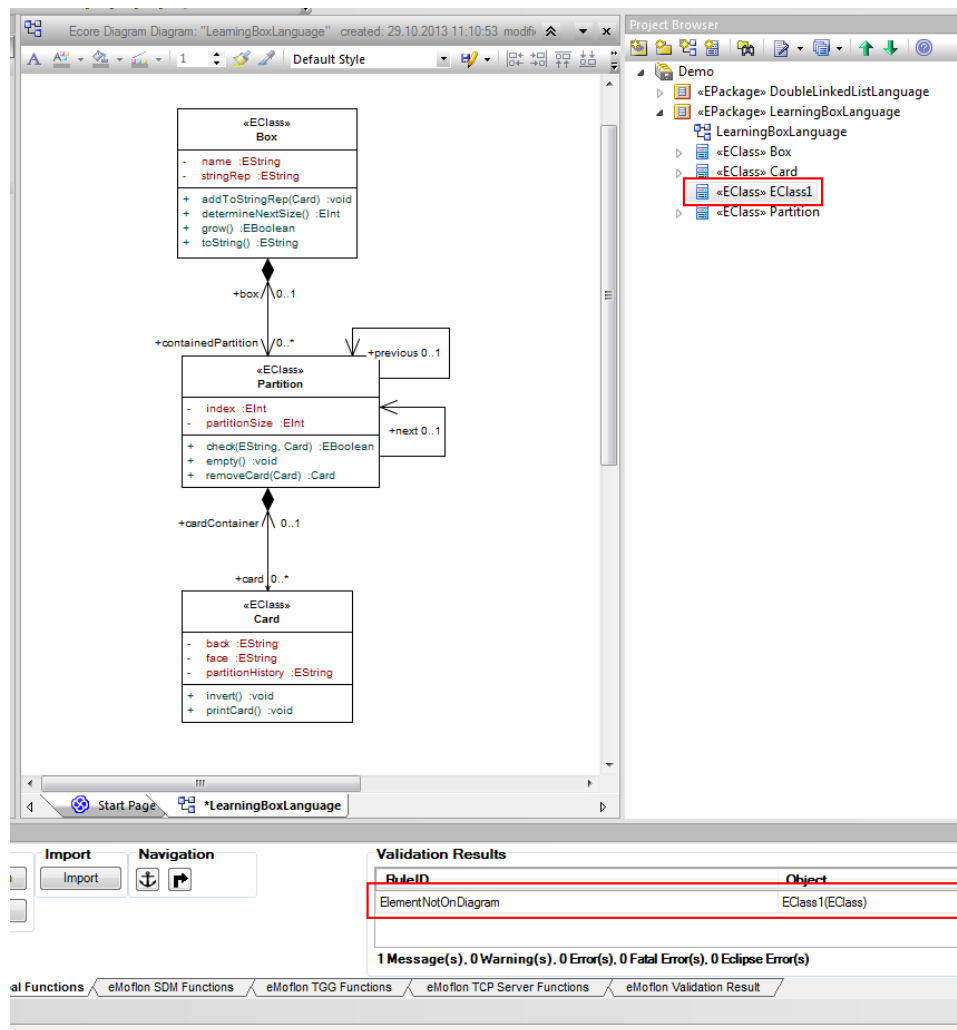► To check to see if there are any quick fixes available, *double*-click the information message to invoke the "QuickFix" dialogue. In this case, there are two potential solutions - add the element to the current diagram or (properly) delete the element from the metamodel (Fig. 49). Since the latter was the original intent, click `Ok`.
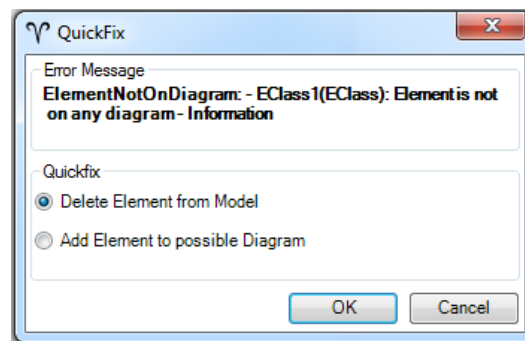
Figure 49: Quick fix for elements that are not on any diagram

► `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 Fig. 50. 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 (Fig. 51). 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`.

► Ensure that your metamodel closely resembles Figure 43 again, and that there are no error messages before proceeding.
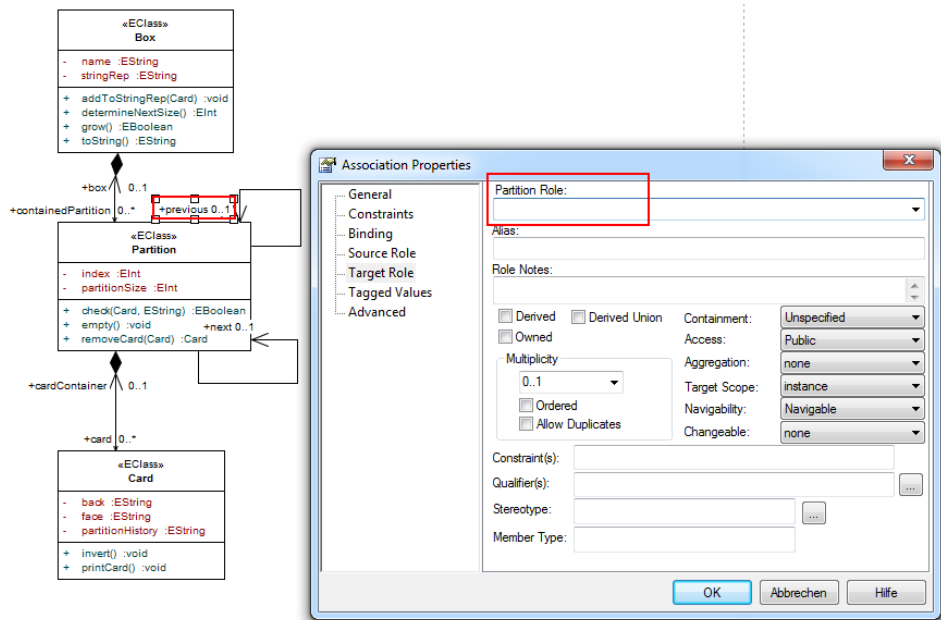
Figure 50: Deleting a navigable role name of an EReference
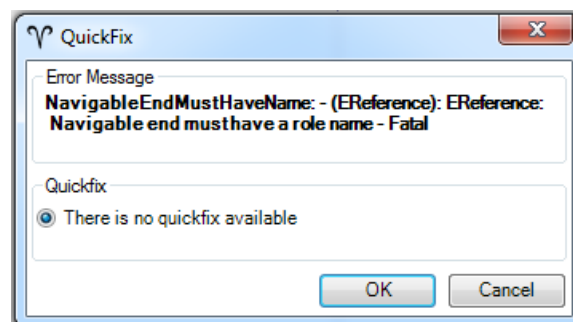
Figure 51: Fatal error after deleting a navigable role name

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.

2.9   eMoflon support with the MOSL Builder

Our MOSL language is accompanied with its own builder that proves support for validating your metamodel. Integrated with the Eclipse IDE, if there's an error in your files when you save, a message will appear in the console. Lets try to make an error, just to see how this works.

Go to your `Partition` EClass and change the parameter type in `removeCard` from `Card` to `card`. Press save. An error should immediately appear below the editor to inform you of your terrible mistake. Change your file back to the way it was, and the message should disappear.

In addition to validation on saving files, MOSL also lets you know whether or not your project has recently been built (if code has been generated from the metamodels). You may have noticed this feature in the previous sections. Similar to how Eclipse informs you that your file has changed by placing a `*` beside the name in the editor tab, MOSL places a `***` symbol beside your metamodel folder title in the package explorer to remind you to build your project.

2.10   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 `Partition`s. A `Partition` in the `Box` has a `next` and `previous Partition` that can be set. Finally, `Partition`s contain `Card`s.

A `Box` has a `name`, and can be extended by calling `grow`. A `Box` can print out its contents via 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 `Card`s. 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`. `Card`s also maintain a `partition-History`, 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.

On a final note, we encourage you to review how the metamodel was constructed in the alternative syntax to the one you primarily used. Compare the differences between modelling classes and references visually using a separate program (Enterprise Architect), then exporting to Eclipse to build, versus modelling textually entirely within Eclipse. Which do you find easier to work with?

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 Fig. 52.

- ▶ 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 (Fig. 53).

- ▶ 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).
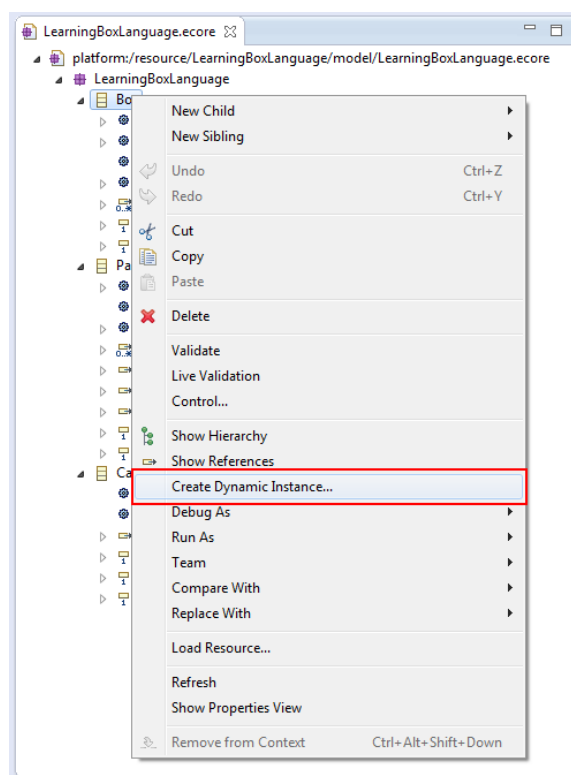
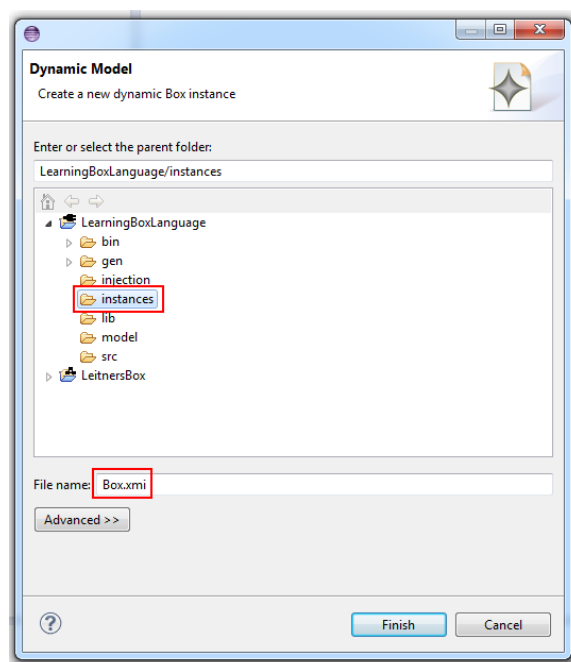Figure 52: Context menu of an EClass in the Ecore editor

Figure 53: 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 54. Note that EMF supports you by respecting your metamodel, and reducing the choice of available elements to valid types only.[11]
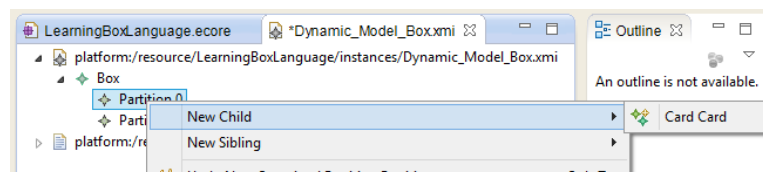

Figure 54: Context menu for creating model elements

► 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 (Fig. 55). Here you'll see the attributes you defined earlier in each EClass. The `Box`, `Next`, and `Previous` values are its (undefined) EReferences,[12] 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.
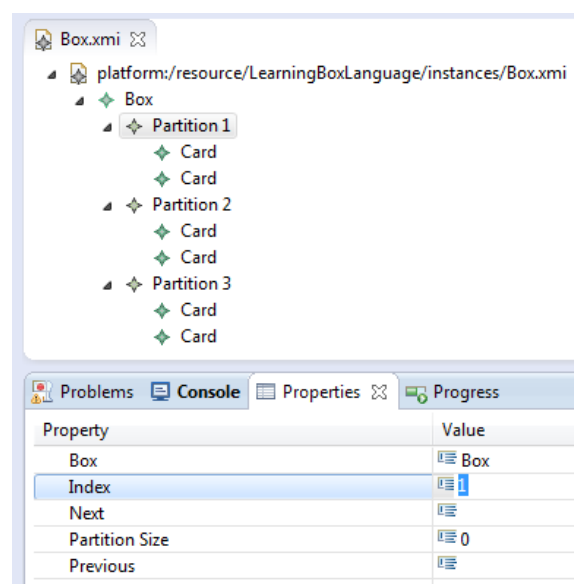

Figure 55: Change the `Index` value in the partition's property tab

---

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

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

▶ 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 (Fig. 56). You now have a unique, customized learning box! Save your model, and ensure no errors exist before proceeding.
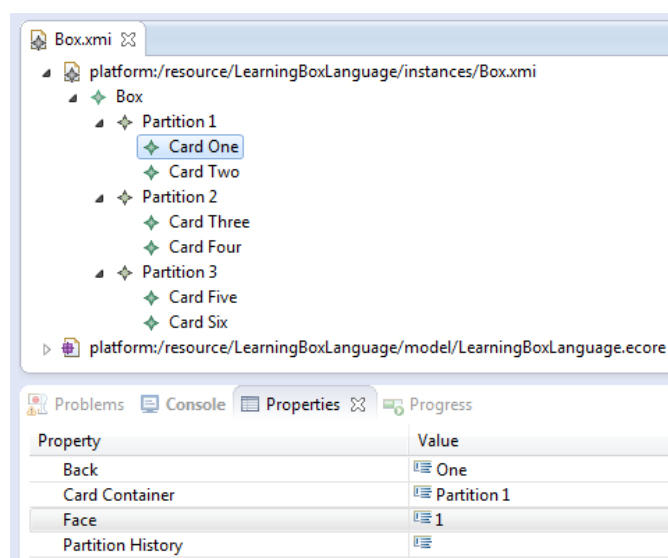
Figure 56: A vocabulary-style learning box

...

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." [13]

▶ 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, visualised with all its immediate children (Fig. 57).
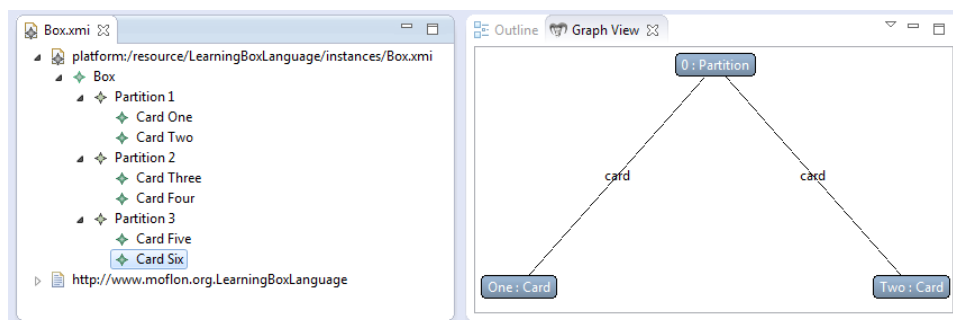
Figure 57: A single partition in eMoflon's graph view

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

---

[13]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"

▶ 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" (Fig. 57).

▶ 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. Your entire metamodel is now completely displayed in the viewer (Fig. 58). 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.
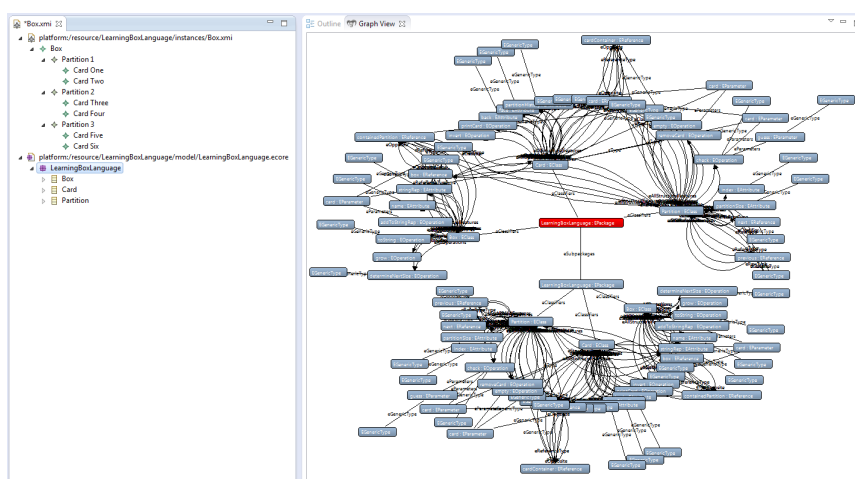

Figure 58: Our instance's complete type graph

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 358. 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.

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

Figure 59: 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 (Fig. 60).

▶ 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 (Fig. 61).

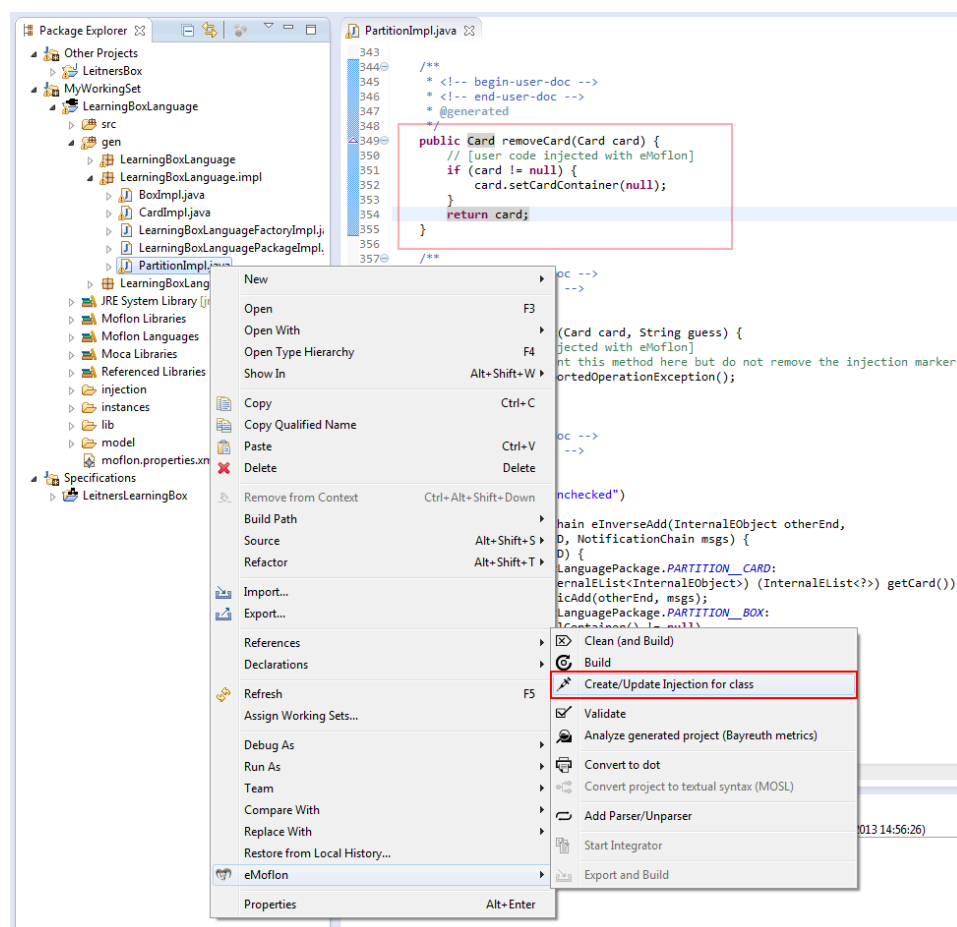▶ Double click to open and view this file. It contains the definition of a *partial class* (Fig. 62).
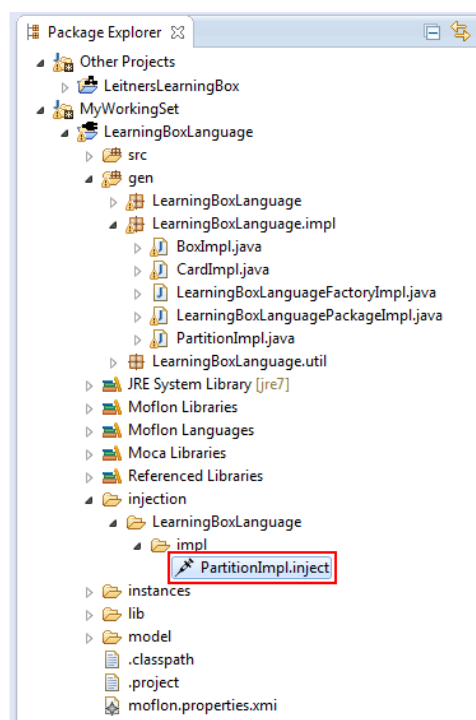
Figure 60: Create a new injection

Figure 61: Partition injection file

Figure 62: Generated injection file for `PartitionImpl.java`

▶ As a final step, clean and 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.

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 top left of your toolbar and start the "New" wizard.

▶ Load "Examples/eMoflon Handbook Examples/Part II GUI Download" (Fig. 63). This will load the new project into your workspace.[14] Right click `LeitnersBox` to invoke the context menu and select "Run as/Java application."
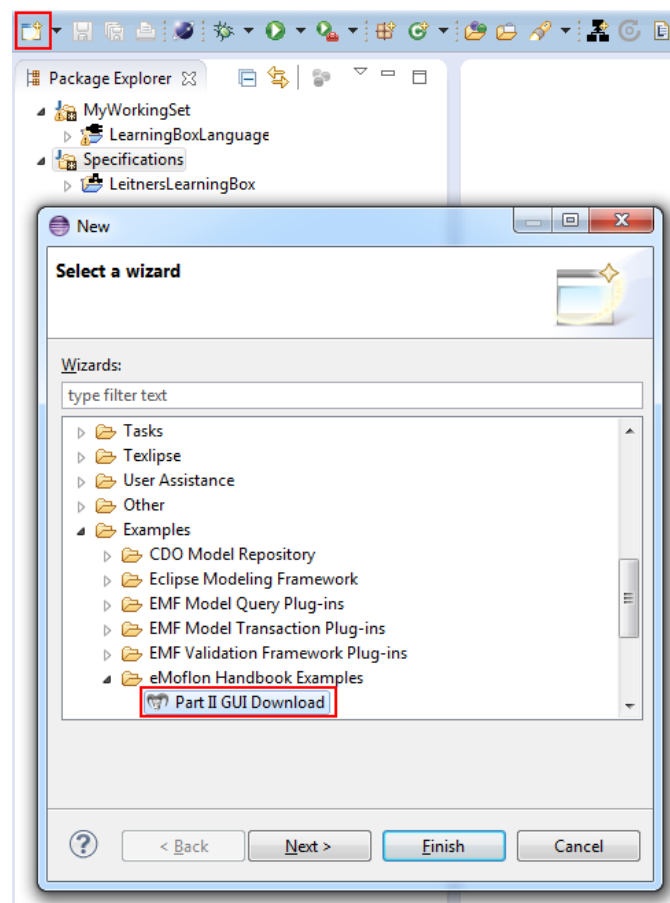
Figure 63: Load Leitner's Box GUI

---

[14]If nothing appears, go to the small arrow to the right of the window, and select "Configure Working Sets..." Make sure `Other Projects` is selected, and press `Ok`.

▶ 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" (Fig 64). While we'll implement "Check Card" in Part III, "Remove Card" is currently active, implemented by the injection you just created.
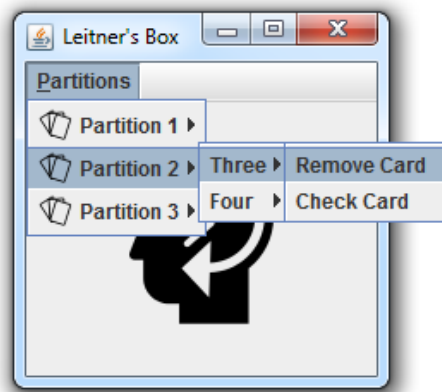
Figure 64: Using the GUI with your instance

▶ 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! Both the visual and textual specifications are 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[15]! 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 some of the other features eMoflon has to offer. Check out Triple Graph Grammars (TGGs) in Part IV[16], or Model-to-Text Transformations in Part V[17]. 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[18].

Cheers!

---

[15]Download: `http://tiny.cc/emoflon-rel-handbook/part3.pdf`

[16]Download: `http://tiny.cc/emoflon-rel-handbook/part4.pdf`

[17]Download: `http://tiny.cc/emoflon-rel-handbook/part5.pdf`

[18]Download: `http://tiny.cc/emoflon-rel-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.