TUHH
Technische Universität Hamburg-Harburg

STS

Generation of Diagram editors, taking the Enterprise Application Integration Patterns as Case study

Master Thesis

Submitted by:

Madanagopal Doraiswamy Venkatesan
madanagopal.doraiswamy @tu-harburg.de
Information and Media Technologies
Matriculation Number : 27125

Supervised by:

Prof. Dr. Ralf MÖLLER
STS - TUHH

Prof. Dr. -Ing. Erik. PASCHE
Institut für Wasserbau – TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Hamburg, Germany
17th October 2006

Declaration

I declare that:
this work has been prepared by myself,
all literally or content-related quotations from other sources are clearly pointed out,
and no other sources or aids than the ones that are declared are used.

Hamburg, 17 October 2006

Madanagopal Doraiswamy Venkatesan

Acknowledgments

I would like to extend my sincere gratitude to **Prof. Dr. Ralf Möller** for giving me an opportunity to do my Student Project at Software, Technology & Systems (STS) Department of Hamburg University of Technology, Germany.

The major credit for the work in this thesis must go to **M.Sc. Miguel Garcia**. I am gratefully appreciative for his many valuable suggestions, tremendous contributions, always prompt feedback, guidance, encouragement and support during the research. My gratitude is beyond words.

Last but not least, I would like to thank the Software, Technology & Systems (STS) Department for supporting me throughout my program and research.

Finally, I would like to thank my parents and my friends for their continuous support and encouragement.

Table of Contents

List of Figures

1 Introduction

1.1 Motivation

Generating complete code from models has been an industry goal for many years. Models serve as mechanism for better understanding and documentation, but they can also find their purpose in generating complete and working code. This automates development leading to improved productivity, quality and complexity hiding. Many existing modeling languages are based on code and offer only modest possibilities to raise design abstraction and to achieve full code generation. With such modeling languages, the level of abstraction in models and the generated code is the same. As a consequence of this, developers easily find themselves making models to describe functionality and behaviour and end up generating the code. The limited code generation possibilities force developers to start manual programming after design. This leads to having same information in two places, code and models.

In Domain Specific Modeling, the domain elements represent things in the domain world and not in the code world. The modeling language following the domain abstractions and semantics, gives modelers a feel that they perceive themselves as working directly with the domain concepts. The rules of the domain can be included into the language as constraints, making it impossible to specify illegal or unwanted design models. Being free from manual creation and maintenance of source code can significantly improve developer productivity. The reliability of automatic generation compared to manual coding will also reduce the defects in source code, thus improving quality. In software system development, often domain specific visual notations are used for which a tool environment consisting of full fledged visual editors, simulators, model transformers, etc. is needed.

Several Eclipse projects head for meta technology to define domain specific languages. The Eclipse Modeling Framework (EMF) provides a modeling and code generation framework for Eclipse applications based on structured data models called metamodel. The metamodel defines all symbols and relations of the domain specific model which can be used to provide an editor with basic editing commands. EMF generates a set of Java Classes for manipulating the model and a basic, tree based editor for model instances. For complete language description, the generated model has to be extended by additional syntax checks implementing certain well-formedness rules e.g. by Object Constraint Language (OCL). Generating a graphical view can be hand coded with Eclipse Graphical Editor Framework (GEF), offering basic and advanced editor functionalities based on a model-view-controller architecture.

On the other hand, a visual editor can be generated using Graphical Modeling Framework (GMF) project. This aims at providing a fundamental infrastructure and components for developing visual design and modeling surfaces in Eclipse. In essence, GMF will form a generative bridge between EMF and GEF, whereby a diagram definition will be linked to a domain visual language model which serves as input to the generation of visual editor. Figure 1 shows the dependencies present among the frameworks in generating the visual editor.

GMF is a generative approach allowing to add diagramming capabilities to a visual language model expressed in EMF, if a visual diagram editor is desired. In many ways, GMF is an extension to the capabilities of EMF. GMF has OCL support to verify its domain model instance's well-formedness with the help of Eclipse Modeling Framework Technologies (EMFT).

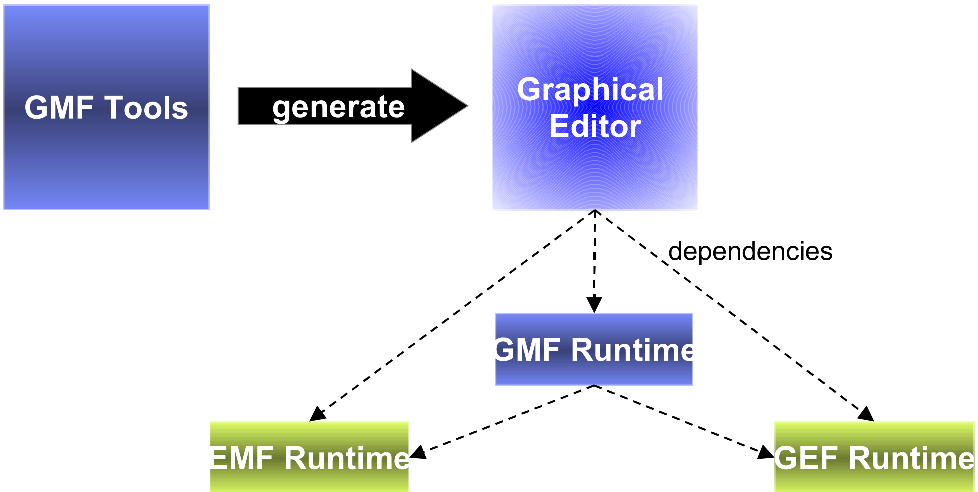


Figure 1: Graphical Modeling Framework - Its Dependencies[4]

1.2 Objective

In this project, we propose an approach to specify OCL invariants as constraints at the domain model level with reference to [1], ideally making it impossible to specify illegal and unwanted design models. This design approach results in including the well-formedness constraints at the domain model that results in a higher level of abstraction.

As a case study for describing the domain model, the Enterprise Application Integration Patterns specified in [2] is considered to create a visual modeling editor for the same.

1.3 Structure of the Work

In the next chapter we will review the state of the art of model-based graphical editor generation. We will discuss the editor generation definitions provided by Eclipse Graphical Modeling Framework. Chapter 3 discusses the language definition used in EMF and GMF for creating domain models and the editor support provided by Emfatic and OCTOPUS [3] for creating the same. Chapter 4 discusses the EMFT technologies supported by GMF and providing the validation support with EMFT OCL for creating visual editor with the domain model integrity maintained at the domain level as specified in [1]. Chapter 5 presents the constraints for Enterprise Application Integration Patterns to check the well-formedness of the domain model instances [2]. Chapter 6 offers the conclusion and future works.

2 Graphical Modeling Framework

The Eclipse Graphical Modeling Framework (GMF) started as an Eclipse Technology subproject aims at providing generative component and runtime infrastructure for developing graphical editors in Eclipse. Before GMF, the effort required to build a custom diagram editor was an uneconomic proposition for most custom visual notations.

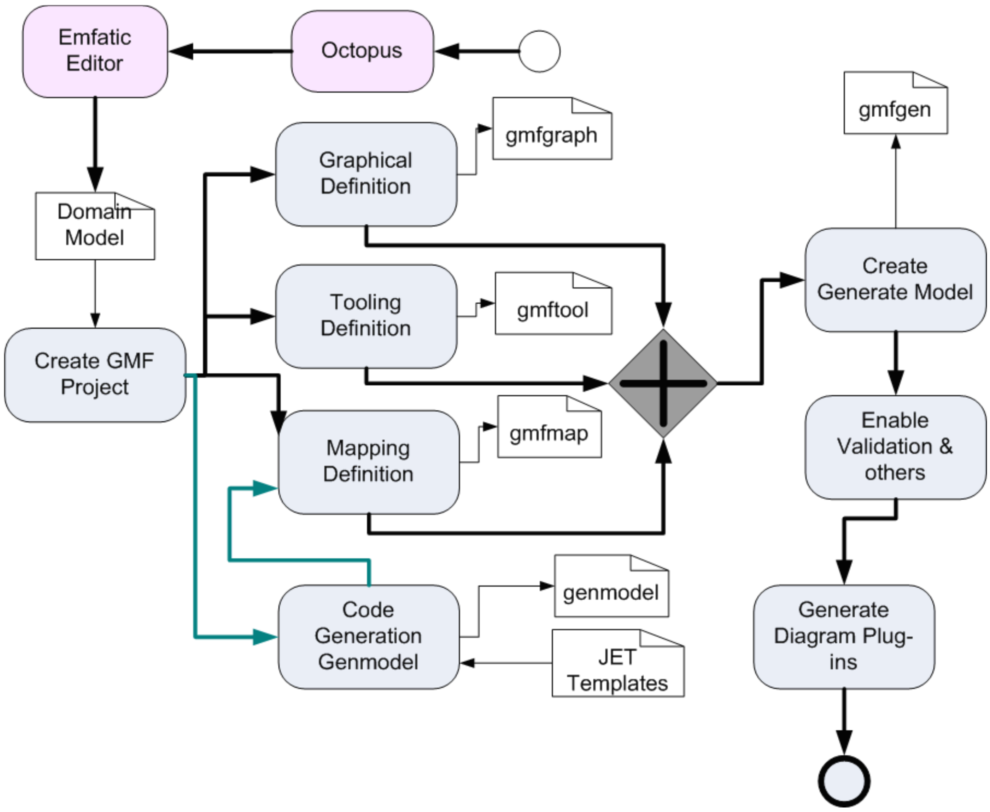


Figure 1: Graphical Modeling Framework Overview[4]

Figure 1 shows the overview of the implementation steps with different modeling definitions that specify a complete Diagram Editor. The information contained in these definitions helps to get a feeling for the capabilities of the editor to be generated. For details on EMF (Eclipse Modeling Framework of which GMF is a user), please consult the Chapter 3.

2.1 *Metamodel*

Ecore is the domain language used in EMF to express a datamodel. Ecore Models can be specified using annotated Java, XML Documents, or modeling tools like Rational Rose, then imported into EMF. We will use Octopus Tool, an Integrated Development Environment for UML + OCL specifications and Emfatic, a text editor supporting a more compact textual syntax for ecore datamodels [5].

2.2 GenModel and other necessary artifacts:

A single ecore file (the domain model) is not enough to specify a diagram editor. In fact, it is not even enough to specify the generated Java code for the M part of the MVC pattern. A Genmodel is required for this which serves as a decorator indicating properties such as Java package names. This Code Generation Genmodel would further support our implementation templates that help to generate code for the OCL invariants that we include in later stages as annotations.

It is mandatory to change to `"Base Package"` property in the genmodel to match the project name and the generating package structure.

As to the concrete steps, to generate the model and edit plug-ins, right-click the root of the generator model and select `"Generate Model Code"` and `"Generate Edit Code"` from the context menu.

But upon generation, the generated model code and edit plug-in will contains the following,

1. **`Model`** – provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (meta data) implementation class. Taking a closer look into `src` folder reveals an interface and an implementation class for each metamodel element.
2. **`Adapters`** – generates implementation classes (called `ItemProviders)` that adapt the model classes for editing and display.
3. The **.edit** project contains a number of utilities for building editors like standard table and property sheet views.

2.3 Building the Graphical Editor

Building a graphical editor for a Domain Specific Language is more intricate than the steps taken so far in visual modeling. The graphical definition files together with our domain model and other generator files are discussed in this section.

2.3.1 Conceptual Overview / Project

For building a GMF editor we start with a metamodel (the `.ecore` file) and the `.genmodel`. To generate a complete visual editor, a number of additional models have to be defined,

1. **`.gmfgraph`** - A model defining the graphical notation including shapes decorations and graphical nodes and connections.
2. **`.gmftool`** - A model for the editor's palette and other tooling.
3. **`.gmfmap`** - A mapping model that binds `gmfgraph` and `gmftool` to the ecore file. The two models defined above are technically (but usually, not conceptually) independent of our domain metamodel.

Take a look at Figure 1 to understand how the above mentioned models fit together. From all of these additional models, GMF creates the `.gmfgen` model – again a "low level" model that the code generator uses as an input, finally creating the `.diagram` plugin which contains the desired diagram editor.

2.3.2 Graphical Definition

A graphical definition model represented as **`.gmfgraph`** is used to define several things [6].

1. We can define a set of figures to represent the domain model elements. The default editor for this definition has dimension and color attributes such as line widths, foreground and back ground color attributes and static decorations. Any sort of figure can be constructed with the available options by adding them as a `New Child` to the present Figure that represents our Class i.e. `AggregatorContainer`. For example the figure for `Aggregator` shown in Figure 3 can be created as follows.
   - Create a rectangle with `Maximum Size Dimension, Minimum Size Dimension` and `Preferred Size Dimension` with X and Y attributes of values 70 and 50 respectively.
   - Create four rectangles and arrange them within the Figure with `XY Layout.`
   - Each of the four rectangles is set with a `Foreground Color Constant Color` attribute and `Background Color Constant Color` attribute to `LightGray`.
   - The Arrow Header in the figure is created with a `Polyline Connection` whose attributes are set so that the `Polyline` path is traced resulting in the required polyline structure resembling the arrow head.
2. We also define graph nodes and connections. Those domain model elements that are to be placed on the diagram editor canvas as nodes are defined as `Node.` The nodes in case of Figure 2 are `Node DataMessage`, `Node MessageContainer`, `Node AggregatorContainer` and `Node Header` . Domain elements that are to be specified as connections to link the domain elements are defined as `Connection.` The Figure 2 shows the `Connection PointToPointChannel` and `Connection LinkChannel` included.
3. We also define Compartments. `Compartments` are sections in nodes that can be collapsed and themselves contain other nodes or list of elements. We specify `MessageContainer` domain element as `Compartment` since it must hold `Header` element and `DataMessage` element.
4. Finally, we define diagram labels to show text associated with the graphical elements. The Label for our `AggregatorContainerFigure` if set as child element to Canvas instead of itself would result in an external label as shown in Figure 3 and therefore will be allowed to float and be positioned according to user's will.
5. In the properties view for each node element we connect each node element with their respective figures. For example, the `Node AggregatorContainer` is associated with `Rectangle AggregatorContainterFigure`.

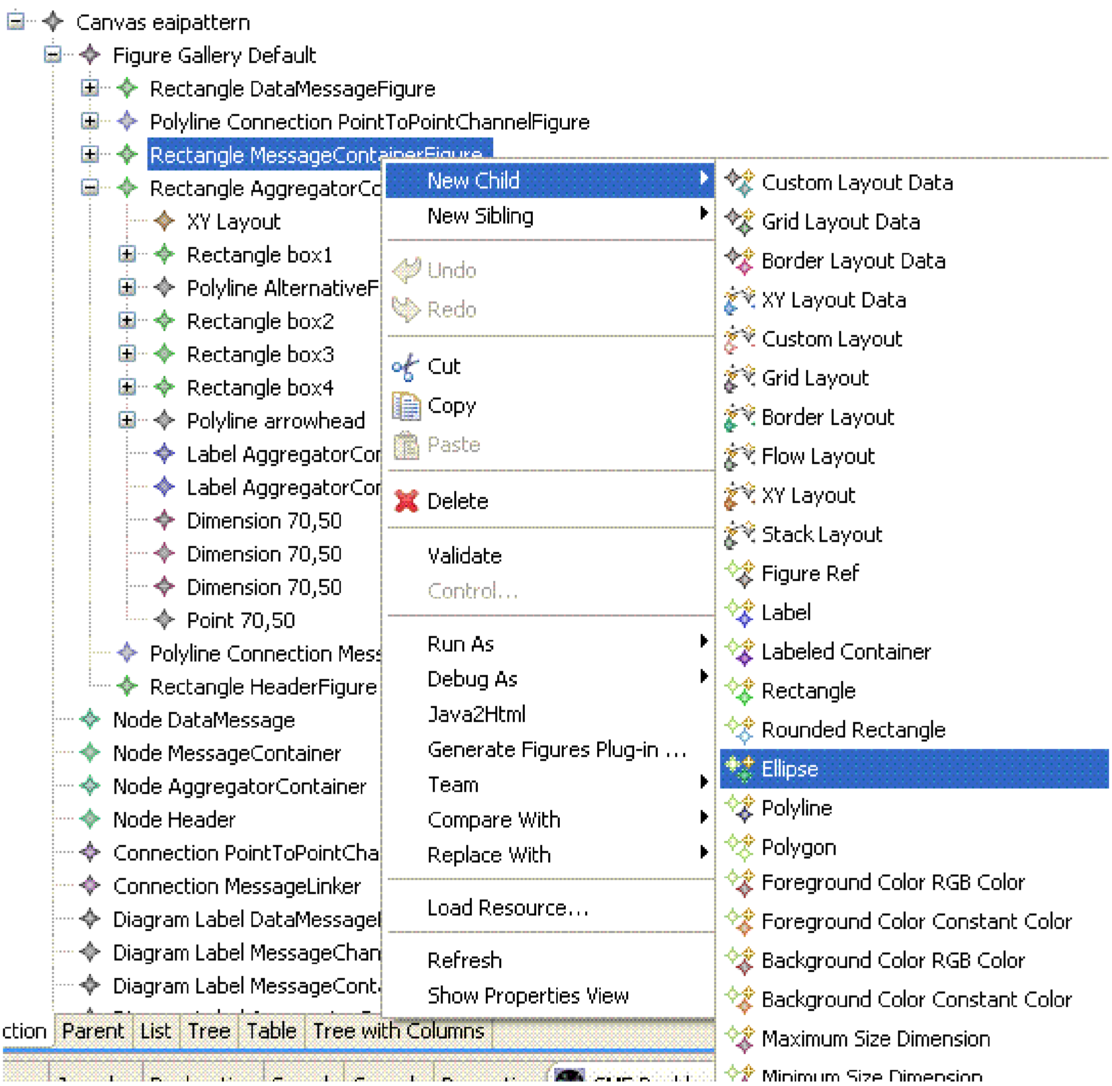


Figure 2: Graphical Definition

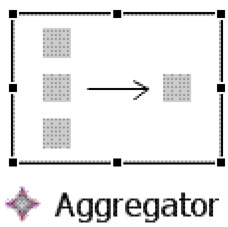
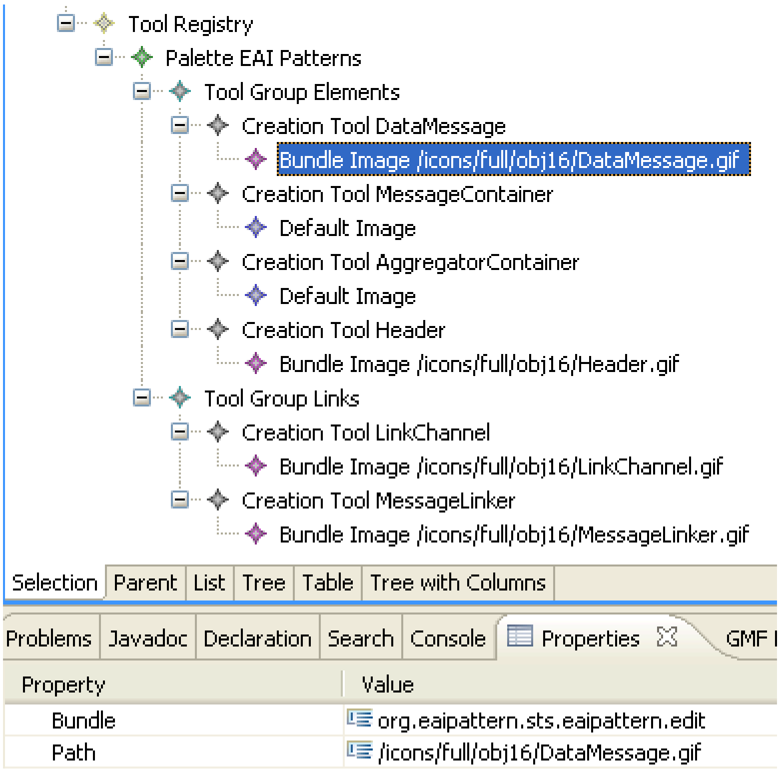


Figure 3: Aggregator Pattern

2.3.3 The Tool Model

The **`.gmftool`** model defines the set of palette entries. The palette is the set of buttons on the right of an editor that allows adding model elements to the domain model instance.

We assign icons to each of the of the creation tools. The icons for the model elements are created with a dimension of 16 X 16 pixels. The model elements can be grouped to form a `Tool Group`. If its collapsible property is set to `true`, this property gives a collapsible look to the `Tool Group` which can be retracted upon requirement. The effect of `Tool Group Elements` and `Tool Groups Links` can be seen in the palette on Figure 5.



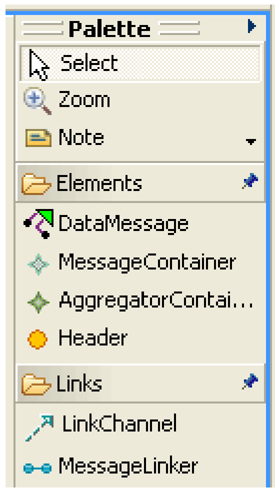


Figure 5: Tool Palette

Figure 4: Tooling Definition

Figure 5 shows how the generated tool palette for the tool smith to use would look like.

2.3.4 The Mapping Model

This is the most complex model. Here we map the tool definition and the graph definition to the domain metamodel. For example (1 in Figure 6) shows the mapping definition of `AggregatorContainer`. We will explain this mapping in more detail below. To be able to actually map the various elements, we have to add these other Resources to the editor that constitutes of `eaipattern.gmfgraph`, `eaipattern.ecore` and `eaipattern.gmftool`.

1. For each domain model element that we want to map directly onto the diagram surface (the eaipattern) we have to first define a `Top Node Reference`. For example, `AggregatorContainer` has its `Top Node Reference` defined at 1 in Figure 5.
2. Below that, we add a normal `Node Mapping`. It contains information about the model element to map `AggregatorContainer` and the property, in which the set of these elements is stored in the container (the container being the `eaipatterns` here and the property that contains the elements would be `EReference ref_eaipattern_AggregatorContainer`).
3. Below the `Node Mapping` we add a `Label Mapping`. This associates the label defined in the `gmfgraph` with the respective model element properties (here: we map the `name` provided for `AggregatorContainer`).
4. In case of `MessageContainer` we do like to have elements like `Header` and `DataMessage` added to its figure. So we define a `Compartment Mapping` below the `Node Mapping` of `MessageContainer`.
5. We also have to define `Header` and `DataMessage` as a `Child Reference` to the `MessageContainer` which identifies them as its children that is shown in 3 of Figure 6.
6. The `Child Reference` is associated with the `Compartment`, to ensure that the child collection is actually shown in the respective compartment.

The 4 of Figure 6 shows a mapping of a link, links being the mappings of the `Connections` of the `gmfgraph`. In the properties view, we can see some of the parameterization of the respective link:

1. The `Containment Feature` is the `EReference` in the containing metaclass that contains the reference objects (here: the `EReference ref_eaipattern_LinkChannel`).
2. Then we map the element that should represent the `Link`; this is the `EClass LinkChannel` in our case.
3. Next we have to tell GMF which feature of the link metaclass (here: `LinkChannel`) should take the reference to the source element (`Source Feature`)
4. We have to do the same thing with the target, this being stored in the `Target Feature` property.
5. We also have to define the graphical element (defined in the `.gmfgraph` model) that should represent the connection; here this is the `LinkChannel.`
6. Finally we have to define which tool should be used to actually instantiate such a link; this is `Tool LinkChannel` in our case.

All of this has to be mapped with a number of properties. The editors for doing that are just the usual tree editors, which makes all the process a bit cumbersome. Specialized editors are still under development which could make the link mapping easier in the future.

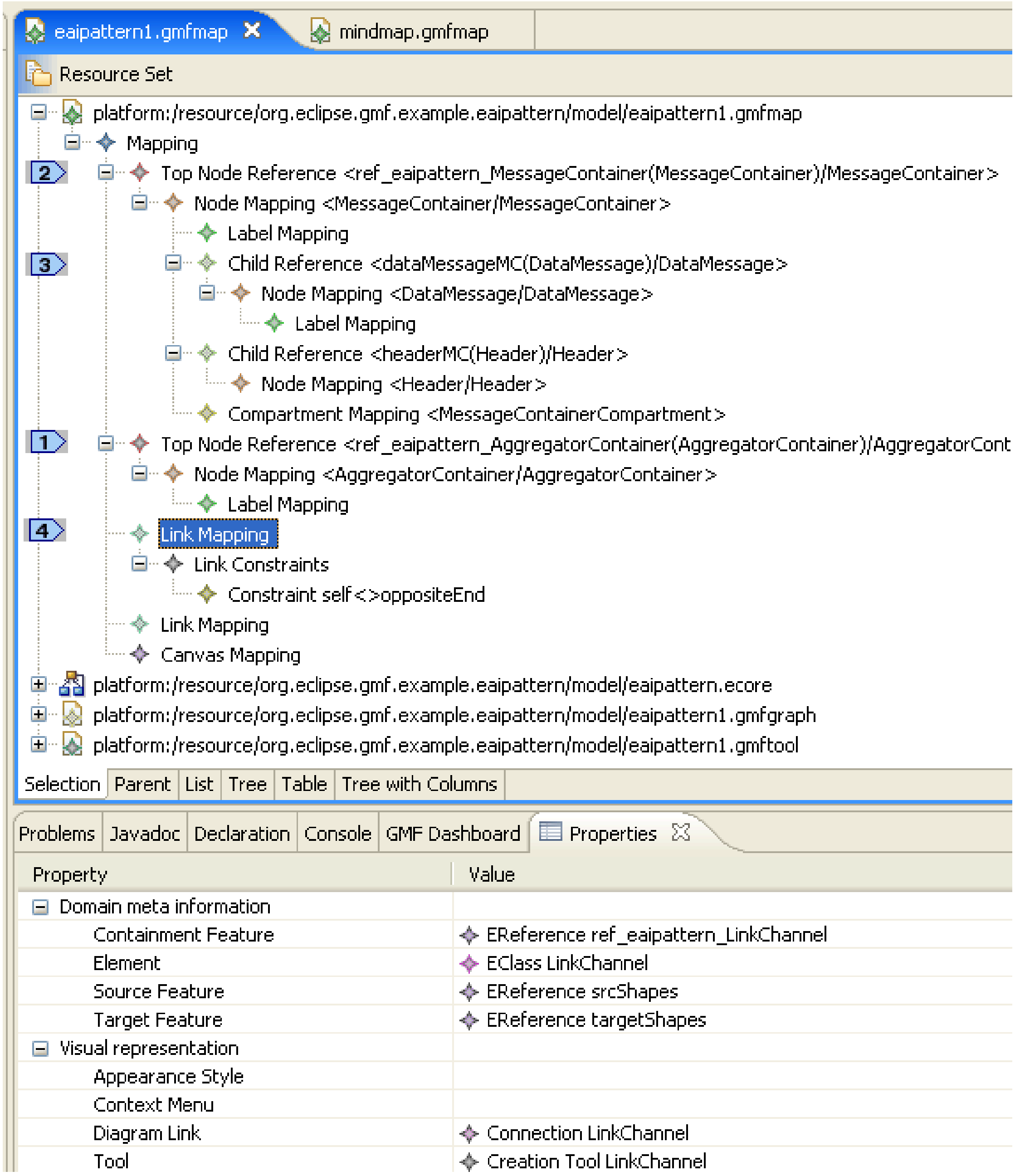


Figure 6: Mapping Model

2.3.5 Link Constraints

Link Constraints are used to validate the links upon creation between any two model elements. The constraints can be specified with OCL as `Source End Constraint` and `Target End Constraint`. More information on the usage of Link Constraints will be discussed in this section.

To add a constraint, we right-click on the "`Link Mapping`" and select **`New Child`** → **`Link Constraints`**. Further right-click on `Link Constraint` and select **`New Child`** → **`Source End Constraint`**. The "`Language`" property is set "`OCL`" and we will be needed to add the following OCL statement to the "Body" property as the Link Constraint:

```
self <> oppositeEnd
```

As shown in Figure 6, we have added a constraint to the creation of a link, based on its source end; that is, the srcShapes element from which a link is being created. In the OCL we have specified the only condition that will evaluate to true, and therefore will allow the link to be created. The condition explains that the source element should not be equal to the "oppositeEnd" of the link (the target). In this case, the context of "self" is the source srcShapes, and "oppositeEnd" is a custom variable added to the parser environment for link constraints.

Two types of constraint that can be specified in Link Mapping are,

1. **Source End Constraint:** In this type of Constraint, oppositeEnd is undefined until the other end of the connection is available. This type of constraint is first evaluated when the connection is started.
2. **Target End Constraint:** In this type of Constraint, oppositeEnd value and the value of self is known and is evaluated when the connection is tried to be created to a specific target element.

To take a look at a more complicated Target End Constraint. Consider a scenario where a link should not be allowed between a domain element representing Class AggregatorContainer and Class MessageContainer. With reference to the metamodel both classes inherit from a Class Shapes.

The OCL UML model would look like

```
+<class> Shapes
<attributes>
+name:String;
<endclass>

+<class> AggregatorContainer <specializes> Shapes
<endclass>

+<class> MessageContainer <specializes> Shapes
<endclass>
```

To model a Target End Constraint select **New Child → TargetEndConstraint**. The "Language" property is set to "ocl" and we will be need to add the following OCL statement to the "Body" property as mentioned before.

```
self.oclIsTypeOf(AggregatorContainer) <>
                    oppositeEnd.oclIsTypeOf(MessageContainer)
```

2.4 The Generator Model

The transformation from the mapping model to the generator model is described here. Referring to the generated `.gmfgen` model, one can notice quite a few things that were created during the process. A general overview of this transformation can be seen in Figure 7. From this diagram, one can see that the selected mapping model is first opened and validated. The canvas is processed, followed by each node, and then each link. Finally, the new generator model (`.gmfgen`) is saved and validated [4].

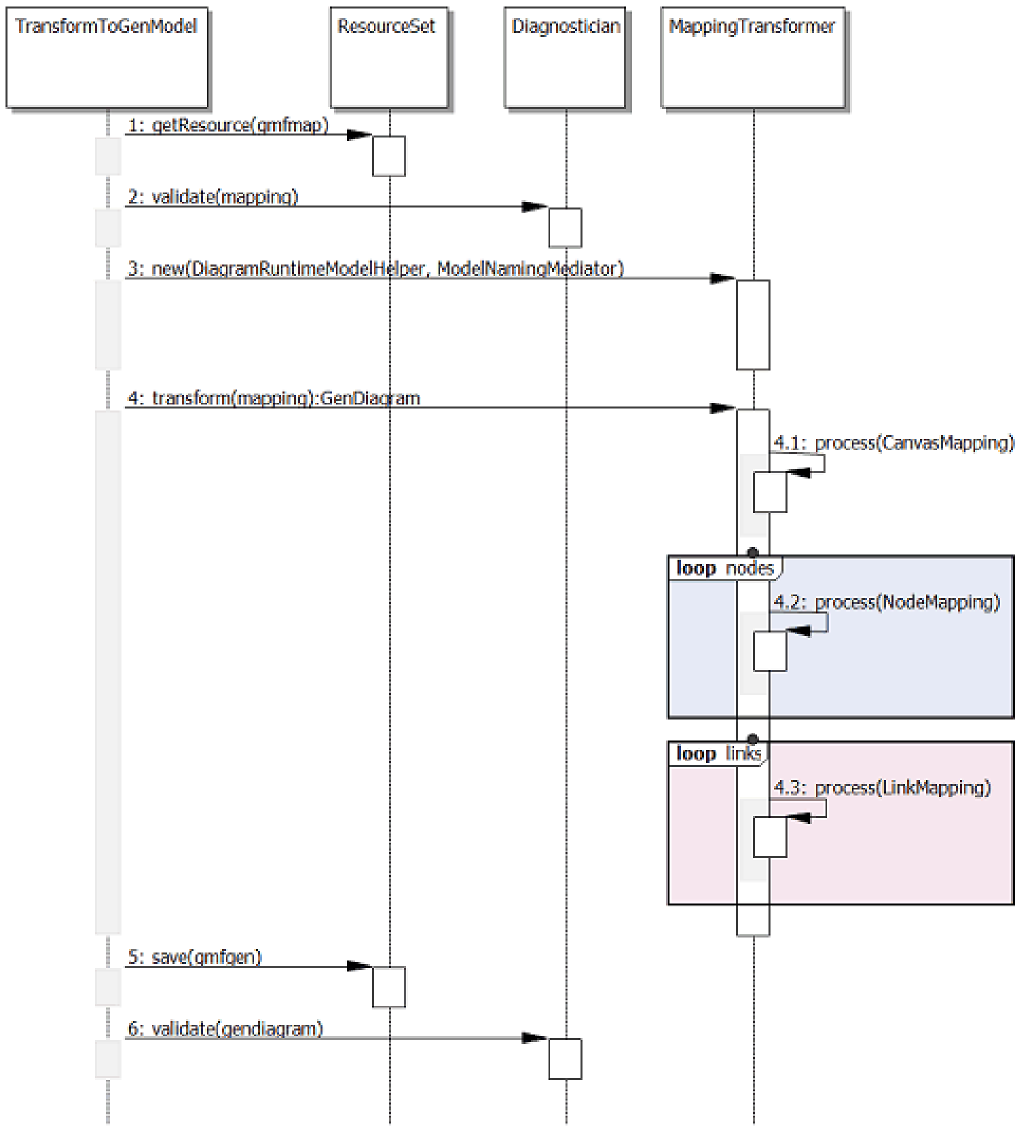


Figure 7: Mapping Model to Generator Model

During the processing of the canvas, a `GenModelMatcher` is created and the EMF `genmodel` for the domain model is located. With a quick look at the generator model itself, one can find that a large number of properties related to the canvas are need to be set, in addition to the plug-in that is used to deploy our editor.

Custom Properties like enabling the Validation Framework for GMF and setting the layout of Figures on canvas can be changed at this stage. We discuss in detail these custom properties in later chapters.

2.5 Summary

In this chapter we have reviewed the different modeling definitions employed in creating a diagram editor using the Eclipse Graphical Modeling Framework. We have discussed about the Link constraints that are employed here to validate the links even before creating them. Other way of providing validation for domain model instances is by using Audit Containers for our domain model. This topic is further discussed in section 4.1.

3 Editors for EMF Ecore

In this chapter, we review the ecore language definition and the ecore editors used in our implementation. Thereafter, we review the syntax for specifying the domain models using these editors. We will discuss about using Emfatic[1] editor and Octopus[2] Tool Kit [3] for developing EMF Ecore metamodels.

3.1 EMF Ecore

Eclipse Modeling Framework (EMF) is a Java based framework for developing model-driven applications and other integrated software tools. EMF is a modeling framework and code generation facility for building tools by taking a datamodel specified in ecore as starting point.

With the model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model and a basic editor.

An ecore model can be specified using annotated Java, UML, XML Schema or with modeling tools like Rational rose or Omondo. Figure 8 shows the main constructs of Ecore. The kernel model contains elements `EClass`, `EDataType`, `EAttribute` and `EReference`. These model elements are needed to define classes (`EClass`), their attributes (`EAttribute`) and associations (`EReference`). `EClasses` can be grouped to `EPackages` which might be again structured into subpackages. In addition, each model element can be annotated by `EAnnotation` which we use to specify OCL constraints. Furthermore, there are some abstract classes to better structure the Ecore model, such as `ENamedElement`, `EtypedElement`, etc. It is important to note that the EMF metamodel (Ecore) again is expressed itself in ecore.

From an EMF model, a set of Java classes for the model and a tree based editor can be generated. The generated classes provide basic support for creating/deleting model elements and persistency operations like loading and saving. Relations between EMF model classes are handled by special EMF lists, extending the Java list classes. Moreover, EMF models can be used as underlying models in new application plugins. But in many cases, the EMF model by its own is not powerful enough to express the complete model behavior. Therefore the generated code can be extended by the developer in order to add new functionalities that are not expressed in the EMF model.

For creating EMF metamodels other than the default EMF editor available for creating Ecore metamodel, other approaches can also be used in creating such files. The following sections will discuss about using Emfatic Editor [5] and Octopus Tool Kit [3] for developing EMF Ecore metamodels.

[1] Text based editor for creating metamodels.

[2] OCL Tool for Precise UML Specifications

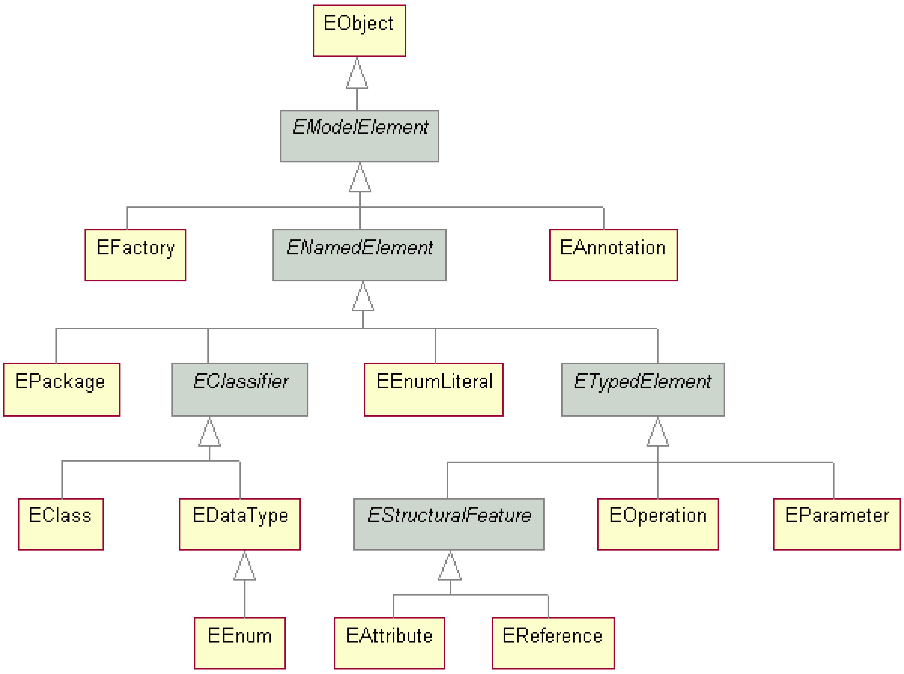


Figure 8: Class Hierarchy of an Ecore Metamodel

3.2 Emfatic

Emfatic is a language designed to represent EMF Ecore models in a textual form. It can be a useful tool for viewing and building the models. Emfatic generator can convert the existing EMF models into emfatic textual format and can generate EMF models from emfatic textual model. Figure 9 shows the metamodel for emfatic.

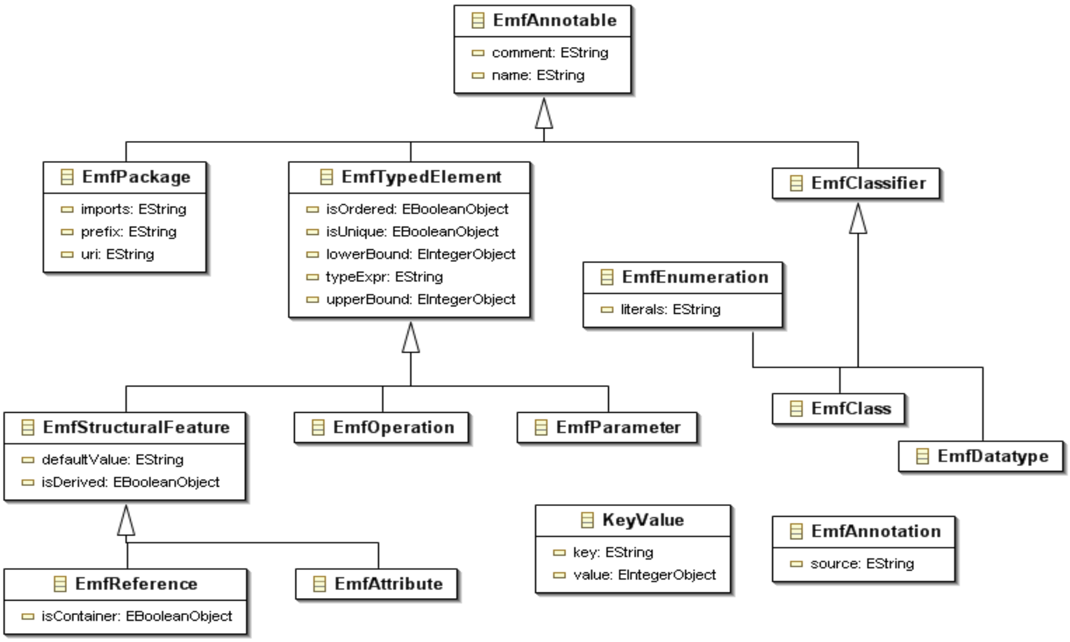


Figure 9: Emfatic Metamodel

Consider the EMF Diagram shown in Figure 10, representing the `Class DataMessage` and `Class MessageLinker`. The Code snippet below is the equivalent in Emfatic format,

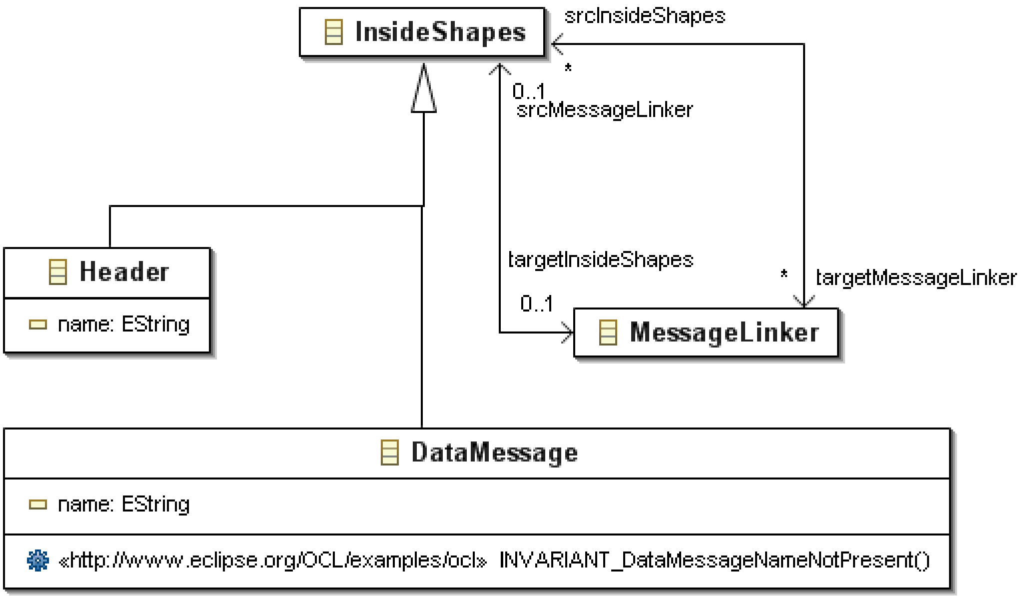


Figure 10: Emfatic Example

```
abstract class InsideShapes {
            .
            .
}

class DataMessage extends InsideShapes {
 1  attr String name;
 2  @"http://www.eclipse.org/OCL/examples/ocl"(
       invariant = "self.name.size() > 0")
 3  op boolean INVARIANT_DataMessageNameNotPresent(
                   ecore.EDiagnosticChain diagnostics,ecore.EMap context);
}

class MessageLinker {
 4  !ordered ref InsideShapes[0..*] #targetMessageLinker srcInsideShapes;
     ref InsideShapes#srcMessageLinker targetInsideShapes;
}
```

1. Emfatic syntax for class declarations is very similar to Java, however a few additional quirks are required to allow for all possibilities of Ecore creation. The code snippet contains simple class declarations demonstrating the use of keywords `class` and `abstract`.
2. Inheritance is specified with the keyword `extends`. Unlike in Java, there is no `implements` keyword to distinguish inheritance from interface implementation.
3. A number of datatypes defined in `Ecore.ecore` have shorthand notation in Emfatic. The table in Appendix B lists the Emfatic shorthand and corresponding `Ecore.ecore` type name for each of the basic types and as well as corresponding Javatype.
4. As shown by 1 `Class DataMessage` has an attribute `name` of type DataType `String`.

5. Constraints for EMF are written in the form of annotations. Generally, annotations can be attached to any EMF element. And only the source and detail features of resulting `EAnnotation` can be specified in Emfatic. The Syntax for Annotation in Emfatic follows with a @ symbol and value for `EAnnotation` source attribute. Key/Value pairs for annotation may appear in paranthesis following the `source` attribute. As shown by 2 the constraints are implemented as annotations with **`http://www.eclipse.org/OCL/examples/ocl`** being the value of `source` attribute. The `invariant` and the constraint expression constitutes the Key/Value pair.
6. 3 shows `INVARIANT_DataMessageNameNotPresent`() that returns a value of type `Boolean.` This method is used to handle the declared constraint. The input parameters for the method are a `DiagnosticChain` and a `Map.` Violation of a constraint adds a `Diagnostic` to the chain and results in a `false` return value.
7. 4 shows the Ecore Class features `EReference` represented in our emfatic example. The other Ecore class features that can be represented in emfatic are `EAttribute,` `EOperation` and `EParameter.` Refer to Appendix C that shows the list of emfatic keywords for the Ecore C`lass Features`. To represent the EMF `Class Features` in emfatic, the following syntax is used to introduce and differentiate attributes, references and operations,

modifiers featureKind typeExpression name ';'

With reference to the below `Class Feature` implementation,

```
!ordered ref InsideShapes[0..*] #targetMessageLinker srcInsideShapes;
```

- **`modifiers`** - !`ordered` refers to the modifiers implementation. The other modifiers available are `readonly`, `volatile`, `transient`, `unsettable`, `derived`, `unique`, `ordered`, `resolve` and `id.` Please refer to Appendix B for further details on modifers.
- **`featureKind`** – `ref` is the `featurekind` in the above implementation. The other `featureKinds` are shown in Appendix B.
- **`typeExpression`** – `typeExpression` specifies the `lowerbound` and `upperbound` attributes of `ETypedElement.` In our case the `lowerbound` = 0 and `upperbound` = *. When the `typeExpression` is not specified then the `ETypedElement` gets the defaults (`lowerbound` = 0 and `upperBound` = 1).

3.3 Octopus

Octopus is an acronym for **OCL** **T**ool for **P**recise **U**ML **S**pecifications. This Eclipse based tool provides support for UML in textual format and OCL. Octopus offers two main functionalities [3],

1. **Statically check OCL expressions** - It checks the syntax, as well as the expression types, and the correct use of model elements like association roles and attributes.
2. Transform the UML model, including the OCL expressions, into Java code.

Octopus fully conforms to version 2.0 of the OCL standard. All new constructs, like derivation rules and initial value specifications, are completely supported. Furthermore, Octopus offers the possibility to view expressions in an SQL-like syntax. The semantics of the original expressions, written in the standard syntax, remain fully intact, while their appearance becomes more familiar for those who have been working with databases [3].

With the support provided for OCL in Octopus we can use plug-ins developed at STS to convert the UML and OCL files into emfatic files. With reference to the functionality provided by the `octopus2emfatic` plug-in, the plug-in takes care of creating invariants as annotations and the required invariant methods that are required for handling constraints.

Consider the below specified UML and OCL code snippets,

```
<package> eaipattern
.
-- Definition for Class Header
+<class> Header
<attributes>
+name:String;
<endclass>
-- Definition for Class MessageContainer
<class> MessageContainer
<attributes>
+name:String;
<endclass>
-- Composite relationship between MessageContainer and Header
-- messageContainer and headerMC denotes the association Roles played by their respective
classes
<associations>
+ MessageContainer.messageContainer[1..*] <composite> -> + Header.headerMC[1..*];
<endpackage>
```

UML Code Snippet

To generate the emfatic file for the Octopus model, right click on the Octopus project containing the model files and select "**`Convert to Emfatic`**" from the context menu as shown in Figure 11.

```
context MessageContainer
        -- This constraint check if the name of the MessageContainer is present
         inv MessageContainerNeedsName:
                 self.name.size()>0

        -- This constraint checks atleast one Header Diagram Element should be present
         inv AtleastOneHeader:
                 self.headerMC->size() >0
```

OCL Code snippet

Figure 11: Convert to Emfatic

The generated emfatic file for the above mentioned `uml` and `ocl` models would be,

```
package eaipattern;

class MessageContainer {
 attr String name;
 !ordered val DataMessage[0..*]  dataMessageMC;
 !ordered val Header[1..*]  headerMC;
 @"http://www.eclipse.org/OCL/examples/ocl"(invariant =
    "self.name.size() > 0")
    op boolean INVARIANT_MessageContainerNeedsName(
             ecore.EDiagnosticChain diagnostics,ecore.EMap context);
 @"http://www.eclipse.org/OCL/examples/ocl"(invariant =
    "self.headerMC->size() > 0")
    op boolean INVARIANT_AtleastOneHeader(ecore.EDiagnosticChain
             diagnostics,ecore.EMap context);
}

class Header {
    attr String name;
}
```

3.4 Summary

We will employ both of the above discussed editors for our domain model specification. The generation of metamodel starts with Octopus where we specify our metamodel in UML and the constraints for the model in OCL. With the built-in editor support for OCL runtime checking, Octopus is a great tool to build models and check the OCL syntax. With the custom developed `octopus2emfatic` plug-in we generate the emfatic equivalent model for the octopus UML model. This custom plug-in creates methods to handle the constrains and adds them as annotations to the required domain model elements. Further we create the domain model by converting the emfatic file into the required model using the Emfatic plug-in.

4 EMFT Technologies for GMF - *Implementation*

In this chapter we will be discussing about the two of the technologies developed around EMF either to complement or extend it. The complete list of technologies comprises of Validation, OCL, Query, Transaction and many more. Each technology has a similar intention in co-ordinating with other technologies in extending EMF.

Figure 12 shows the diagram editor that uses one of the EMFT technologies i.e. EMFT OCL to maintain its domain model integrity with which the model element instances created in the editor are checked for well-formedness with the constraints specified in the domain model. Figure 12 shows the domain model instances for Dead Letter Channel Pattern and Aggregator Pattern. Please consult Section 5.2 for more information on constraints used for well-formedness. The constraints broken against these instances are displayed as Errors, see 1 in Figure 12.

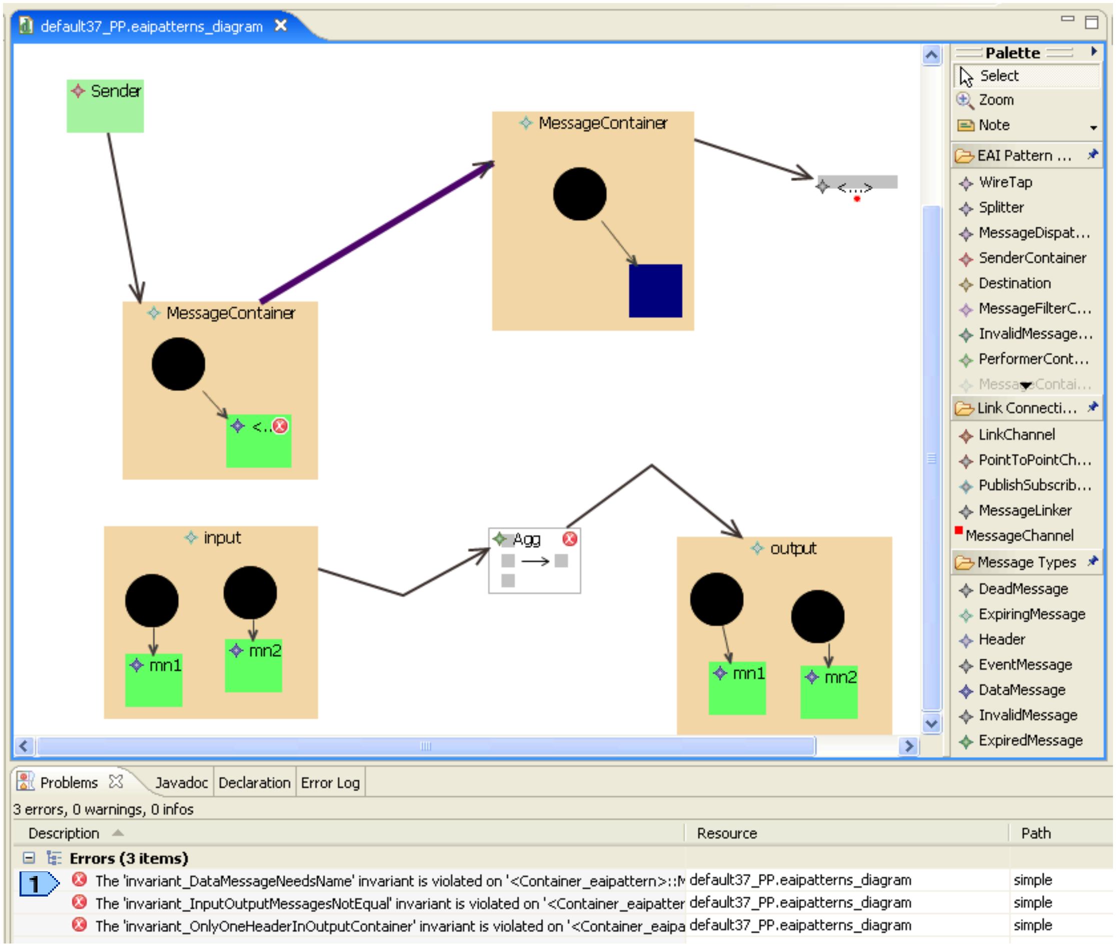


Figure 12: Diagram Editor - EAI Patterns

4.1 *EMFT Validation*

4.1.1 Overview

The EMFT Validation framework provides a means to ensure the well-formedness of EMF models. This framework provides support for constraint providers for any EMF metamodel, customizable model traversal algorithms, parsing for constraint languages, configurable constraint bindings to application contexts and validation listeners.

4.1.2 EMFT Validation in GMF

For the sole purpose of checking the well-formedness of models, this framework finds its way into GMF as Audit Rules. An audit rule accepts constraints that are checked by EMFT Validation for a domain model instance. The constraints specified as audit rule can be enabled / disabled in the central constraint registry. The severity of the audits can be specified as one of the following options - ERROR, INFO and WARNING. Audits are also helpful in providing warning to the user regarding the visual domain model instance. Even if the model is wellformed, audits can provide useful information. For example, use an audit rule to warn the user that the the number of children added in a compartment[1] is more than usual, even though if the number complies with the specified constraint for the compartment.

To consider implementing the functionality of audit containers in our eaipattern example let us consider the constraint below for `AggregatorContainer`.

```
self.name.size() > 0
```

Open the mapping definition of our example (`eaipattern.gmfmap`) and right-click on the `Mapping Node`. Select "**`New Child → Audit Container`**" and give it a name "`Audits for Aggregator`". Assign an `id` and give it a `description`. To the container, add a new "`Audit Rule`" named "`Aggregator Name Check`". Since this Audit Container targets the `AggregatorContainer`, we add a "`Domain Element Target`" to the `Audit Rule` and select "`EClass AggregatorContainer`" as the Element. Add a new child "`Constraint`" to the `Audit Rule` and enter the above constraint in the body and specify the language as `OCL`. This specific constraint detects if the "`name`" attribute which is mapped as a `Label` to `AggregatorContainer` instance is provided with a name. After generating the `eaipattern.gmfgen` model, it is necessary to set the "`Validation Enabled`" Property of Gen Diagram element to "`true`" in order for our audit to run. To view familiar decorators when the audits for our domain instance model are broken, set "`Validation Decorators`" property to true.

Finally, set the "`Validation Provider Priority`" to any value higher than "`Lowest`". After the above modifications, regenerate the editor code using "`Generate Diagram Code`". Doing so will result in many new extensions listed in the diagram editor's `plugin.xml` file. A closer examination will reveal the extension-points of EMFT Validation Framework to which our diagram editor contributes.

[1]Term used in graphical defintion to specify that a particular domain element can hold other child elements with which the former domain element has a composite relationship

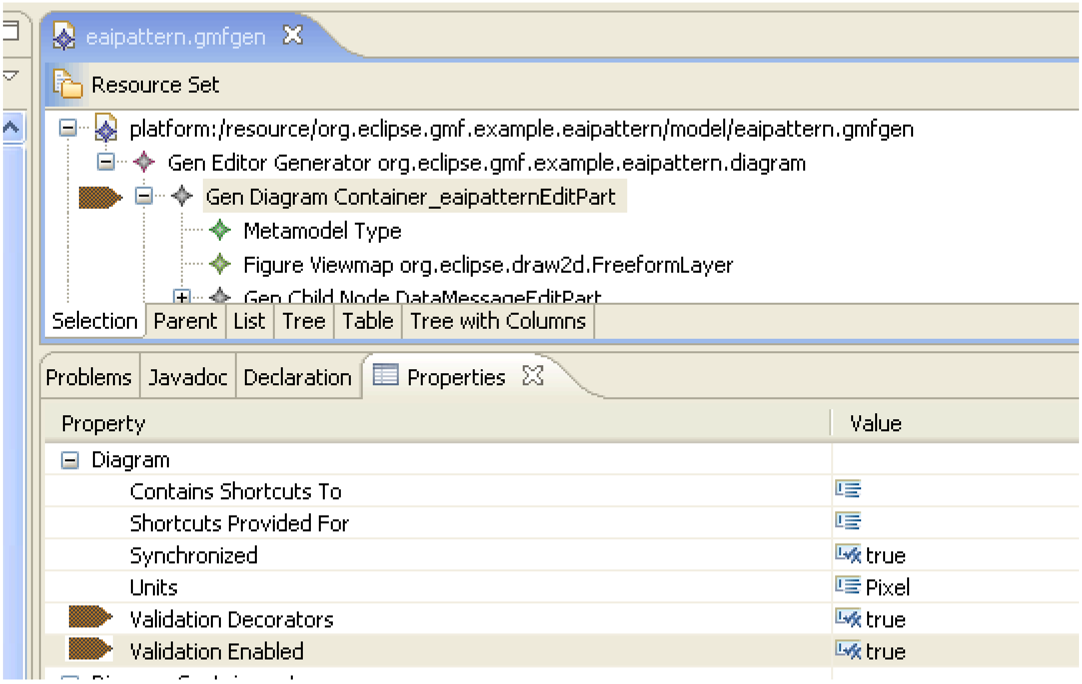


Figure 13: Validation Enabled - Mapping Definition

The following are the two main important extension points that are implemented when the audit containers are used in GMF.

1. 1 shows the extension point `org.eclipse.emf.validation.constraintProviders`. This is used to provide constraints. Two ways of providing constraints are possible with this extension point.
   - **`Static Constraint Providers`** - We declare constraints in `plugin.xml`. This is how audit containers implement their constraints.
   - **`Dynamic Constraint Providers`** – These address situations where constraints cannot be declared in `plugin.xml` e.g. when the constraints are defined in models or other resources. These providers declare a class implementing the `IModelConstraintProvider` interface which is responsible for making constraints available on appropriate triggers, organizing them into categories, etc.
2. 2 shows the extension point `org.eclipse.emf.validation.constraintBindings`. This allows clients of the EMF Validation framework to define "`Client Contexts`" that describe the objects that they are interested in validating the constraints, and to bind them to the same. But in our extension point implementation, 3 uses an alternative which is to define a selector class using a `selector` element. `Client Contexts` can be bound to constraints, individually, or with constraint categories.

```
1 <extension point="org.eclipse.emf.validation.constraintProviders">
    <category
          id="org.eclipse.gmf.example.eaipattern"
          mandatory="false"
          name="Audits for Aggregator">
    <![CDATA[AggregatorName Audits]]>
    </category>
    <constraintProvider cache="true">
    <package
namespaceUri="http://de.tuhh.sts.octopus/octopus2emfatic/2006/eaipattern"/>
          <constraints categories="org.eclipse.gmf.example.eaipattern">
                <constraint id="AuditName"
                      lang="OCL"
                      name="Aggregator Name Check" mode="Batch"
                      severity="ERROR" statusCode="200">
                      <![CDATA[self.name.size()>0]]>
                    <description><![CDATA[This Audit.....]]></description>
                    <message><![CDATA[No Name found in
Aggregator]]></message>
                      <target class="eaipattern.AggregatorContainer"/>
                </constraint>
          </constraints>
    </constraintProvider>
  </extension>
2 <extension point="org.eclipse.emf.validation.constraintBindings">
    <clientContext default="false"
id="org.eclipse.gmf.example.eaipattern.diagram.DefaultCtx">
3 <selectorclass="org.eclipse.gmf.example.eaipattern.diagram.providers.
            EaipatternValidationProvider$DefaultCtx"/>
    </clientContext>
    <bindingcontext="org.eclipse.gmf.example.eaipattern.diagram.
                                                       DefaultCtx">
     <constraint ref="org.eclipse.gmf.example.eaipattern.diagram.
                                                       AuditName"/>
    </binding>
  </extension>
```

4.1.3 Testing the Audit Containers

After generating the diagram plug-in for our project, launch a new runtime workspace to test the diagram. The generated editor uses the domain model as input for specifying the editor commands. For each model element, the editor contains insertion, deletion, editing and moving commands.

Start with creating an empty GMF project and invoke **New → Eaipattern Diagram.** Create a new instance Diagram Element of AggregatorContainer. Before proceeding with validation of our domain model instance, a look at the same plugin.xml generated in the diagram plug-in will show the extension point implementations inserted because of enabling the Validation Providers and Validation Decorators as shown in Figure 13.

1. 1 in plugin.xml shows the extension which contributes to the Menu with providing a "Validate" under "Diagram" Menu in our launched runtime workspace.
2. 2 shows the extension which enables the familiar Eclipse decorators for our elements when the implemented audits are violated.

```
1   <extension id="ValidationContributionItemProvider" name="Validation"
      point="org.eclipse.gmf.runtime.common.ui.services.action.contribution
ItemProviders">
      <contributionItemProvider checkPluginLoaded="true"
         class="org.eclipse.gmf.example.eaipattern.diagram.providers.Eaipat
ternValidationProvider">
         <Priority name="Medium"/>
         <partContribution
id="org.eclipse.gmf.example.eaipattern.diagram.part.EaipatternDiagramEditor
ID">
            <partMenuGroup menubarPath="/diagramMenu/"
id="validationGroup"/>
            <partAction id="validateAction"
menubarPath="/diagramMenu/validationGroup"/>
         </partContribution>
      </contributionItemProvider>
   </extension>

2   <extension id="validationDecoratorProvider"
name="ValidationDecorations"
point="org.eclipse.gmf.runtime.diagram.ui.decoratorProviders">
      <decoratorProvider
class="org.eclipse.gmf.example.eaipattern.diagram.providers.EaipatternValid
ationDecoratorProvider">
         <Priority name="Medium"/>
         <object
class="org.eclipse.gmf.runtime.diagram.ui.editparts.IPrimaryEditPart(org.ec
lipse.gmf.runtime.diagram.ui)" id="PRIMARY_VIEW"/>
         <context decoratorTargets="PRIMARY_VIEW"/>
      </decoratorProvider>
   </extension>
```

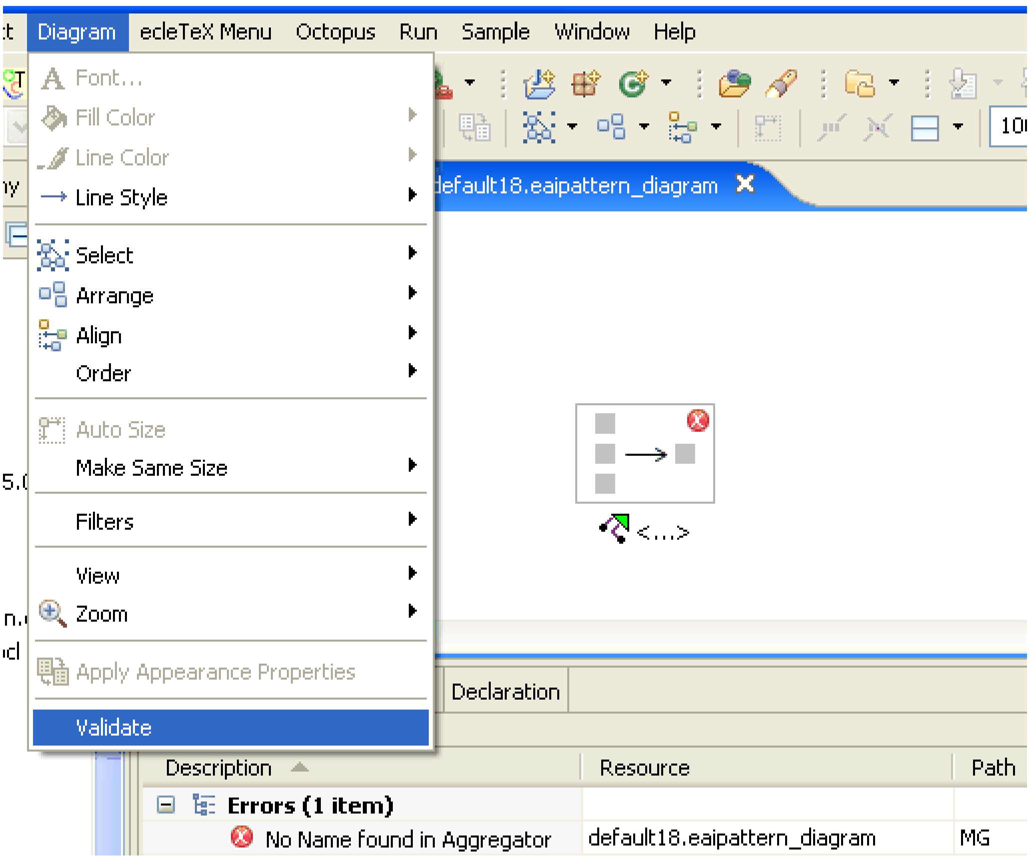


Figure 14: Validation in Domain Model Instance

Figure 14 shows the validation of our diagram element belonging to `AggregatorContainer`. By implementing the audits for validating our domain model instance we can provide important information and suggestions to correct the model instance. The Validation decorators come into play with generating the error symbols on the diagram elements. The implemented Constraints in the form of audits help to generate error messages commenting on the result of the invariants that were broken during validation.

4.2 *EMFT OCL in GMF*

4.2.1 Overview

The EMFT OCL framework provides basic infrastructure for OCL constraint parsing, content assist for user models, OCL constraint validation and specifying OCL queries and conditions. It provides API for constructing, validating and evaluating OCL constraints and queries on EMF model elements. This framework includes a parser/interpreter for Object Constraint Language version 2.0 for EMF. Using this parser one can evaluate OCL expressions on elements of any EMF metamodel.

The whole approach of implementing EMFT OCL framework for GMF is with reference to article [1] explains the implementation of model integrity in EMF with EMFT OCL. We will follow the same approach in implementing the model integrity for Domain Model instances created in GMF that can be evaluated with EMFT OCL. Before this approach let us look at the support provided for constraints in EMF without using any external frameworks. This will help us understand the approach that will be used in GMF.

4.2.2 EValidator API

OCL Constraints can be specified in EMF Metamodel as annotations. EMF Codegen generates validator classes for the model elements containing constraints. The validator classes generated have dedicated method skeletons that if provided with validation code could evaluate the constraints for the model elements [7].

Figure 15 shows the way constraints are implemented as annotations in EMF. Here the invariant named `INVARIANT_MessageContainerNeedsName` has an annotation of OCL constraint that looks as follows,

```
self.name.size() > 0
```

Let us look at the method generated for `MessageContainer` for the above specified constraint.

The `EValidator` API generates individual Message Body 1 for each constraint, but these methods simply delegate to the invariant methods on the objects themselves [8]. The framework prescribes the form of invariant constraints: boolean-valued operations with a `DiagnosticChain` and a `Map` as input parameters. Violation of a constraint adds a `Diagnostic` to the chain and results in a `false` return value.

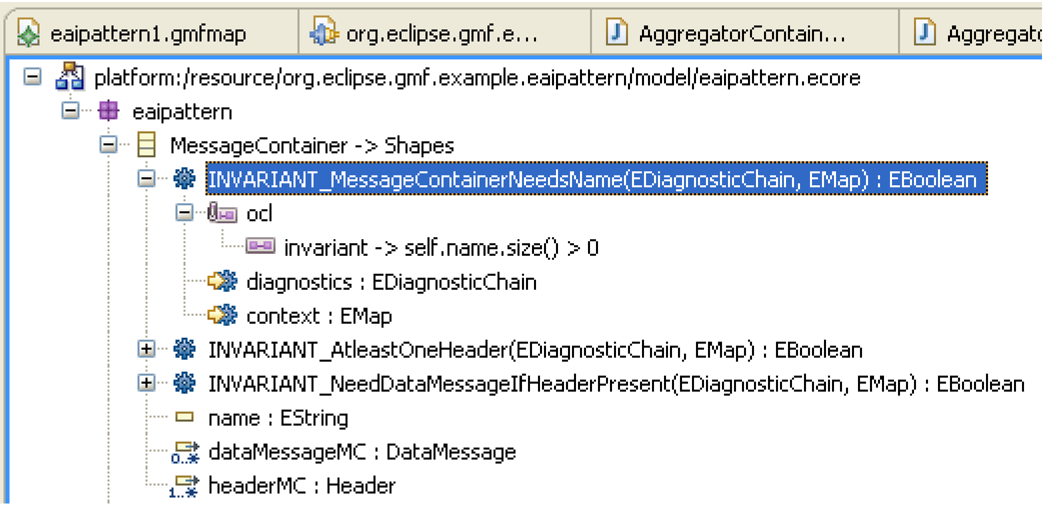


Figure 15: Invariant - MessageContainer

The message body as shown at 2 is incomplete and the generated code must be modified by hand or by other means to explain EMF how to implement a constraint. To accomplish this we tell EMF to use the additional code that is generated with the help of JET templates and its detailed approach is discussed in section 4.3.

```
  /**
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
1 public boolean INVARIANT_MessageContainerNeedsName(
                                        DiagnosticChain diagnostics, Map context
) {
2   // TODO: implement this method
  // -> specify the condition that violates the invariant
  // -> verify the details of the diagnostic, including severity and
message
  // Ensure that you remove @generated or mark it @generated NOT
    if (false) {
      if (diagnostics != null) {
        .
      }
      return false;
    }
    return true;
  }
```

Code Snippet : MessageContainerImpl.java

The base class for each invariant provides validation on below listed aspects [8]:

1. The actual multiplicities of the attributes and references match the bounds defined in the model.
2. The defined data type of the attributes is respected.
3. Any cross referenced objects are container in resources.
4. Every proxy is properly resolved.

4.3 *Adding Constraints with JET Templates*

Continuing with the article [1], we will follow the specified approach to get the OCL expression transformed into the EMF model and integrate them with GMF for validation at runtime. The approach starts with specifying the OCL expressions as annotations to the model elements that is taken as context for the OCL invariant to perform its well-formedness checking.

The mechanism of conversion of EMF metamodel with the specified approach [1], is to involve the EMF Codegen along with the additional JET templates [9]. Figure 16 shows the Flow chain process in generating the diagram editor with the Template Engine (JET) and its position in our implementation.

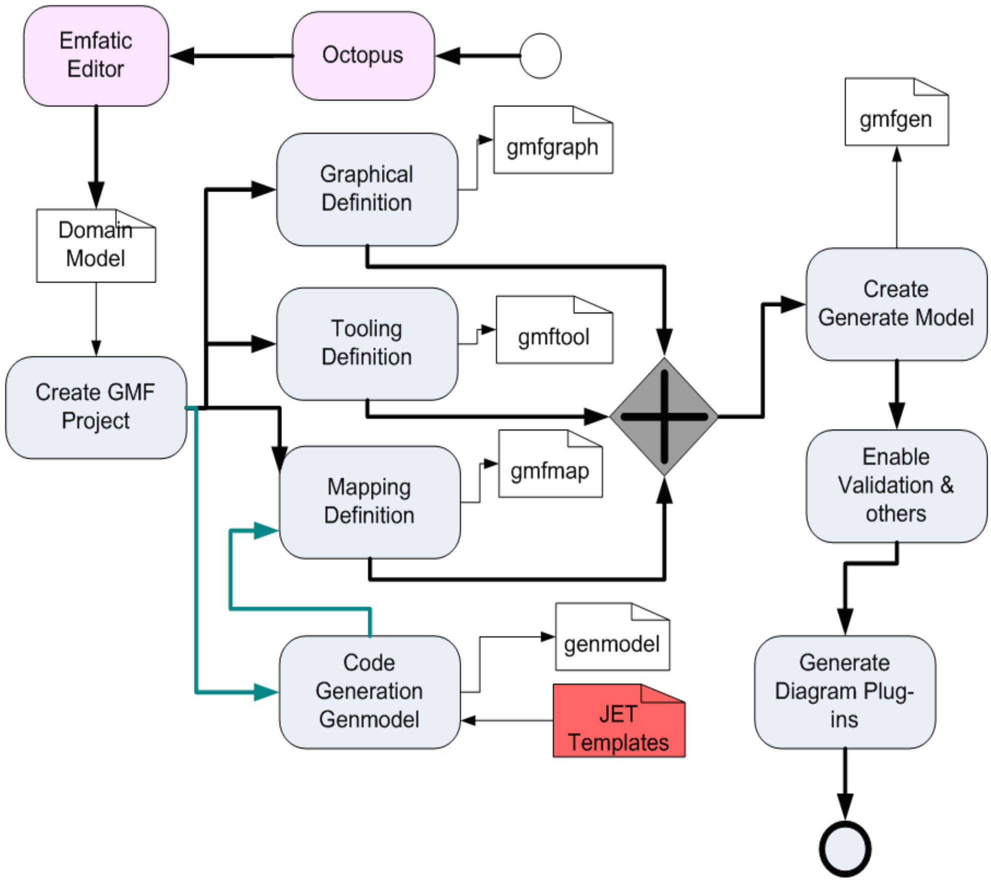


Figure 16: Flow Chain Process – Outline of our implementation

Such templates are needed to generate the validation operation body that was left to fill up by the EValidator API. The scripting statements in these templates parse the annotations containing the constraints. At runtime, the constraint is available as a `String`, which is interpreted to obtain a result. They further may also generate additional support fields.

The code snippet below shows the additional code added to the invariant method body 3 that we previously saw in `MessageContainerImpl.java`. This code will tell EMF how to implement this constraint.

```
public boolean INVARIANT_MessageContainerNeedsName(
                DiagnosticChain diagnostics, Map context) {
      if (INVARIANT_MessageContainerNeedsNameInvOCL == null) {
      EOperation eOperation = (EOperation)
eClass().getEOperations().get(0);
      Environment env =
ExpressionsUtil.createClassifierContext(eClass());
      EAnnotation ocl =
eOperation.getEAnnotation(OCL_ANNOTATION_SOURCE);
      String body = (String) ocl.getDetails().get("invariant");

      try {
            INVARIANT_MessageContainerNeedsNameInvOCL =
               ExpressionsUtil.createInvariant(env, body, true);
      } catch (ParserException e) {
            throw new
UnsupportedOperationException(e.getLocalizedMessage());
      }
}

  Query query =
  QueryFactory.eINSTANCE.createQuery(INVARIANT_MessageContainerNeeds
NameInv OCL);
  EvalEnvironment evalEnv = new EvalEnvironment();
  query.setEvaluationEnvironment(evalEnv);
  if (!query.check(this)) {
        .
        .
  }
return true;
}
```

Code Snippet: MessageContainerImpl.java

The following sections in this chapter discuss the steps taken in implementing the EMFT OCL Framework with our GMF implementation in validating the Domain Model Instance created using our generated graphical diagram editor.

4.3.1 Prerequisites

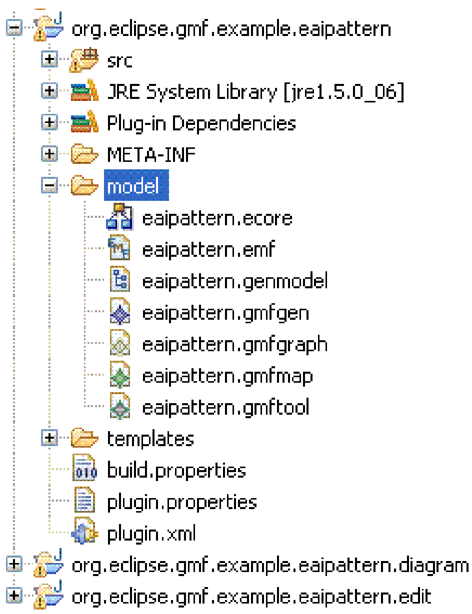


Figure 17: GMF Project Layout

We create a new GMF project `org.eclipse.gmf.example.eaipattern`. We add a templates folder to it. This project will have the Ecore model, `genmodel`, and custom JET templates as shown in Figure 17.

Now we create the Eaipattern model. Find the constraints implemented as annotations in the ecore model. Use the GMF dashboard shown in Figure 18 to create `eaipattern.genmodel`. Other definition files like `gmfgraph`, `gmftool`, `gmfmap` and `gmfgen` can be created with the dashboard.

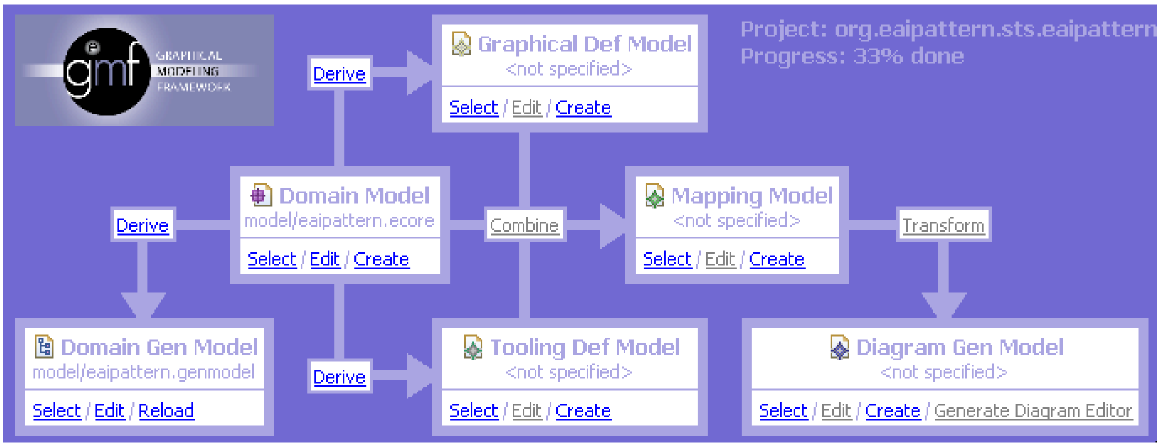


Figure 18: GMF Dash Board

Within the generated genmodel editor we change the `"Base Package"` property for the genmodels's Eaipattern to `org.eclipse.gmf.example` as shown in Figure 19. This will help to generate packages matching with the project name. In genmodel editor, enable dynamic generation templates and specify the `templates/` directory as shown in Figure 20.

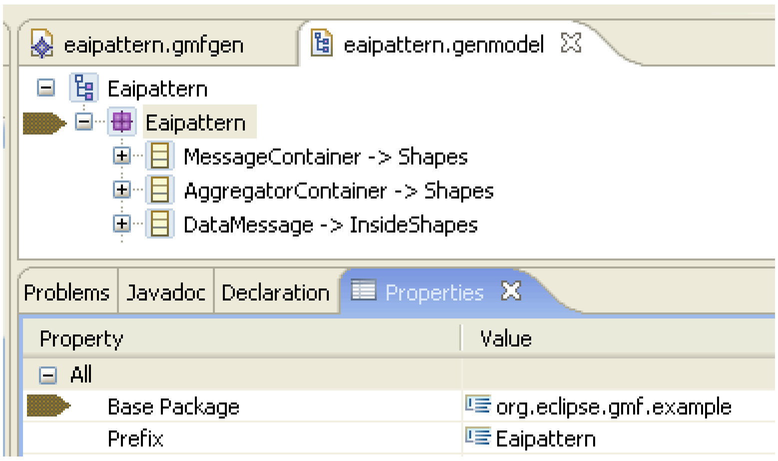


Figure 19: Base Package Properties

It is mandatory to add `org.eclipse.emf.ocl` plug-in as a Model Plug-in Variable. Further, edit the `class.javajet's` package attribute that can be found in `Project:\templates\model\Class.javajet` and point it to the location where the templates are located. We add "`org.eclipse.gmf.example.eaipattern.templates.model`" as its value.

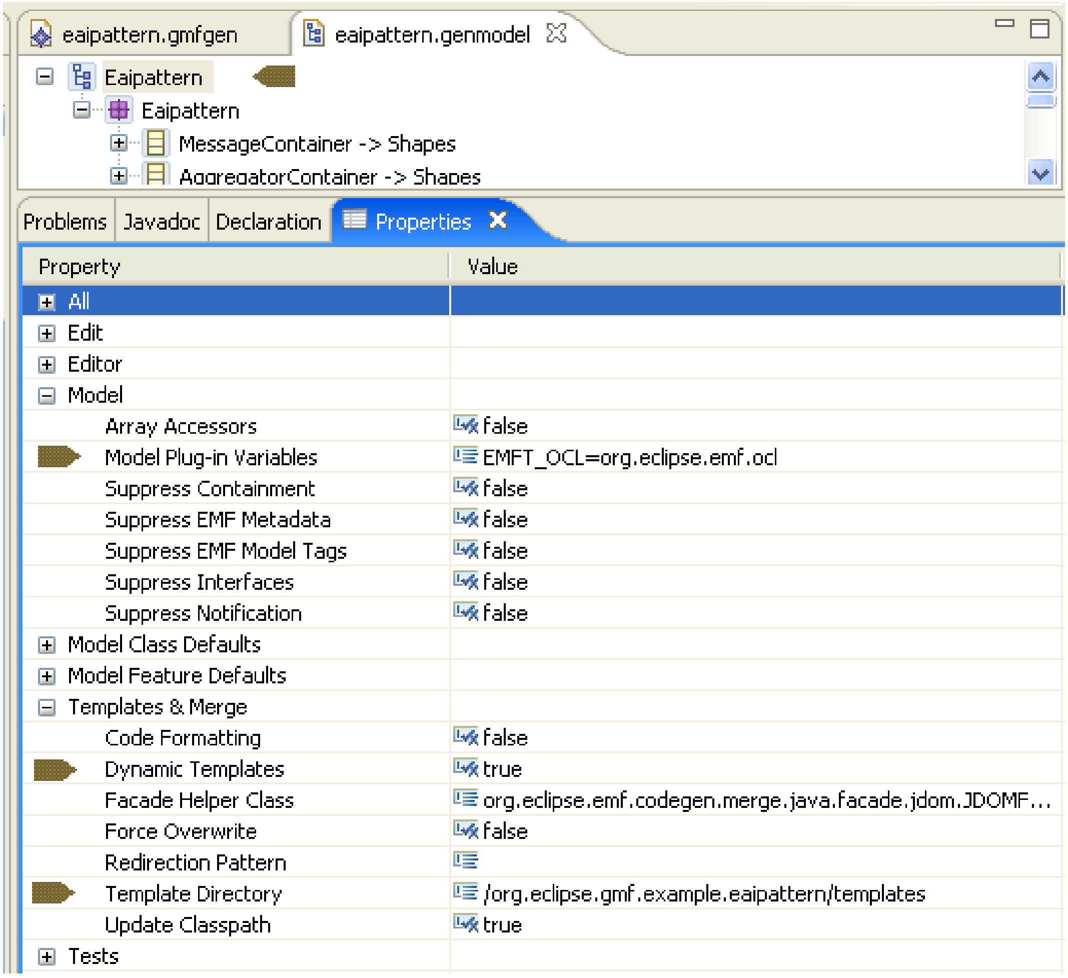


Figure 20: Enabling Templates

4.3.2 Further Steps to invoke the constraints from Ecore in GMF:

In `gmfgen`, Select `Gen Diagram` element and select `Properties Menu` with a right-click on it. Enable `Validation Decorators` and `Validation Enabled` to `true` as shown in Figure 21. Further change the `Validation Decorator Provider Priority` and `Validation Provider Priority` to any value other than `Lower` and `Low`. `We` have chosen `Medium` as priority value.

In the model, we have compartment elements to add other figure elements into the containers. To accomplish the same, I require a different layout other than the default layout provided by GMF. The GMF displays all elements placed in the container with a default `List Layout`. Setting the `List Layout` in `Diagram Compartment` to `false` will make the Layout as `XY Layout` with which the elements can be rearranged within the Compartment.

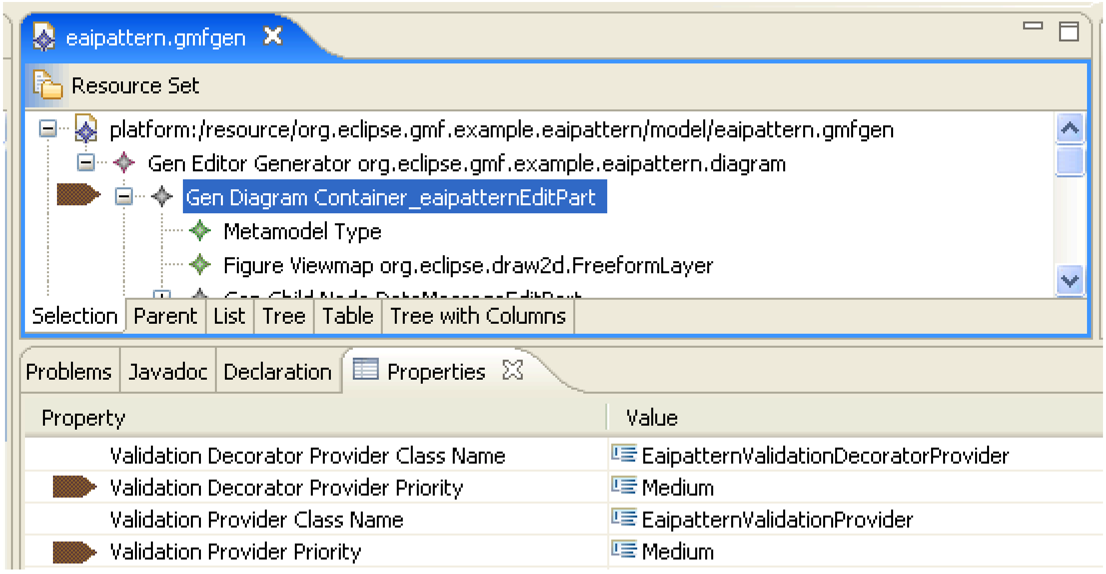


Figure 21: Validation Decorator and Provider Priority – Gmfgen

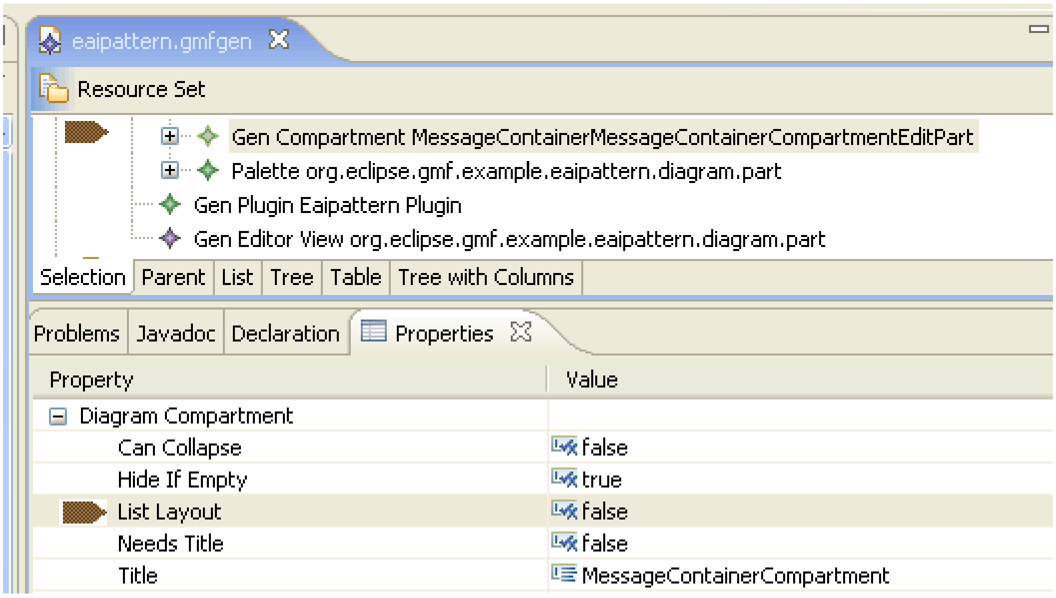


Figure 22: Enabling XY Layout

4.3.3 Enabling OCL Console for GMF

OCL Console is provided as an example for EMFT OCL Technology. With this console we can test our constraints on the domain model instances created with EMF. Such a console can be used to validate Constraints against the domain model instances specified with GMF. This will help to write constraints without ambiguities and thereby can add them at ecore level to reflect the desired well-formedness.

To have the Console running, find the `editorId` of the diagram plugin for `org.eclipse.ui.editors` extension point. In our case, the `editor id` would be `"org.eclipse.gmf.example.eaipattern.diagram.part.EaipatternDiagramEditorID"`.

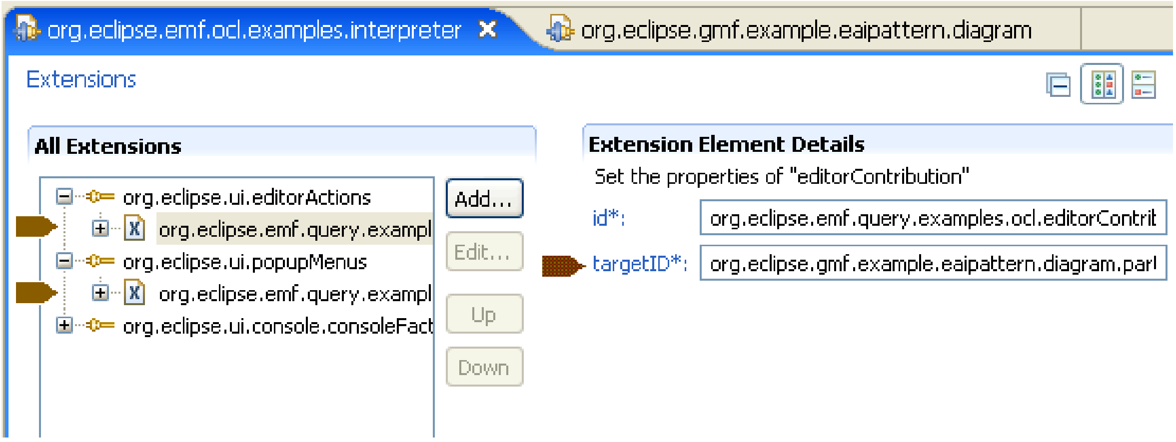


Figure 23: Identifying targetID in Diagram Editor Plugin

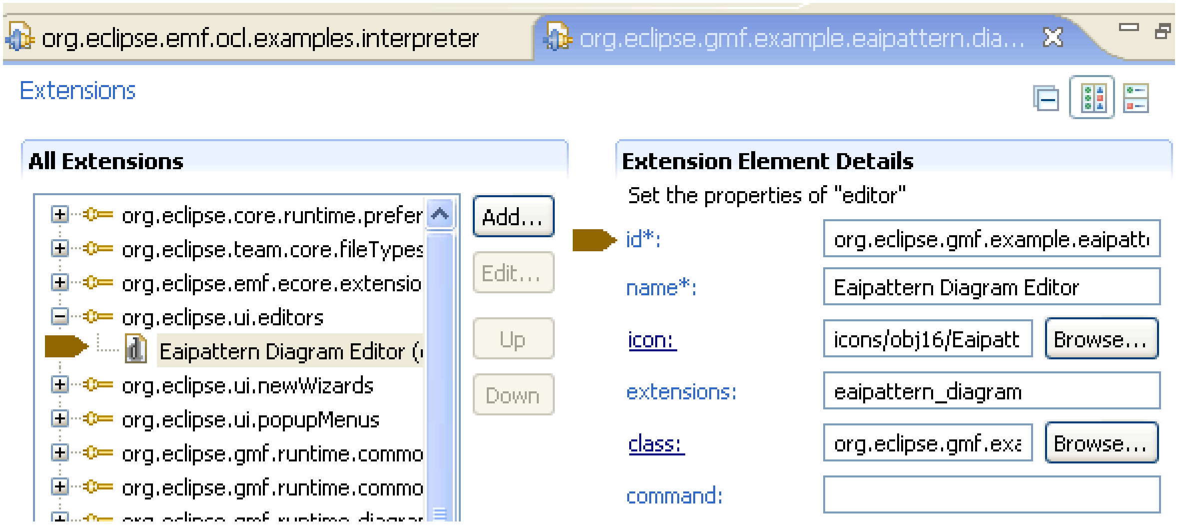


Figure 24: Specifying targetID in OCL Interpreter plugin

Import the `org.eclipse.emf.ocl.examples.interpreter` plug-in into the workspace. Within its `plugin.xml`, search for extension point of `org.eclipse.ui.editorActions` and point its `targetId` attribute to the `editorId` of our diagram plugin (i.e. `org.eclipse.gmf.example.eaipattern.diagram.part.EaipatternDiagramEditorID`). Further, search for extension point of `org.eclipse.ui.popupMenus` and point its `targetId` attribute to the `editorId` of the diagram plugin.

Include the `org.eclipse.emf.ocl.examples.interpreter` plugin with our GMF project and launch a new run-time workspace. After constructing the domain model instance diagram using the generated editor, it can be tested with OCL constraints by specifying them in OCL console with selecting the context diagram on which the constraint should be tested for. In the below diagram we specify the constraint on `MessageContainer` and test if each of the `header` instance have 1:1 relationship with `dataMessage` instance. This specified constraint evaluates to `false`.

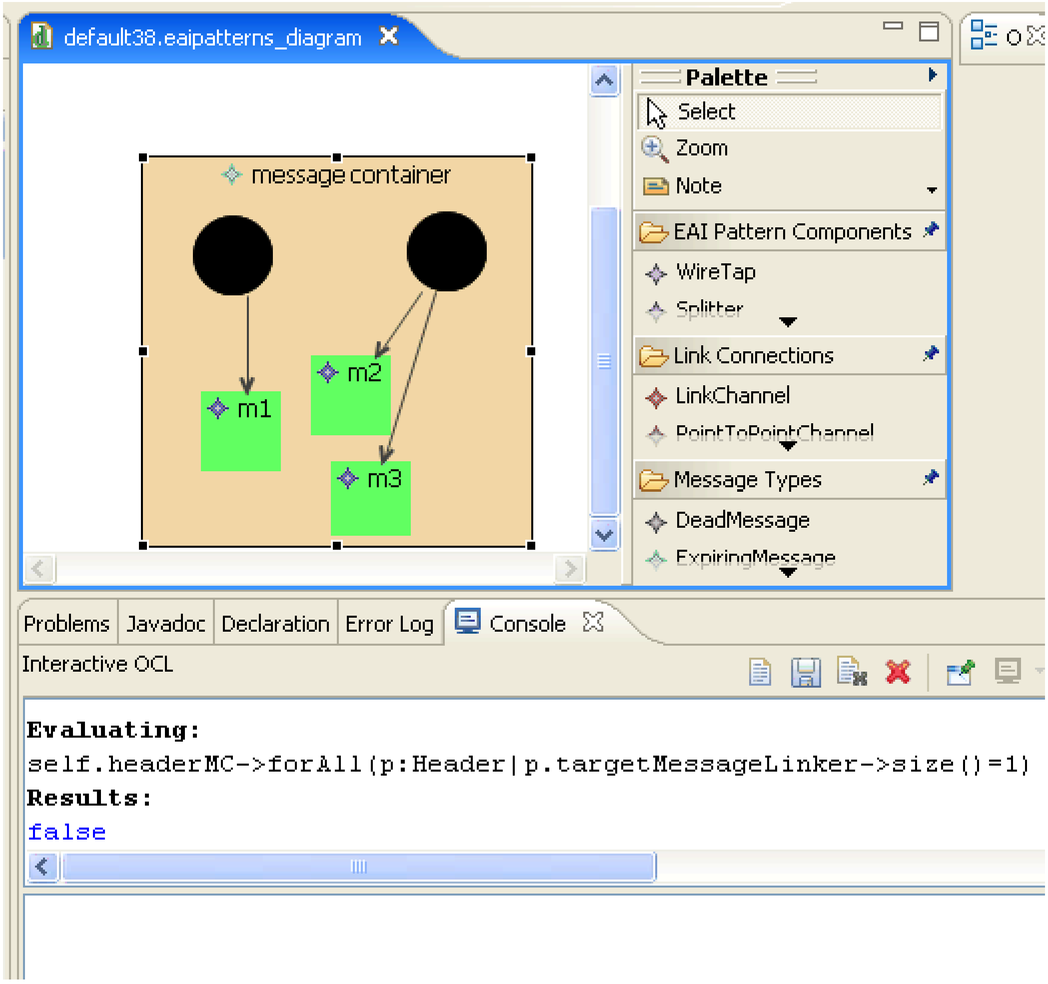


Figure 25: OCL Console

The OCL Console can be opened by selecting the `Interactive OCL` from the `Console View`'s action bar as shown in Figure 26.

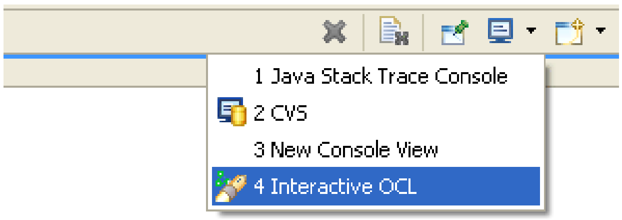


Figure 26: Enabling Interactive OCL

4.3.4 Validating Diagram Editor

Now after generating the diagram plugin for our project, launch a new runtime workbench and test the diagram. After laying out the domain elements by selecting them from the Tool Palette, the diagram can be validated with navigating to **`FileMenu`** → **`Diagram`** → **`Validate`**. This action invokes the appropriate invariant methods belonging to the domain models elements used in creating the diagram model instance on the canvas. Invariants available as annotations are taken as input by the invariant methods which are parsed and evaluated. The invariants that are broken while validating them against the model instance are displayed as Errors in the Problems Explorer.

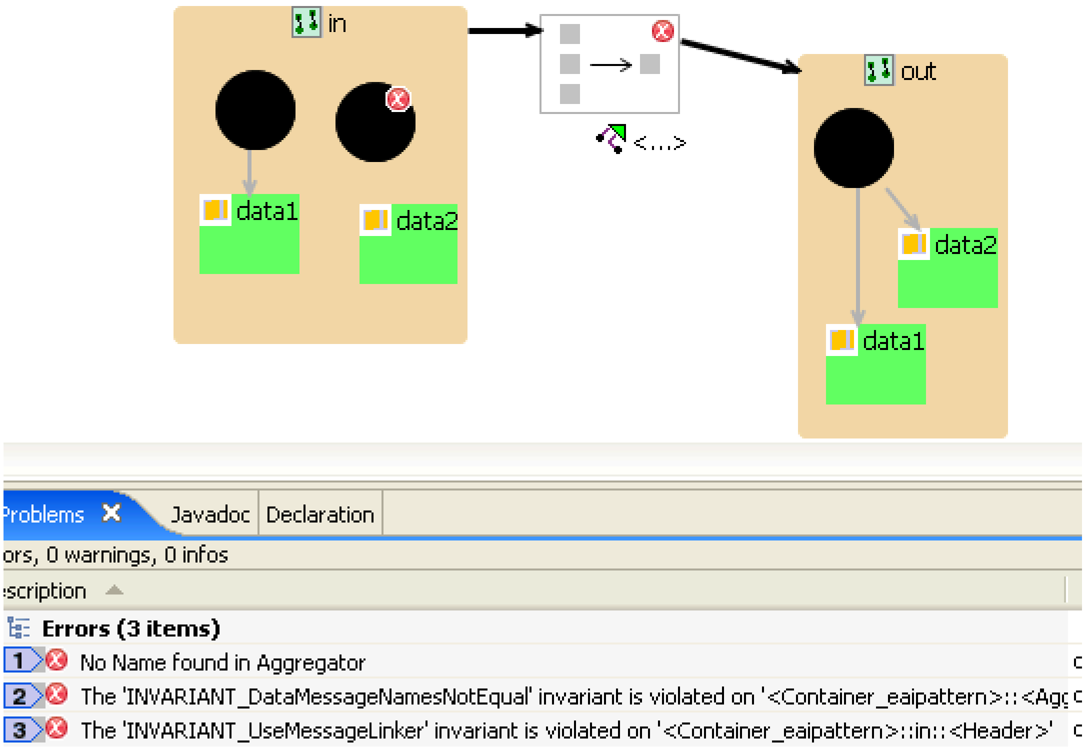


Figure 27: Diagram Editor for AggregatorContainer

1 in Figure 27 shows the invariant broken specified on the context of `AggregatorContainer` that checks for the presence of name. The Error console displays the text that is included as Message Property while including this constraint as an Audit Rule. 2 In Figure 27 displays the broken invariant specified on the context of AggregatorContainer. This constraint is included in the domain model i.e. ecore. This constraints checks for the `DataMessages` name to be identical in both the message containers. 3 In Figure 27 displays the broken invariant specified on the context of Header model element. It checks for the presence of a connection between the `Header` and its `DataMessage`.

4.4 Summary

We have discussed the role played by some EMF Technologies (namely OCL and Validation) in developing and extending the functionalities of EMF to GMF. We discussed the implementation of model integrity in domain level that can be reflected in the diagram editors generated with GMF using the EMFT Technologies.

We have discussed the step by step usage of custom JET templates and their purpose in validating the domain model instances. In the next chapter, we will look at OCL constraints that maintain the model integrity of Enterprise Application Integration patterns.

5 Enterprise Integration Patterns – Case Study

In this chapter we will be discussing the patterns which are considered to be represented as the domain model. Each pattern will be explained with the constraints that implement the validation of their domain model instances along with their respective UML diagrams. It is these constraints that will be available in the domain model to enforce domain model integrity.

5.1 Overview

Enterprise Integration is a complex field, and there is no simple answer. The patterns provide a useful way to convey experience that is gained through experience by the architects. Patterns are accepted solutions for recurring problems within a certain context. They work with most integration technologies, but specific enough to provide hands-on guidance to designers and architects. Patterns also provide vocabulary for developers to efficiently describe their problem [2].

Enterprise Integration Patterns help integration architects and developers design and implement integration solutions more rapidly and reliably. The patterns discussed in [2] are not tied to any specific implementation. The total number of patterns identified counts to 65. The patterns are organized in the following categories [2].

1. **Integration Styles** documents the different ways applications can be integrated.
2. **Channel Patterns** describe the fundamental attributes of a messaging system. These patterns are implemented by most commercial messaging systems.
3. **Message Construction Patterns** describe the intent, form and content of the messages that travel across the messaging system. The base pattern for this section is the Message pattern.
4. **Routing Patterns** discuss mechanisms to direct messages from a sender to the correct receiver. Message routing patterns consume messages from one channel and republish the message to another channel that is determined by a varying set of conditions.
5. **Transformation Patterns** change the information content of a message. In many cases, a message format needs to be changed due to different data formats used by the sending and the receiving system.
6. **Endpoint Patterns** describe the behavior of messaging system clients. They illustrate different ways in which applications can produce or consume messages.
7. **System Management Patterns** provide the tools to keep a complex message-based system running. The solution has to deal with error conditions, performance bottlenecks and changes in the participating systems. Message management patterns address these requirements.

5.2 Enterprise Integration Metamodel

In this section we will be looking at subsets of our model that will show how the model subsets are going to represent the patterns for the diagram editor.

The following steps are taken in denoting the components that are being described in text with respect to the graphical perspective.

1. All the graphical model elements that can be placed in `MessageContainer` inherit from `Class InsideShapes`.
2. **`MessageContainer`** – The messages to be represented as input to a graphical component or as an output from a pattern or to represent an intermediate state among patterns are represented with `Class MessageContainer`. With reference to Figure 28, MessageContainer has containment relationship with `Header`, `DataMessage`, `ExpiredMessage`, `InvalidMessage`, `EnrichMessage`, `DeadMessage` and `ExpiringMessage`. This means that the messagecontainer will have a containment association to all the above components. Such containers are modeled as `Compartments` during the graphical definition of the diagram editor. Each message that is represented in our graphical editor will have a `Header` and one or more different siblings of `Header`. The `Class MessageLinker` creates link connections between the domain elements of `Class InsideShapes`. There exist two association relationships between the `MessageLinker` and `InsidesShapes` that is navigable in both directions. With the first association, we see `srcInsideShapes EReference` in `MessageLinker` and `targetMessageLinker EReference` in `InsideShapes`. In the second association, we see `targetInsideShapes EReference` in `MessageLinker` and `srcMessageLinker` in `InsideShapes`.
3. The `Class MessageLinker` is modeled as connection link between the `Header` and the other siblings of `Header`.
4. The constraints below are specified for the `MessageContainer` model element.
   - The constraint below checks if the `name` attribute of `MessageContainer` is specified.

     ```
     context MessageContainer
        inv MessageContainerNeedsName:
           self.name.size()>0
     ```

   - The constraint below is specified on the context from `Class Header`. The constraint checks that if a `Header` instance is created then the `Class MessageLinker` is to be used from the Tool Palette to create a connection starting with the `Header` as the source.

     ```
     context Header
        inv UseMessageLinker:
           self.targetMessageLinker->size()>0
     ```

   - The constraint below check that atleast one header should be specified for an instance of `Class MessageContainer`.

```
context MessageContainer
    inv AtleastOneHeader:
        self.headerMC->size()>0
```

- The constraint below checks that if a `Header` element is created then the `DataMessage` element should also be present.

```
context MessageContainer
    inv NeedDataMessageIfHeaderPresent:
        (self.headerMC->exists(oclIsTypeOf(Header))
         implies
             self.headerMC.targetMessageLinker.
             targetInsideShapes->isEmpty()
```

5. `Class LinkChannel` provides link connections among domain model elements that could not be represented with traditional messagechannels like `MessageChannel`, `Point-To-Point Channel`, `Publish – Subscriber Channel` etc.

The below are the list of patterns that would be discussed and implemented using our metamodel in designing the graphical diagram editor.

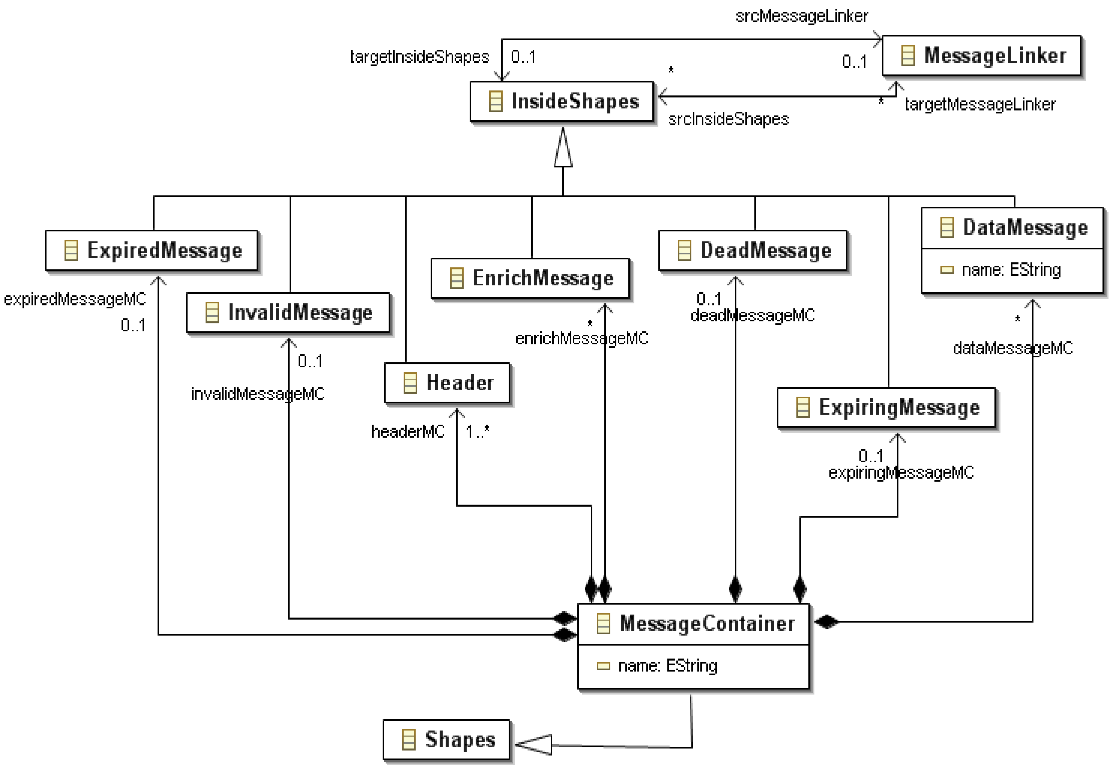


Figure 28: MessageContainer - Class Diagram

5.2.1 Message Channel Pattern

Connect the applications using a MessageChannel, where one application writes information to the channel and the other one reads that information from the channel [2].

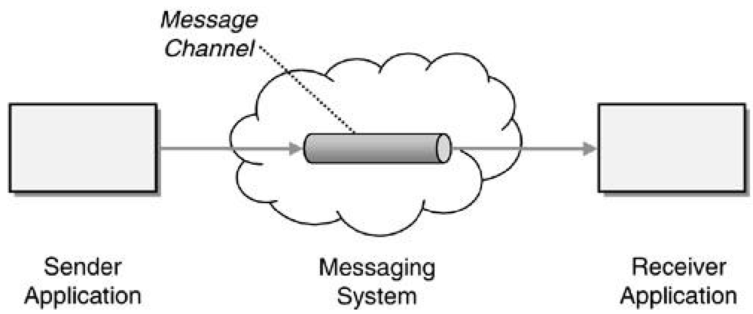


Figure 29: Message Channel Pattern

The subset of our domain model representing Figure 29 is shown in UML Figure 30. In this model `SenderContainer` represents Sender Application, `ReceiverContainer` represents `ReceiverApplication` and `MessageChannel` represents the connection (i.e. the messaging system). Message Channel, represented by `Class MessageChannel` creates connection link between a `SenderContainer` element and a `ReceiverContainer` elements. To create a link connection between these domain model elements say, we create associations between `MessageChannel` to `SenderContainer` and between `MessageChannel` to `ReceiverContainer` that is navigable in both directions. With the association between `MessageContainer` and `SenderContainer` we see the

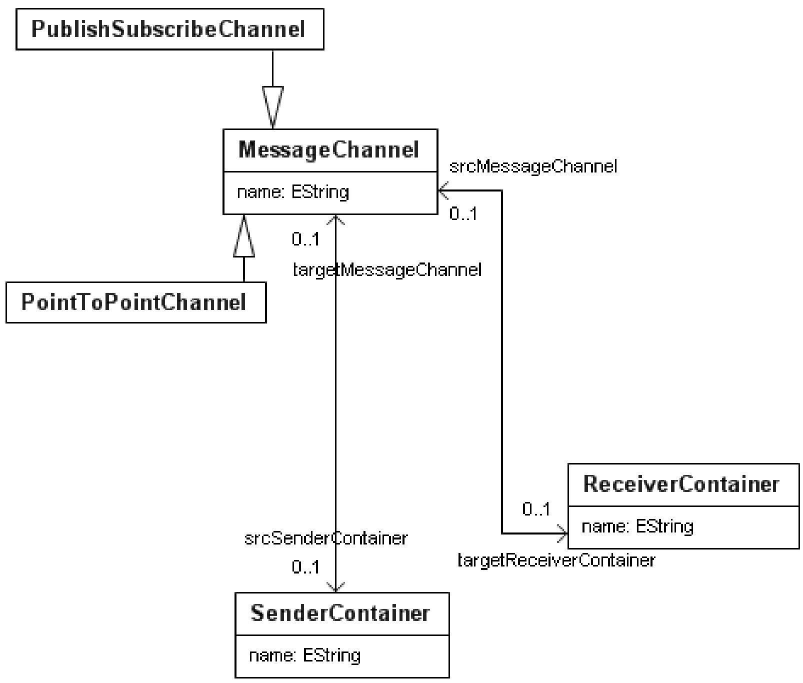


Figure 30: Message Channel Pattern - Class Diagram

`srcSenderContainer` `EReference` in `MessageChannel` and `targetMessageChannel` `EReference` in `SenderContainer`. With the association between MessageContainer and ReceiverContaner, we see the targetReceiverContainer EReference in MessageChannel and the srcMessageChannel EReference in ReceiverContainer.

MessageChannel has two derived classes, namely Publisher Subscriber Channel and Point-To-PointChannel. They are represented by Class PublishSubscribeChannel and Class PointToPointChannel respectively.

5.2.2 Aggregator Pattern

> *Use a stateful filter, an Aggregator, to collect and store individual messages until it receivers a complete set of related messages. Then, Aggregator publishes a single message distilled from the individual messages [2].*

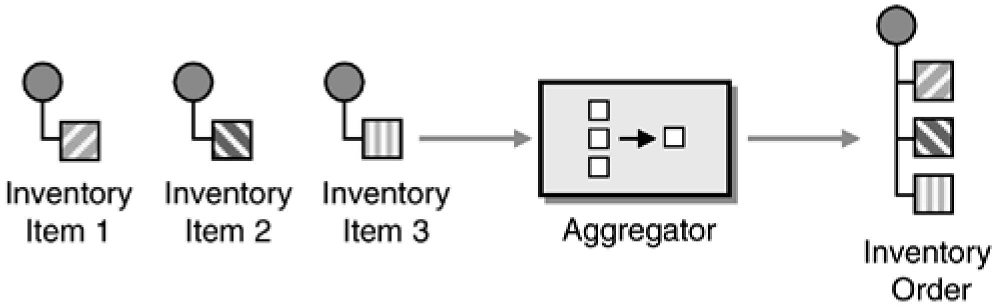


Figure 31: Aggregator Pattern

The subset of our domain model representing Figure 31 is shown with UML in Figure 32. Aggregator is represented by AggregatorContainer. AggregatorContainer and MessageContainer inherit from Shapes. This inheritance hierarchy gives out two benefits. This helps to classify the domain elements that have reference to LinkChannel. Another is to reduce the number of Link mapping definitions that are requried to represent each individual mapping between domain elements.

In Figure 32, MessageContainer is used to represent a compartment for placing input messages and output messages. Each message consists of a Header and a Message. With reference to the above class diagram Header and DataMessage are two derived classes of InsideShapes. We would be looking at other derived classes of InsideShapes in the following sections. MessageContainer has a containment association with Header and DataMessage. Link Channel, represented by Class Link Channel creates connection between Class Shapes. To create a link connection representing Link Channel between Shapes we create two associations between LinkChannel and Shapes that is navigable in both directions. In one association we can see srcShapes EReference from LinkChannel and targetLinkChannel EReference from Shapes. In the other association we can see targetShapes EReference from LinkChannel and srcLinkChannel EReference from Shapes.

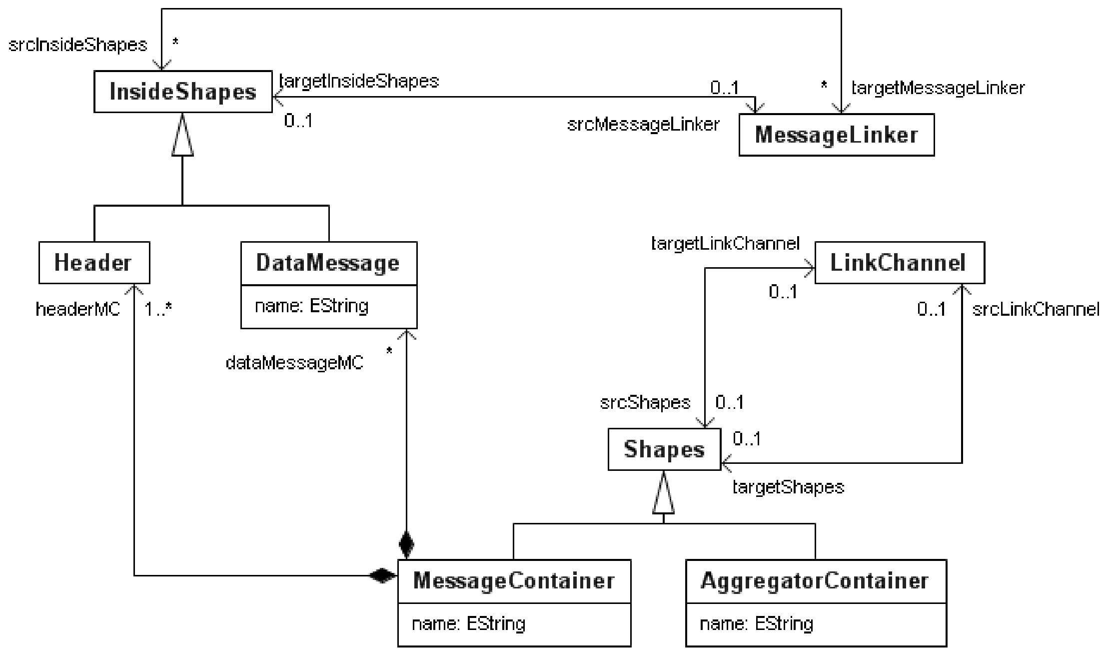


Figure 32: Aggregator Pattern - Class Diagram

Usually to create a link between two classes, the linking classes will have uni-directional association with those two classes. But in the case of enterprise patterns there would be a need to navigate through association from the particular domain model's context to verify the well-formedness of the domain model instance. This requires us to have a bi directional association when it comes to the classes that provide Linking between the model elements.

5.2.2.1 Constraints

The constraints below help in checking the well-formedness of the `Aggregator` Pattern,

1. The constraint below checks if the `name` attribute for `Aggregator` is specified.

```
context AggregatorContainer
  inv AggregatorNameNotPresent: self.name.size()>0
```

2. The constraint below checks that the number of `Header` instances in the input MessageContainer should be more that the number of `Header` instances in the output `MessageContainer`. We create some check conditions for the constraint to work. This constraint works only when the connection link is created between Input MessageContainer to `AggregatorContainer` and from `AggregatorContainer` to Output MessageContainer.

```
context AggregatorContainer
  inv InputOutputMessagesNotEqual:
      not ( srcLinkChannel->isEmpty() )
            and
      not ( targetLinkChannel->isEmpty() )
            implies
      self.srcLinkChannel.srcShapes.
            oclAsType(MessageContainer).headerMC->size() >
      self.targetLinkChannel.targetShapes.
            oclAsType(MessageContainer).headerMC->size()
```

3. The Constraint below checks that atleast one `Header` instance must be placed in the input Message Container.

```
context AggregatorContainer
   inv AtleastOneHeaderInInputContainer:
      not (srcLinkChannel->isEmpty()) implies
      self.srcLinkChannel.srcShapes.
            oclAsType(MessageContainer).headerMC->size()>=1
```

4. The Constraint below checks the `name` attributes specified for the data messages in the input message container to be equal to the name attributes of data messages in the output message container. This constraint is to enforce that the same messages are being created in both the message containers.

```
context AggregatorContainer
   inv DataMessageNamesEqual:
      self.srcLinkChannel.srcShapes.
            oclAsType(MessageContainer).headerMC.
            targetMessageLinker.targetInsideShapes.
            oclAsType(DataMessage).name->asBag() =
      self.targetLinkChannel.targetShapes.
            oclAsType(MessageContainer).headerMC.
            targetMessageLinker.targetInsideShapes.
            oclAsType(DataMessage).name->asBag()
```

5. The Constraint below checks that atleast one `Header` instance should be created in Output Message Container.

```
context AggregatorContainer
   inv OnlyOneHeaderInOutputContainer:
      not (targetLinkChannel->isEmpty()) implies
      self.targetLinkChannel.targetShapes.
           oclAsType(MessageContainer).headerMC->size()=1
```

6. The constraint below checks that each message in the input message container for the receiver container should have 1:1 relationship between the `Header` instance and `DataMessage` instance.

```
context AggregatorContainer
   inv OneHeaderWithOneDataMessageInput:
      not (srcLinkChannel->isEmpty()) implies
      self.srcLinkChannel.srcShapes.
           oclAsType(MessageContainer).headerMC->
           forAll(p:Header| p.targetMessageLinker
           ->size()=1)
```

5.2.3 Content Filter Pattern

Use a Content Filter to remove unimportant data items from a message, leaving only important items [2].

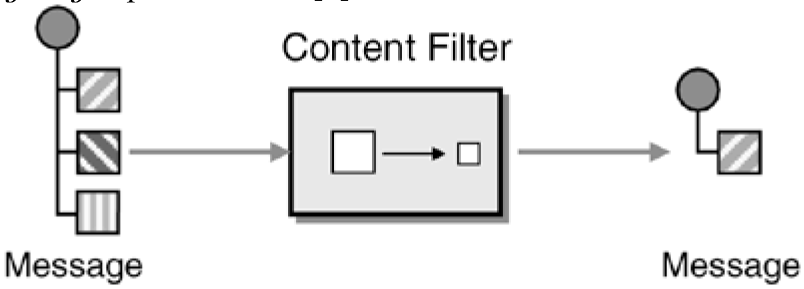


Figure 33: Content Filter Pattern

The subset of the domain model representing Figure 33 is shown with UML in Figure 34. The `Content Filter Pattern` is represented by `Class ContentFilter`. `ContentFilter` has aggregation of `DataMessage` because it is a container that must have a datamessage that has to be filtered from the rest of the messages. This is accomplished by having an containment association with the `DataMessage` with a rolename `dataMessageToBeFiltered`. The requirement of making the `DataMessage` to be present can be enforced with a constraint or with association from the content filter to `DataMessage` with `lowerbound = 1 and upperbound = 1` instead of `lowerbound = 0` and `upperbound = 1`. The templates implemented with EMFT OCL framework will generate validation errors when the above association fails as well.

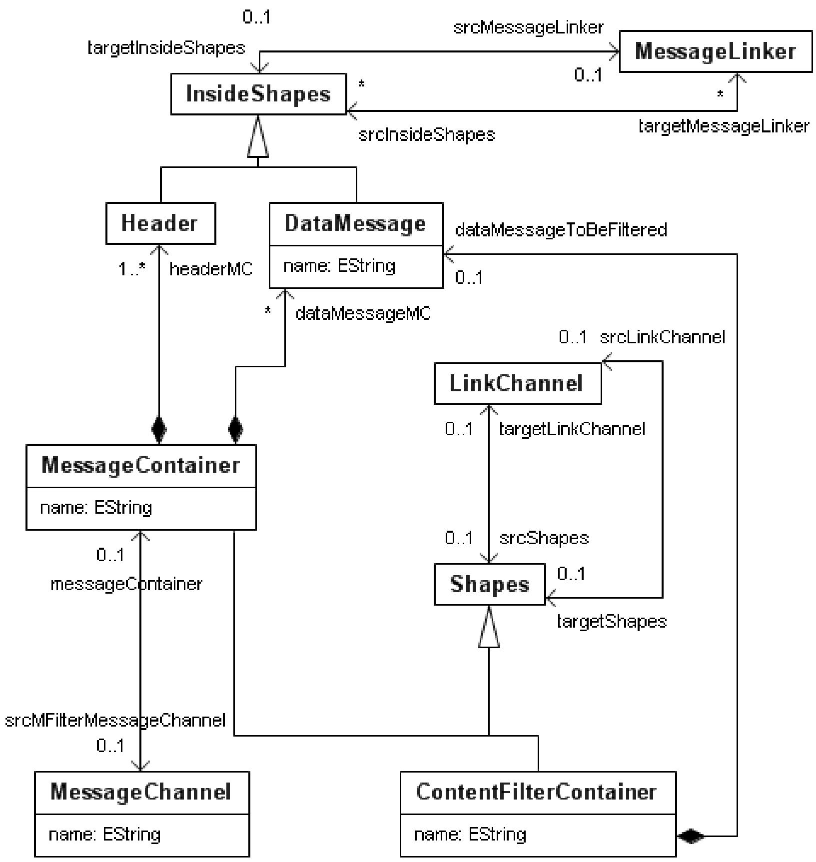


Figure 34: Content Filter - Class Diagram

MessageContainer is used to represent the input and output messages for the content filter pattern.

5.2.3.1 Constraints

The constraints below help in checking the well-formedness of the Content Filter Pattern,

1. The constraint below checks if the name attribute for Content Filter is specified.

```
context ContentFilterContainer
  inv ContentFilterNameNotPresent: self.name.size()>0
```

2. The constraint below checks that the data message to be filtered should be present.

```
context ContentFilterContainer
   inv MessageToBeFilteredForNotPresent:
      self.dataMessageToBeFiltered->
           exists(oclIsTypeOf(DataMessage))
```

3. The constraint below checks the data messages to be filtered should be equal to the data messages present in the output Message Container.

```
context ContentFilterContainer
   inv MessageBeFilteredNotEqualToOutputMessage:
      self.dataMessageToBeFiltered.name->asBag() =
      self.targetLinkChannel.targetShapes.
           oclAsType(MessageContainer).headerMC.
           targetMessageLinker.targetInsideShapes.
           oclAsType(DataMessage).name->asBag()
```

4. The constraint below checks the collection of data messages present in the output Message Container to be present in the collection from the datamessages present in the input message container.

```
context ContentFilterContainer
   inv ContentFilterInvalid:
      not (srcLinkChannel->isEmpty()) and
      not (targetLinkChannel->isEmpty()) and
      (self.targetLinkChannel.targetShapes.
          oclAsType(MessageContainer).
       headerMC.targetMessageLinker->size()>0) implies
      self.srcLinkChannel.srcShapes.
          oclAsType(MessageContainer).headerMC.
                  targetMessageLinker.targetInsideShapes.
          oclAsType(DataMessage).name->
      includesAll(self.targetLinkChannel.targetShapes.
          oclAsType(MessageContainer).headerMC.
                  targetMessageLinker.targetInsideShapes.
          oclAsType(DataMessage).name)
```

5.2.4 Splitter Pattern

Use a Splitter to break out the composite message into a series of individual messages, each containing data related to one item. [2]

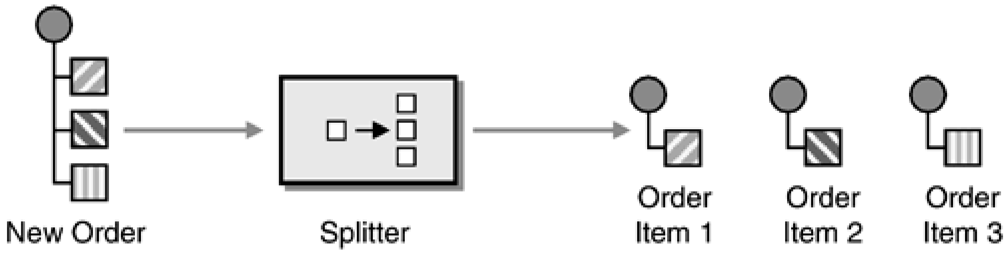


Figure 35: Splitter Pattern

The subset of domain model representing Figure 35 is shown with UML in Figure 36. The Splitter is represented by `Class Splitter`. `Class Splitter` and `Class MessageContainer` are the derived classes of `Class Shapes`. The input messages and output messages are placed in the `MessageContainer` which make the later to have an composite association with `Header` and `DataMessage`.

Link Channel is represented by `Class Link Channel` which represents the connection between `MessageContainer` and `Splitter`. To create a link connection between the `MessageContainer` instance that becomes an input message container and `Splitter`, we create association between them that is navigable in both directions. With the defined association we can see the `srcShapes EReference` in `LinkChannel` and `targetLinkChannel EReference` in `Shapes`. For creating a link connection between the `Splitter` and the MessageContainer Instance that becomes the output MessageContainer we create

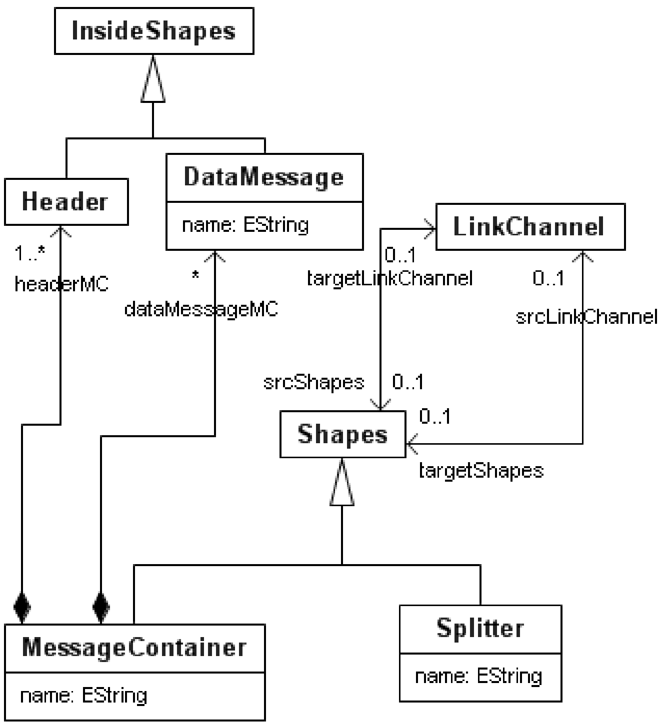


Figure 36: Splitter Pattern - Class Diagram

another association between them where we see the targetShapes in LinkChannel and srcLinkChannel in Shapes.

5.2.4.1 Constraints

The following constraints are considered to enforce the well-formedness of Splitter Pattern.

1. The constraint below checks if the Header instances created in input MessageContainer is equal to the number of DataMessage instances created in output MessageContainer.

```
context Splitter
   inv SplitterInvalid:
      not(srcLinkChannel->isEmpty()) and
      not (targetLinkChannel->isEmpty()) implies
      self.srcLinkChannel.srcShapes.
         oclAsType(MessageContainer).headerMC.
            targetMessageLinker.targetInsideShapes.
         oclAsType(DataMessage)->size() =
      self.targetLinkChannel.targetShapes.
         oclAsType(MessageContainer).dataMessageMC->size()
```

2. The constraint below checks if the name attributes of DataMessage instances in input MessageContainer and is same as the name attributes of Data Message instances present in output MessageContainer.

```
context Splitter
   inv SplitterMessageTypesInvalid:
      not(srcLinkChannel->isEmpty()) and
      not(targetLinkChannel->isEmpty()) implies
      self.srcLinkChannel.srcShapes.
         oclAsType(MessageContainer).headerMC.
            targetMessageLinker.targetInsideShapes.
         oclAsType(DataMessage).name->asBag() =
      self.targetLinkChannel.targetShapes.
         oclAsType(MessageContainer).dataMessageMC.name
         ->asBag()
```

3. The constraint below checks the messages created in the output MessageContainer to have only one DataMessage instance to each Header instance created.

```
context Splitter
   inv SplitterOutputContainerInvalidMessageLinks:
      not(targetLinkChannel->isEmpty()) implies
      self.targetLinkChannel.targetShapes.
         oclAsType(MessageContainer).headerMC->
         forAll(p:Header|p.targetMessageLinker->size()=1)
```

5.2.5 Point-to-Point channel

Send the message on a Point-to-Point Channel, which ensures that only one receiver will receiver a particular message. [2]

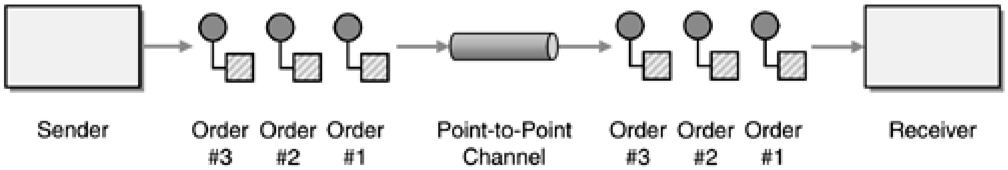


Figure 37: Point to Point Channel Pattern

The subset of domain model representing Figure 37 is shown with UML in Figure 38. `Class PointToPointChannel` represents the Point-to-Point Channel Pattern. This class is a derived class of `Class MessageChannel`.

We represent this pattern as a Connection to draw links between Message Containers. We create two associations between MessageContainers and PointToPointChannel that is navigable in both directions. With the first association between the instance of MessageContainer that is going to be formed as input MessageContainer and PointToPointChannel, we can see `srcMCPointToPointChannel EReference` from MessageContainer and `targetPTPmessageContainer EReference` from PointToPointChannel. The second association between the PointToPointChannel the instance of MessageContainer that is going to be formed as output MessageContainer,

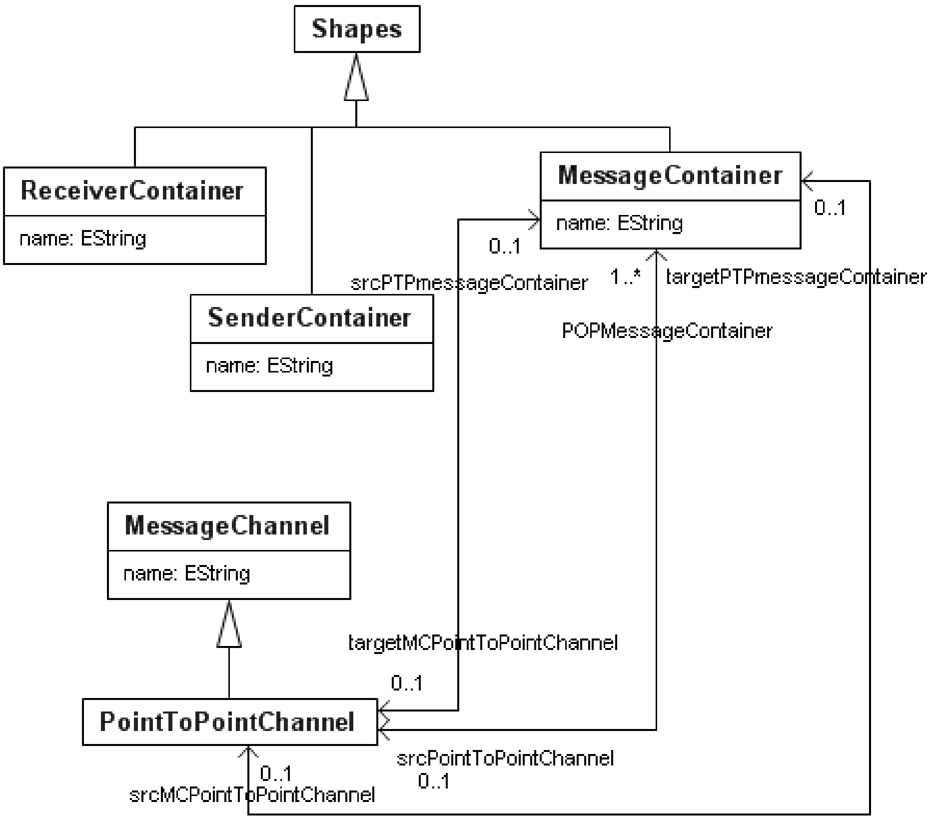


Figure 38: Point To Point Channel Pattern - Class Diagram

we can see srcPTPmessageContainer EReference in PointToPointChannel and targetMCPointToPointChannel EReference in MessageContainer.

SenderContainer and ReceiverContainer have containment association relationship with Header and DataMessage(which is not shown in Figure 38). LinkChannel is used to draw connections from the SenderContainer to the MessageContainer and from the MessageContainer to the ReceiverContainer. Their associations are discussed in previous patterns since the SenderContainer, MessageContainer and ReceiverContainer are the derived classes of Shapes.

5.2.6 Message Filter Pattern

> *Use a special kind of Message Router, a Message Filter, to eliminate undesired messages from a channel based on a set of criteria. [2]*

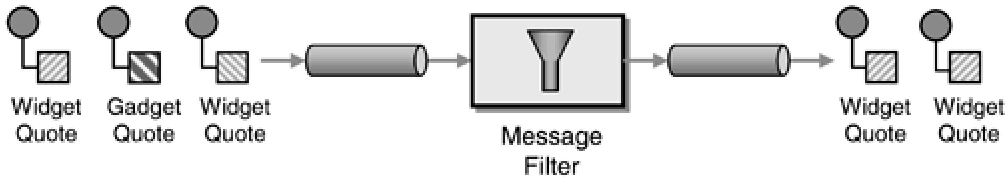


Figure 39: Message Filter Pattern

The subset of domain model representing Figure 39 is shown with UML in Figure 40. Message Filter Pattern is represented by Class MessageFitler. The messages placed in a MessageContainer would constitute the input for the MessageFilter and another instance of MessageContainer as output from MessageFilter.

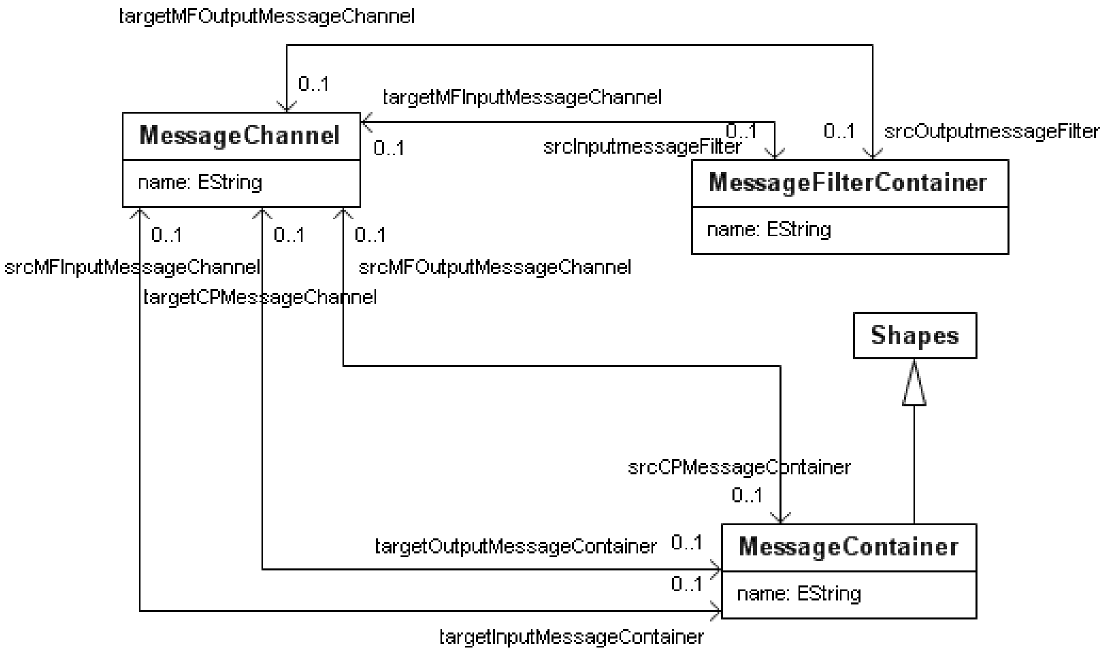


Figure 40: Message Filter Pattern - Class Diagram

MessageFilter has an aggregation relationship with DataMessage. This DataMessage is used by the MessageFilter to remove the specific message content from the input channel and send the remaining messages to the output channel.

With the graphical perspective, we will write constraints from the context of MessageFilter and check if the message to be filter is present in the InputMessageContainer and not present in the OutputMessageContainer.

5.2.6.1 Constraints

The following constraints are considered to enforce the well-formedness of Message Filter Pattern.

1. The following constraint checks that the data message to be filtered should be present in `MessageFilterContainer`.

```
context MessageFilterContainer
   inv MessageToBeFilteredNotPresent:
      self.dataMessageToFilter->size()>0
```

2. The constraint below checks that for all the messages in the input message container and output message container, each header should have only one data message linked with it using the `Class MessageLinker`.

```
context MessageFilterContainer
   inv MessagesNotWithSingleDataMessages:
      not(targetMFInputMessageChannel->isEmpty()) and
      not (targetMFOutputMessageChannel->isEmpty()) implies
      self.targetMFInputMessageChannel.
         targetOutputMessageContainer.headerMC->
         forAll(p:Header | p.targetMessageLinker->size()=1)
      self.targetMFOutputMessageChannel.
         targetInputMessageContainer.headerMC->
         forAll(p:Header | p.targetMessageLinker->size()=1)
```

3. The constraint below checks the union of name of the data message specified in the message filter with that of data messages in the output message container is equal to the list of names of data messages in the input message container.

```
context MessageFilterContainer
   inv MessageFilterInputOutputVaries:
      not(targetMFInputMessageChannel->isEmpty()) and
      not (targetMFOutputMessageChannel->isEmpty()) implies
      self.targetMFOutputMessageChannel.
         targetInputMessageContainer.
         dataMessageMC.name->
         union(self.dataMessageToFilter.name) =
      self.targetMFInputMessageChannel.
         targetOutputMessageContainer.
         dataMessageMC.name
```

5.2.7 Message Dispatcher Pattern

Create a Message Dispatcher on a channel that will consume messages from a channel and distribute them to performers [2].

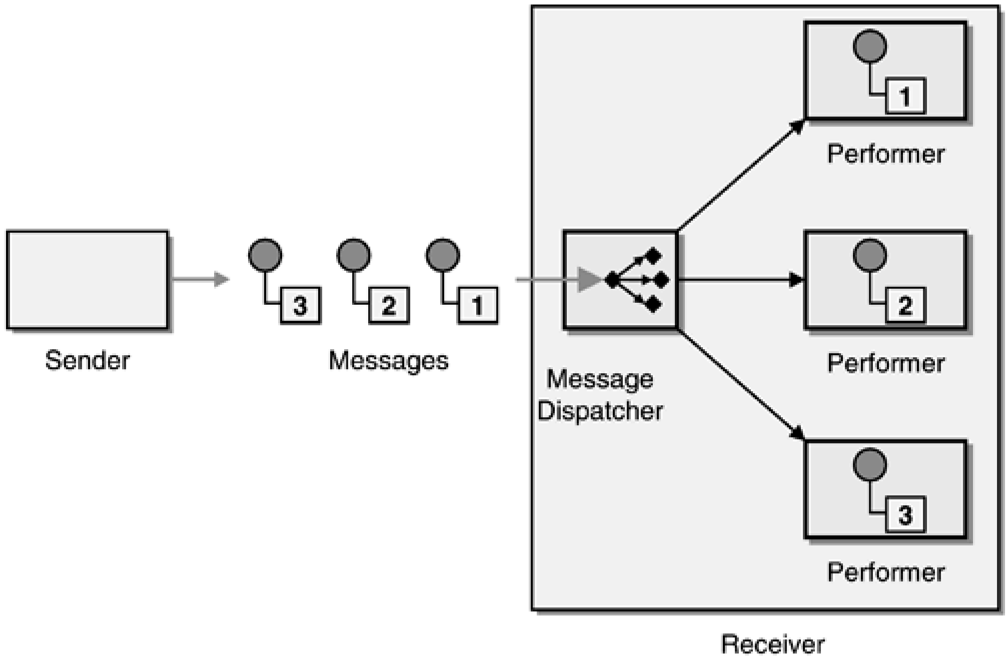


Figure 41: Message Dispatcher Pattern

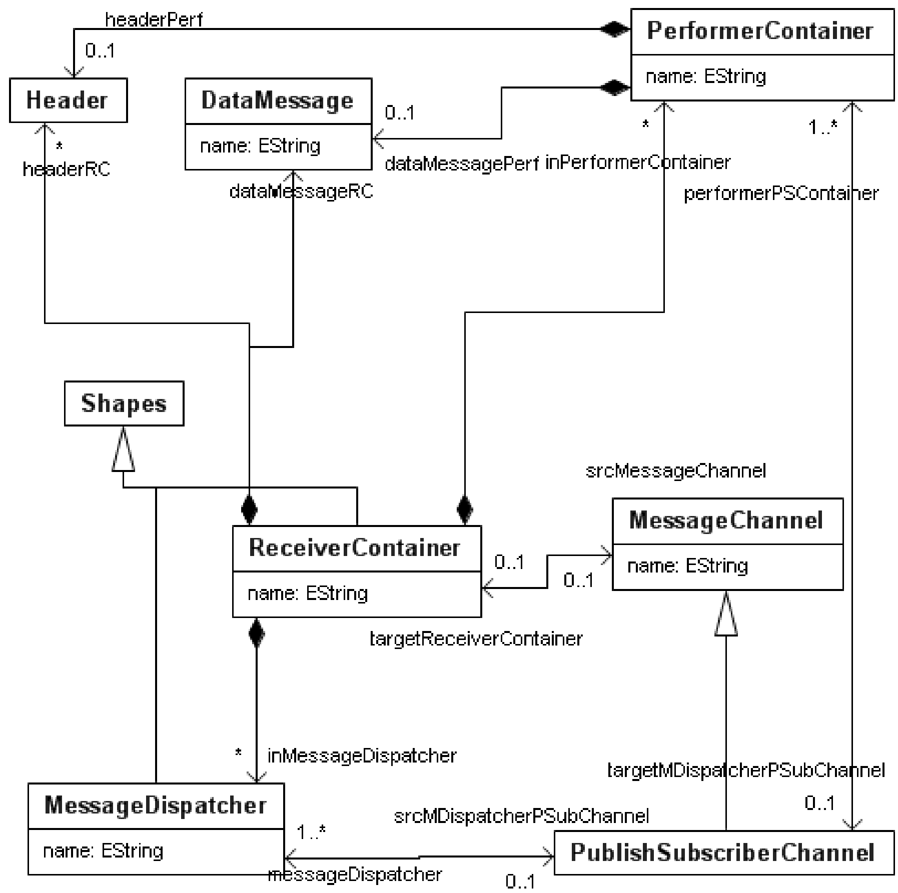


Figure 42: Message Dispatcher Pattern - Class Diagram

The subset of domain model representing Figure 41 is shown with UML in Figure 42. Message Dispatcher Pattern is represented by `Class MessageDispatcher`. Sender is represented by `Class SenderContainer`.

To represent the Receiver part of the Figure 41, the `ReceiverContainer` has containment association with `MessageDispatcher` and `PerformerContainer`. This aggregate relationship will help to make the `ReceiveContainer` have a component-part relationship with the `MessageDispatcher` and `PerformerContainer`. The latter has aggregation relationship with `Header` and `DataMessage` that will help to create a container property for `PerformerContainer` to hold both the `Header` and `DataMessage` domain model instances. Link Channel is represented by `Class LinkChannel` to create connection between an input `MessageContainer` and a `MessageDispatcher`. Since `MessageDispatcher` uses publish subscriber channel to distribute the messages to its performers, we will use `PublishSubscriberChannel` to create link connection between `MessageDispatcher` and `PerformerContainer`.

5.2.7.1 Constraints

The following constraints are considered to enforce the well-formedness of Message Dispatcher Pattern.

1. The constraint below checks the name of datamessages in the message container that is specified as input to the receiver against the names of datamessages specified collectively in each performer that has container-part relationship with the ReceiverContainer.

```
context MessageDispatcher
   inv InvalidMessageDispatcher:
      not(srcLinkChannel->isEmpty()) and
      not(srcMDispatcherPSubChannel.
         performerPSContainer->isEmpty())
      implies
         self.srcLinkChannel.srcShapes.
           oclAsType(MessageContainer).dataMessageMC.name =
         self.srcMDispatcherPSubChannel.performerPSContaine

           dataMessagePerf.name
```

2. The constraint below checks that each message in the input messagecontainer for the receiver container should have 1:1 relationship between the `Header` instance and `DataMessage` instance.

```
context MessageDispatcher
   inv
InputMessageContainerHasMessagesWithSingleDataMessages:
      not (srcLinkChannel->isEmpty()) implies
      self.srcLinkChannel.srcShapes.
         oclAsType(MessageContainer).headerMC->
         forAll(p:Header| p.targetMessageLinker->size()=1)
```

3. The constraint below checks the number of data messages present in the `MessageContainer` that is provided as input to the `ReceiverContainer` to be equal to the number of data messages present in performerContainers within `ReceiverContainer`.

PerformerContainer has containment association with DataMessage and can see the DataMessage with dataMessagePerf EReference.

```
context MessageDispatcher
   inv InvalidMessageDispatcherWithInvalidPerformers:
      not(srcLinkChannel->isEmpty()) and
      not(srcMDispatcherPSubChannel.performerPSContainer
         ->isEmpty())
      implies
         self.srcLinkChannel.srcShapes.
           oclAsType(MessageContainer).
                dataMessageMC->size() =
         self.srcMDispatcherPSubChannel.
                performerPSContainer.
                dataMessagePerf->size()
```

4. The constraint below is specified on the context of PerformerContainer. This constraint checks that only one DataMessage instance should be present in the PerformerContainer.

```
context PerformerContainer
   inv OneMessageInPerformer:
      self.headerPerf.targetMessageLinker.
         targetInsideShapes.
         oclAsType(DataMessage).name->size()=1
```

5.2.8 Invalid Message Channel Pattern

The receiver should move the improper message to an Invalid Message Channel, a special channel for messages that could not be processed by their receivers [2].

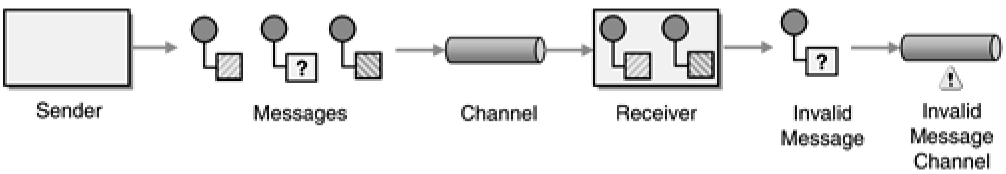


Figure 43: Invalid Message Channel Pattern

The subset of domain model representing Figure 43 is shown with UML in Figure 44. The invalid message is represented by Class InvalidMessage. It is derived class for Class InsideShapes with the Header and DataMessage.

We use message channel to create connection link between MessageContainer and ReceiverContainer. To create this connection we create association between MessageContainer to MessageChannel and MessageChannel to ReceiverContainer. These associations can be navigated in both directions. With the association from MessageContainer to MessageChannel, we see targetCPMessageChannel EReference in MessageContainer and srcCPMessageContainer EReference in MessageChannel. With the association from MessageChannel to ReceiverContainer, we see targetReceiverContainer EReference in MessageChannel and srcMessageChannel EReference in MessageChannel.

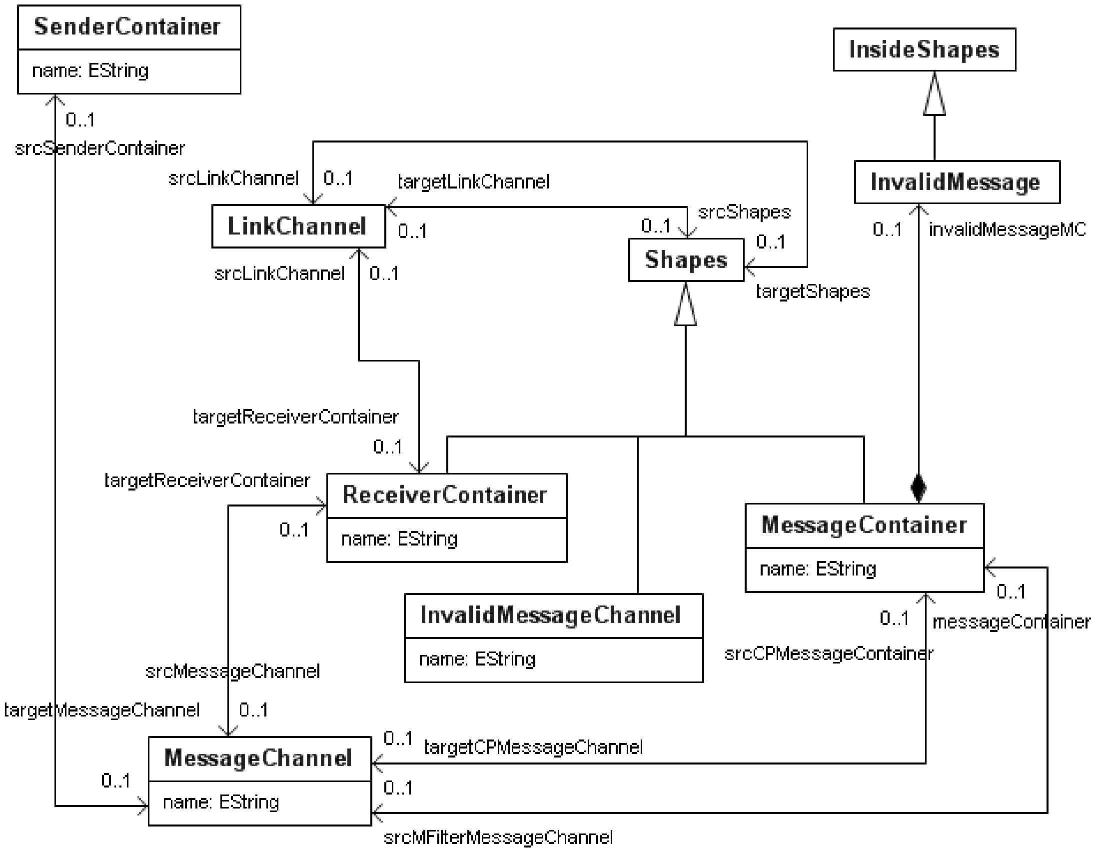


Figure 44: Invalid Message Channel Pattern - Class Diagram

We use Link Channel represented by `Class LinkChannel` to create connection link from the `SenderContainer` to `Messagecontainer` and from `ReceiverContainer` to `MessageContainer` containing the Invalid message that is sent for Invalid `MessageChannel` by the receiver. In the above case we have noticed four link mappings that has to be created while developing the diagram editor.

Classifying the classes into hierarchies of inheritance will help to reduce the number of link mappings. A LinkMapping for `LinkChannel` with `Shapes` as the source element and the target element will enable us to create connections among all children of `Shapes`. In this case, `SenderContainer`, `MessageContainer`, `ReceiverContainer` and `InvalidMessageChannel` are the children of `Shapes`. But restrictions at this level has to be imposed on link mapping through link constraints which was discussed in 2.3.5.

5.2.8.1 Constraints

1. The constraint below is specified within the context of `MessageContainer`. the constraint is checked only if the presence of an `InvalidMessage` is detected. Then this constraint checks whether the `InvalidMessageChannel` is connected to the `MessageContainer` using `LinkChannel`.

```
context MessageContainer
   inv UseInvalidMessageChannelWithLinkChannel:
      not self.targetLinkChannel->isEmpty() and
     (self.headerMC.targetMessageLinker.targetInsideShapes
      ->exists(oclIsTypeOf(InvalidMessage)
      implies
           self.targetLinkChannel.targetShapes.
           oclIsTypeOf(InvalidMessageChannel)
```

2. Constraint on link mapping is enforced to check that no outgoing connections are created from the `InvalidMessageChannel` with `LinkChannel`.

5.2.9 Event Message Pattern

Use an Event Message for reliable, asynchronous event notification between applications. [2]

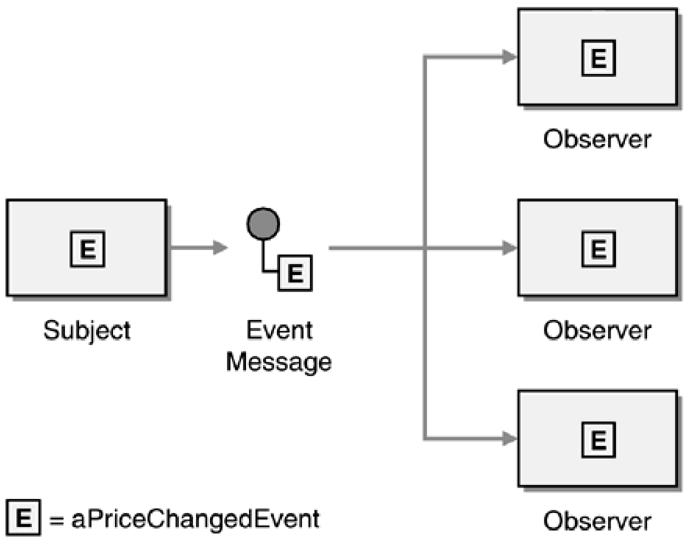


Figure 45: Event Message Pattern

The subset of domain model representing Figure 45 is shown with UML in Figure 46. The event message is represented by `Class EventMessage` which is a derived class of `InnerShapes`.

The SubjectContainer and ObserverContainer have containment association with EventMessage so that the former classes can be created as compartments for holding EventMessage during the graphical definition of Graphical modeling.

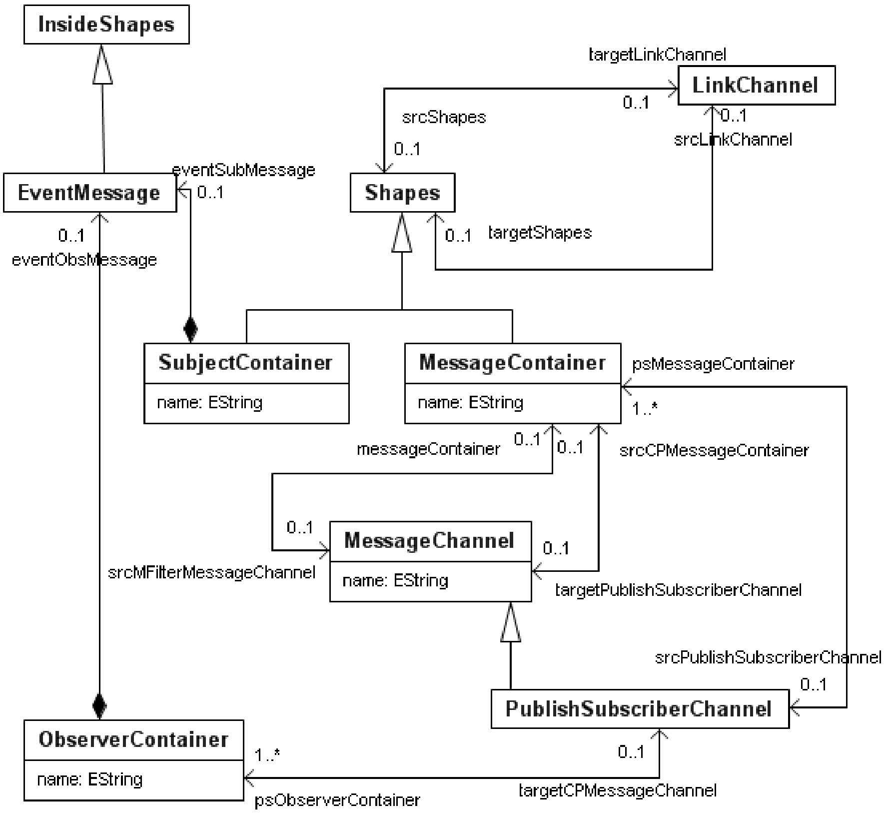


Figure 46: Event Message Pattern - Class Diagram

`MessageContainer` has aggregate relationship with `EventMessage` so that it can be represented as a container to hold event messages. The PublishSubscriberChannel is used to create connection between the MessageContainer containing `EventMessage` and `ObserverContainer`. We create association between `MessageContainer` and `ObserverContainer` that is navigable in both directions. With the association, we see the `targetCPMessageChannel` `EReference` in `ObserverContainer` and `psObserverContainer` `EReference` in `PublishSubscribeChannel`.

5.2.9.1 Constraint

To confirm the well-formedness of this pattern, constraint at the context of `MessageContainer` containing `EventMessage` should check that each `ObserverContainer` connected to it with the `PublisherSubscriberChannel` has got one `EventMessage`. The constraint below fulfils the purpose,

```
context MessageContainer
   inv EventMessageNotConfiguredProperly:
      (self.eventMessageMC->size()>0) and
       not(srcLinkChannel->isEmpty()) and
       not(self.srcPublishSubscriberChannel.PSObserverContainer
           ->isEmpty()) implies
       self.srcPublishSubscriberChannel.PSObserverContainer.
           eventObsMessage->forAll(p:EventMessage|p->size()=1) and
       self.srcLinkChannel.srcShapes.
           oclAsType(SubjectContainer).eventSubMessage->size()>0
```

5.2.10 DeadLetter Channel Pattern

> *When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a Dead Letter Channel. [2]*

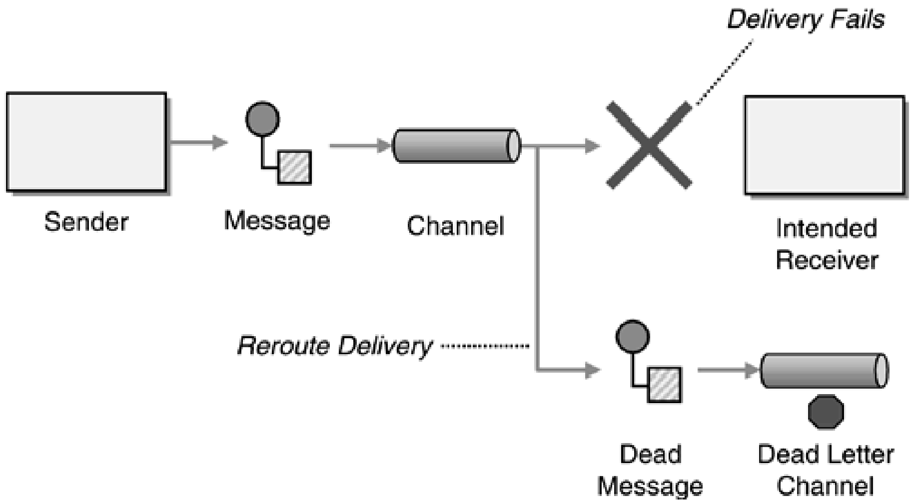


Figure 47: Dead Letter Channel Pattern

The subset of domain model representing Figure 47 is shown with UML in Figure 48. The Dead Letter Channel is represented by `Class DeadLetterChannel`. The DeadMessage is specified as a sub class of InnerShapes. MessageContainer has a containment association with `DeadMessage` together with `Header`.

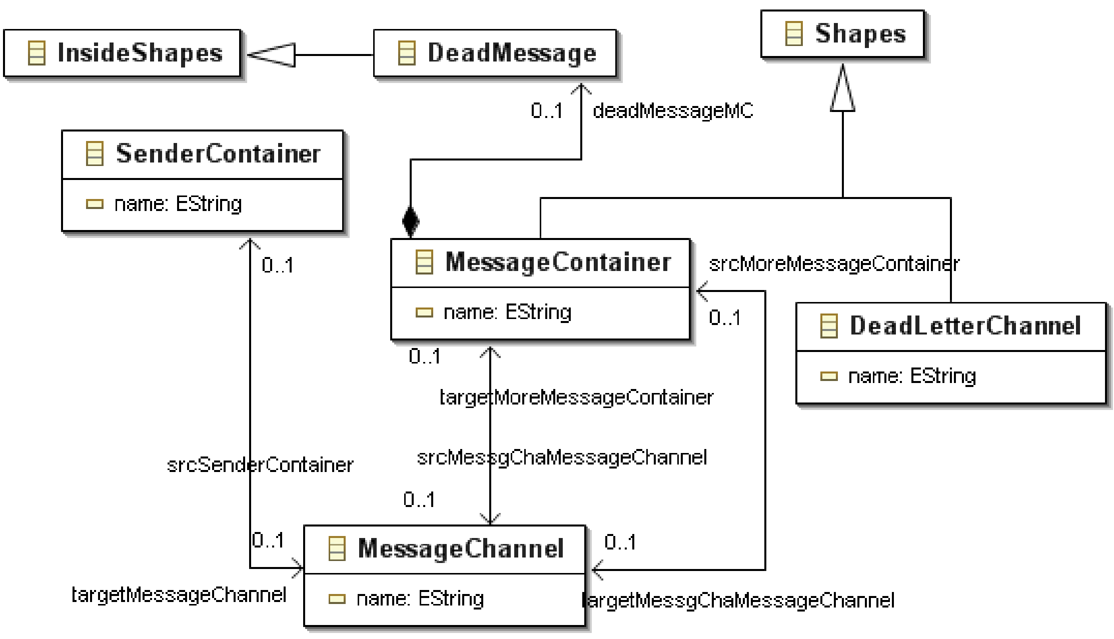


Figure 48: Dead Letter Channel Pattern - Class Diagram

We use `MessageChannel` to create connection between `MessageContainers` and `LinkChannel` to create connection between the `SenderContainer` and `Messagecontainer` and from the `MessageContainer` containing the Dead message to the DeadLetterChannel. To create link connection using `MessageChannel` among MessageContainers we create two associations that denote the link connection from them. The associations are navigable in both directions. With this association, we see `targetMessgChaMessageChannel EReference` in `MessageContainer` and `srcMoreMessageContainer EReference` in `MessageChannel`. With the other association, we see `srcMessgChaMessageChannel EReference` in `MessageContainer` and `targetMoreMessageContainer EReference` in `MessageChannel`.

5.2.10.1 Constraint

The following constraints are considered to enforce the well-formedness of Message Dispatcher Pattern.

1. The Constraint below checks the presence of `DeadLetterChannel` instance to be connected to the `MessageContainer`, when the latter element contains the `EventMessage` instance.

```
context MessageContainer
   inv UseDeadLetterChannelWithLinkChannel:
      not self.targetLinkChannel->isEmpty() and
     (self.headerMC.targetMessageLinker.targetInsideShapes
      ->exists(DeadMessage))
      implies self.targetLinkChannel.targetShapes.
           oclIsTypeOf(DeadLetterChannel)
```

2. Constraint on link mapping will be enforced to check that no outgoing connections are created from the `DeadLetterChannel` with LinkChannel.

5.2.11 Channel Purger

Use ChannelPurger to remove unwanted message from a channel. [2]

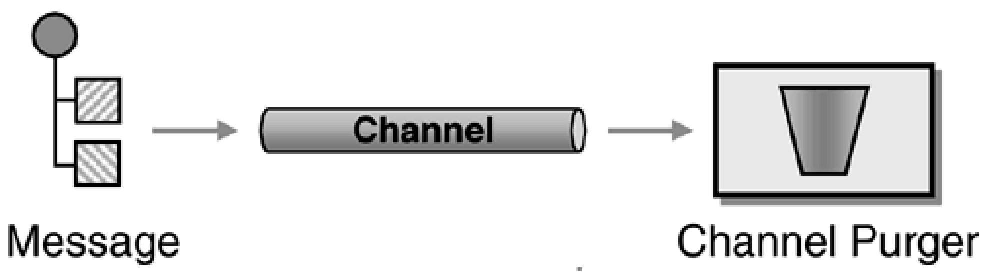


Figure 49: Channel Purger Pattern

The subset of domain model representing Figure 49 is shown with UML in Figure 50. The Channel Purger is represented by `Class ChannelPurger`. MessageContainer has containment association with `Header` and `DataMessage` to function as a compartment for messages.

`MessageChannel` establishes connection link between `MessageContainer` and `ChannelPurger`. This is done by creating association between `MessageContainer` to `MessageChannel` and `MessageChannel` to

ChannelPurger. The association is navigable in both directions. With the association between MessageContainer and MessageChannel, we see srcCPMessageContainer EReference at MessageChannel and targetCPMessageChannel EReference at MessageContainer. With the association between MessageChannel and ChannelPurger, we see targetCPChannelPurger EReference at MessageChannel and srcCPMessageContainer EReference at ChannelPurger.

To check the well-formedness for this pattern, link constraints can be written for link mapping to check that no outgoing connection can be created with having the ChannelPurger as its source, since ChannelPurger would be the last stage a message can reach.

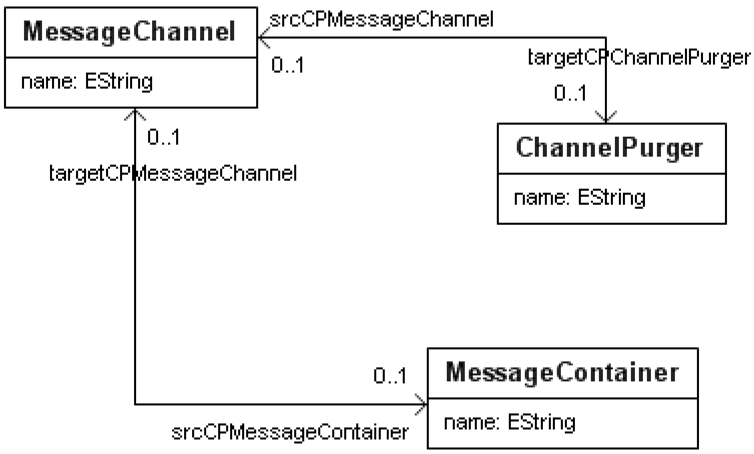


Figure 50: Channel Purger Pattern - Class Diagram

5.2.12 Message Expiration Pattern:

Set the MessageExpiration to specify a time limit for how long the message is viable. [2]

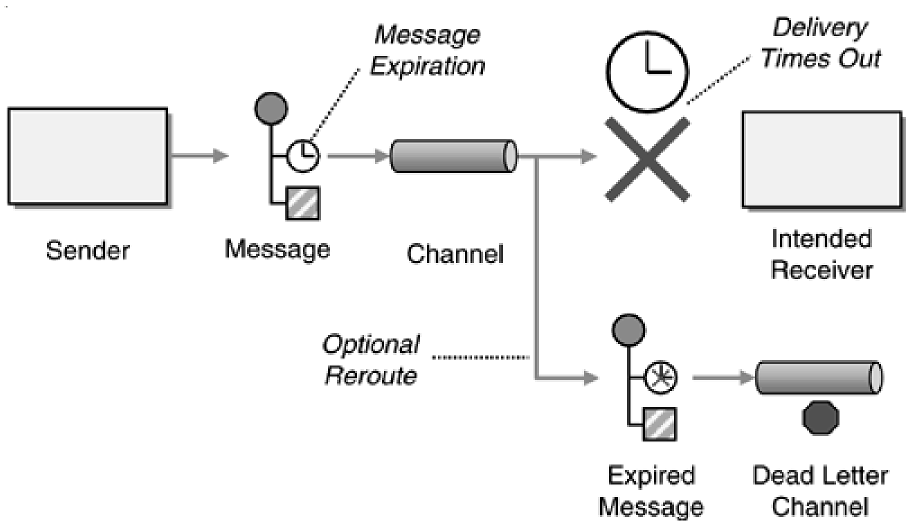


Figure 51: Message Expiration Pattern

The subset of domain model representing Figure 51 is shown with UML in Figure 52. The message expiration and expired message are represented by `Class ExpiringMessage` and `Class ExpiredMessage`. `MessageContainer` have composite association relationship with `ExpiringMessage` and `ExpiredMessage`. This aggregation helps to place the instances of `ExpiredMessage` and `ExpiringMessage` in MessageContainer compartment.

We use `MessageChannel` to create connection between `MessageContainers` and `LinkChannel` to create connection between the `SenderContainer` and `Messagecontainer` and from the `MessageContainer` containing the `Dead message` to the `DeadLetterChannel`. To create link connection using `MessageChannel` among MessageContainers we create two associations that denote the link connection from them. The association are navigable in both directions. With this association, we see `targetMessgChaMessageChannel EReference` in `MessageContainer` and `srcMoreMessageContainer EReference` in `MessageChannel`. With the other association, we see `srcMessgChaMessageChannel EReference` in `MessageContainer` and `targetMoreMessageContainer EReference` in `MessageChannel`.

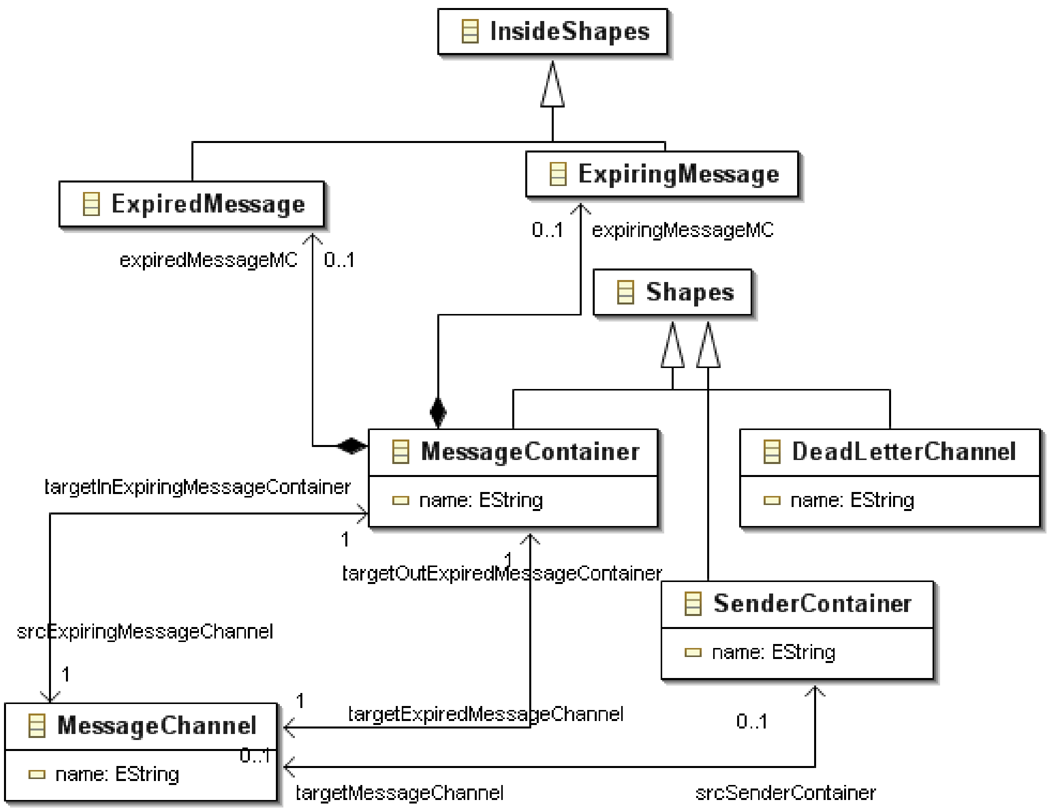


Figure 52: Message Expiration Pattern - Class Diagram

We use `MessageChannel` is used to create connection link between the `MessageContainer` containing `ExpiringMessage` and the `MessageContainer` containing `ExpiredMessage`. To create the connection between them we create two associations that are navigable in both directions. With the first association, we see `targetExpiredMessageChannel EReference` in `MessageContainer` and `targetOutExpiredMessageContainer EReference` in `MessageChannel`. With the other association, we see `srcExpiringMessageChannel EReference` in

MessageContainer and targetInExpiringMessageContainer EReference in MessageChannel.

LinkChannel is used to create connection links from SenderContainer to MessageContainer; and from MessageContainer to DeadLetterChannel.

5.2.12.1 Constraints

The following constraints are considered to enforce the well-formedness of MessageExpiration Pattern.

1. The constraint below checks if an ExpiringMessage is present in the MessageContainer then the MessageContainer with ExpiredMessage should be connected using MessageChannel.

```
context MessageContainer
   inv ExpiringMessageNotConfiguredProperly:
       not (self.srcExpiringMessageChannel.
           TargetOutExpiredMessageContainer->isEmpty()) and
      (self.headerMC.targetMessageLinker.
           targetInsideShapes->
           exists(oclAsType(ExpiringMessage))) implies
       self.srcExpiringMessageChannel.
           targetOutExpiredMessageContainer.headerMC.
           targetMessageLinker.targetInsideShapes
       ->exists(oclIsTypeOf(ExpiredMessage))
```

2. The Constraint below checks the presence of one DeadLetterChannel instance to be connected to the MessageContainer, when the latter element contains the ExpiredMessage instance.

```
context MessageContainer
   inv ExpiredMessageRequiresDeadLetterChannel:
       self.headerMC.targetMessageLinker.
       TargetInsideShapes
       ->exists(oclIsTypeOf(ExpiredMessage))
       and not
       self.targetExpiredMessageChannel.
           targetInExpiringMessageContainer->isEmpty()
       and not self.targetLinkChannel->isEmpty()
     implies
       self.targetExpiredMessageChannel.
           targetInExpiringMessageContainer.headerMC.
           targetMessageLinker.targetInsideShapes->
     exists(oclIsTypeOf(ExpiringMessage)->isEmpty() and
       self.targetLinkChannel.targetShapes.
           oclIsTypeOf(DeadLetterChannel)
```

3. Constraint on link mapping is enforced to check that no outgoing connections are created from the DeadLetterChannel.

5.2.13 DataTypeChannel Pattern

Use a seperate DataType Channel for each datatype so that all data on a particular channel is of the sametype. [2]

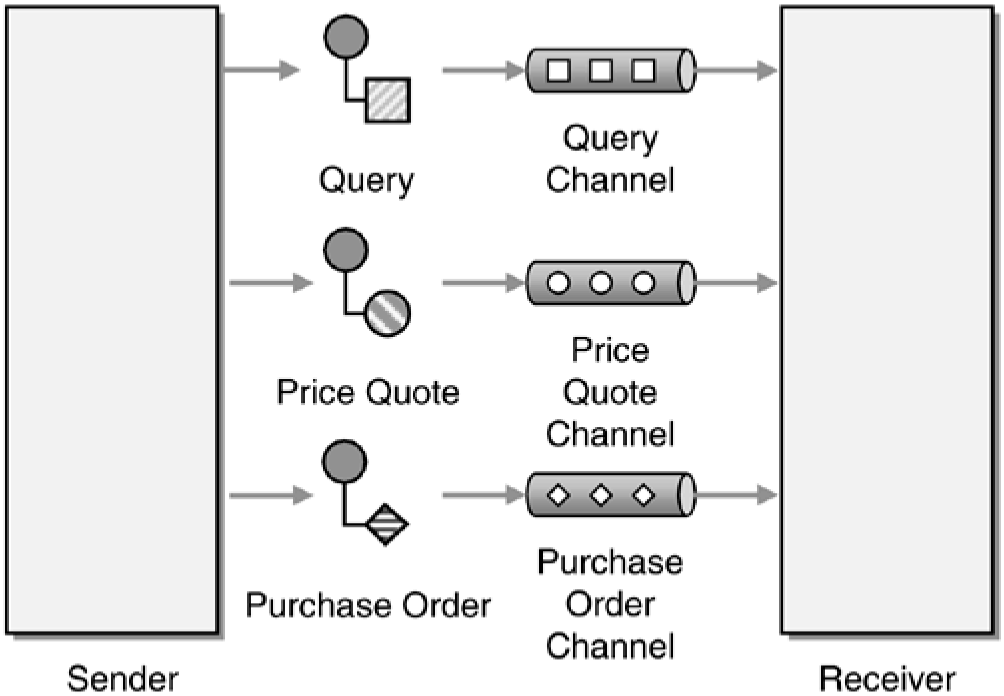


Figure 53: DataTypeChannel Pattern

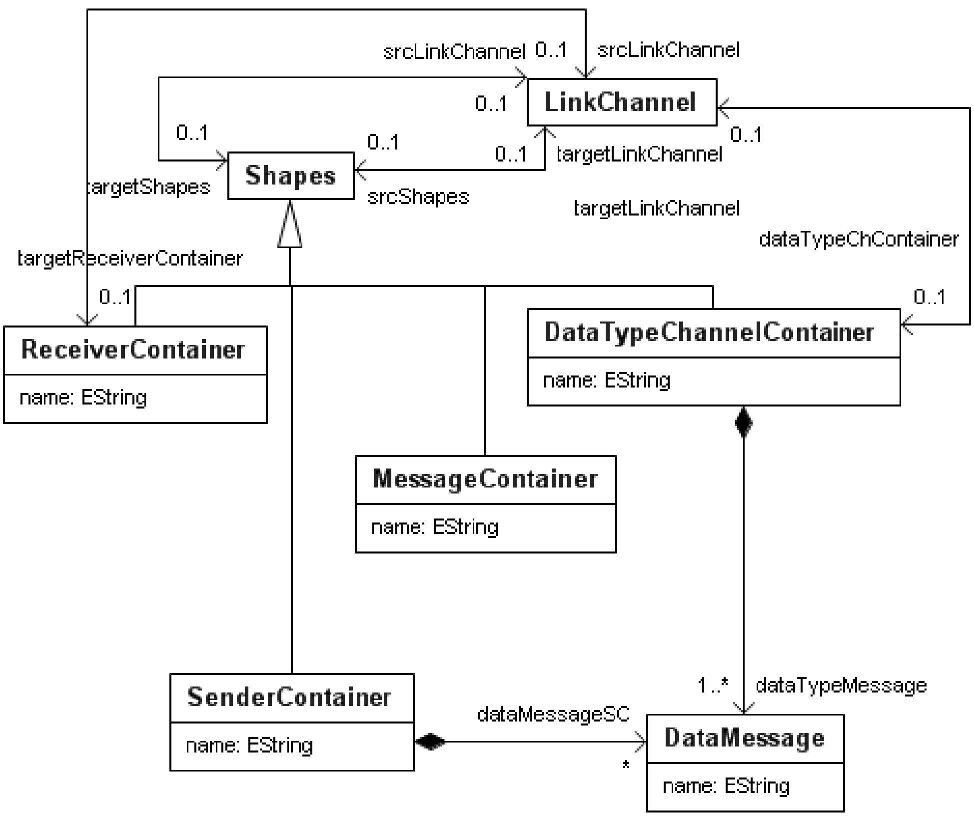


Figure 54: Data Type Channel Pattern - Class Diagram

The subset of domain model representing Figure 53 is shown with UML in Figure 54. The data type Channel pattern is represented by `Class DataTypeChannelContainer`. `MessageContainer` is used to represent the compartment for placing messages that contain a `Header` with a `DataMessage`.

DataTypeChannelContainer is a container that should hold same kind of datamessages that are being provided as input to them. `Class DataTypeChannelContainer` has an aggregation relationship with `dataMessage` thereby making itself a container for placing datamessages. In our implementation we give uniqueness to the datamessage with their name. The datamessages of a datatype should have identical name. Link Channel represented by `Class LinkChannel` creates connection links between the `SenderContainer` to `MessageContainer`, from MessageContainer to `DataTypeChannelContainer` and from the latter to the `ReceiverContainer`.

5.2.13.1 Constraint

To check the wellformedness of `DataTypeChannelContainer`, constraint in the context of the same can check if the datatypes are of same kind, i.e. by checking if the name of all the datamessages are same with that of datamessage of `MessageContainer` that creates a outgoing connection to the `DataTypeChannelContainer`.

```
context DataTypeChannelContainer
   inv DataTypeChannelNotUniqueInDataType:
      not (srcLinkChannel->isEmpty()) implies
      self.srcLinkChannel.srcShapes.
         oclAsType(MessageContainer).dataMessageMC.name->
         includesAll(self.dataTypeMessage.name)
      and
         self.dataTypeMessage->forAll(p1,p2:DataMessage|p1.name=
         p2.name)
```

5.2.14 WireTap Pattern

> *Insert a Wire Tap into the channel, a simple Recipient List that publishes each incoming message to the main channel as well as the secondary channel. [2]*

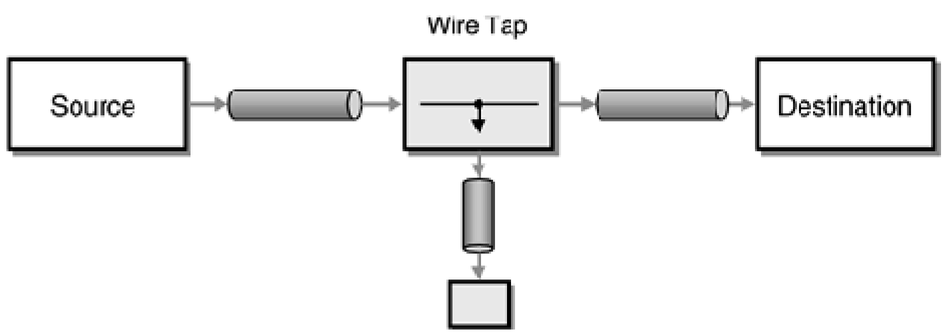


Figure 55: WireTap Pattern

The subset of domain model representing Figure 55 is shown with UML in Figure 56. Wire tap pattern is represented by `Class WireTap`. The Source and Destination are represented by `Class Source` and `Class Destination` respectively. Both of the classes has containment relationship with `Header` and `DataMessage` that makes it a compartment to where the tool smith can create messages.

MessageChannel is used to create connection link between,

1. **Source → WireTap**
   - To represent this connection we create a bi-directional association between `Source` with `MessageChannel` and `MessageChannel` with `WireTap`. With the first association we see `inputSource EReference` in `MessageChannel` and `targetInputWTMessageChannel EReference` in `Source`. In the second association we see `inputWireTap EReference` in `MessageChannel` and `srcInputWTMessageChannel EReference` in `WireTap`.
2. **WireTap → Destination**
   - To represent this connection we create a bi-directional association between `WireTap` with `MessageChannel` and `MessageChannel` with `Destination`. With the first association we see `targetOutputWTMessageChannel EReference` in `WireTap` and `outputWireTap EReference` in `MessageChannel`. In the second association we see `outputDestination EReference` in `MessageChannel` and `srcOutputWTMessageChannel EReference` in `Destination`.

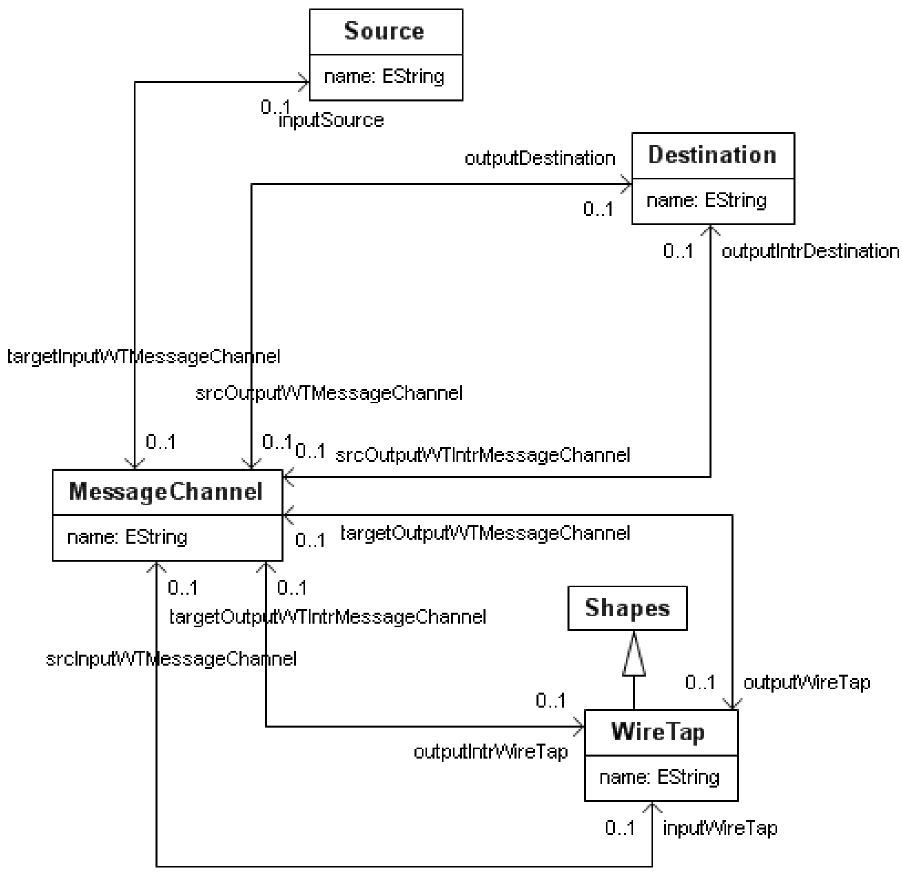


Figure 56: WireTap Pattern - Class Diagram

3. **WireTap → IntermediateDestination**
    - To represent this connection we create a bi-directional association between `WireTap` with `MessageChannel` and `MessageChannel` with `Destination`. With the first association we see `targetOutputWTIntrMessageChannel` `EReference` in `WireTap` and `outputIntrWireTap` `EReference` in `MessageChannel`. In the second association we see `outputIntrDestination` `EReference` in `MessageChannel` and `srcOutputWTIntrMessageChannel` `EReference` in `Destination`.

5.2.14.1 Constraints

The following constraints are considered to enforce the well-formedness of WireTap Pattern.

1. The Constraint below checks that all messages created in `Source`, `Destination` and Intermediate `Destination` Container should have only one `DataMessage` instance connected to each `Header` instance using `MessageLinker`.

```
context WireTap
   inv WireTapNotconfiguredProperly:
      not (srcInputWTMessageChannel->isEmpty()) and
      not (targetOutputWTMessageChannel->isEmpty()) and
      not (targetOutputWTIntrMessageChannel->isEmpty())
      implies
      self.srcInputWTMessageChannel.inputSource.
         headerSource->
         forAll(p:Header | p.targetMessageLinker->size()=1)
         and
      self.targetOutputWTMessageChannel.outputDestination.
         headerDestination->forAll
              (p:Header | p.targetMessageLinker
         ->size()=1) and
      self.targetOutputWTIntrMessageChannel.
         outputIntrDestination.headerDestination->forAll
              (p:Header | p.targetMessageLinker
         ->size()=1)
```

2. The constraint below checks that the name of the datamessage collected as collection in Source, Destination and intermediate Destination should be equal.

```
context WireTap
   inv DataMessageTypesEqualInWireTap:
      not (srcInputWTMessageChannel->isEmpty()) and
      not (targetOutputWTMessageChannel->isEmpty()) and
      not (targetOutputWTIntrMessageChannel->isEmpty())
      implies
      self.srcInputWTMessageChannel.inputSource.
         dataMessageSource.name->asBag()->
      intersection(self.targetOutputWTMessageChannel.
         outputDestination.dataMessageDestination.name
         ->asBag()) =
      self.targetOutputWTIntrMessageChannel.
         outputIntrDestination.dataMessageDestination.name
         ->asBag()
```

5.3 Summary

We have discussed in this chapter about the constraints that are required for the validation of our domain model instances. The constraints for each pattern are mentioned along with description about how the constraint would enforce its functionality. Specifying such constraints at the domain model level will help to enforce domain model integrity, which prevents from creating invalid and wrong domain model instances.

We use the discussed constraints in successfully creating the diagram editor for Enteprise Integration Patterns. The constraints specified here are generally associated between one or more classes. Specifying constraints among a group of classes tend to get more complexier. Alternate way to come up with professional range Graphical Editors would be to make changes in the domain model language so as get additional support. The domain model must evolve further to support the generation of visual editors.

6 Conclusion

6.1 Conclusion

The vision of this master thesis was to study and develop the GMF framework in providing support for domain model integrity (at the domain model level) in the input used to generate diagram editors. As a case study, the patterns specified in Enterprise Integration Patterns [2] were expressed in a domain model and a diagram editor was generated for it.

This project started when GMF was still evolving and effort was required to understand internals of the framework. Before the implementation of our prototype, the available examples in the framework provided limited help in describing a large scale domain model compared to our implementation. One of the advantages of GMF, no requirement to know details of the GEF API, was confirmed. This helped us to generate basic models with which we could scale our implementation to our prototype.

The aim of providing validation for the domain model instances lead to the examination of EMFT Validation Technology in the form of Audit Rules. Since specifying the constraints as audits amounts to specifying procedural details, moreover involving manual coding, this available approach was considered not be appropriate since the constraints considered for well-formedness rules (WFRs) are large in number and declarative in nature. An alternate approach was specified in Article [1] which refers to the domain model integrity of EMF with EMFT OCL Technology, but devoid of any GMF concerns. We were able to implement this approach in GMF framework to enforce domain model integrity in a declarative way. One of the advantages of this approach is the resulting encapsulation of validation code, which does not obscure the graphical definition files dedicated to generating diagram editors. In particular, updating the WFRs and regenerating model code does not require regeneration of the graphical definitions, not even the mapping definition.

Constraints in OCL for checking the domain model integrity for our prototype were developed which prevent the creation of invalid diagram model instances. Due to time limitations the total number of patterns that can be implemented with the generated diagram editor amounts to fourteen, which anyway results in a useful software engineering tool. Further, an OCL interpreter was integrated into GMF. This interpreter can be used to test arbitrary, run-time provided OCL queries (so called "ad-hoc" queries) against the diagram instances being edited.

6.2 Outlook

GMF is still evolving. Future implementations and developments in this framework will ease the generation of rich diagram editors. Knowledge of the framework (as well as know-how around other EMF technologies) is still needed to fine tune the involved software components.

The implementation of all patterns specified in [2], can result in a full fledged Enteprise Integration diagram editor. The domain model creation for enterprise patterns offers a golden opportunity to force modeling concepts to be scaled to the maximum. Modeling the remaining patterns is a direct extension once the feasibility of the software architecture has been demonstrated with the current implementation.

The inclusion of OCL Interpreter for GMF helped us in writing efficient constraints that enforce domain model integrity. The contributions made in this thesis have been well received by the GMF community, and thus have made their way into the best practices around generating diagram editors for Eclipse.

Bibliography

[1] Christian W. Damus, Implementing Model Integrity in EMF with EMFT OCL
URL: [http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html]

[2] Gregor Hohpe, Bobby Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, 2003

[3] Octopus: OCL Tool for Precise Uml Specifications
URL: [http://www.klasse.nl/octopus/index.html]

[4]: Emfatic Language for EMF Development
URL: [http://www.alphaworks.ibm.com/tech/emfatic]

[5]: From Front End To Code - MDSD in Practice,
URL: [http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html]

[6] GMF Tutorial - Part 1
URL: [http://wiki.eclipse.org/index.php/GMF_Tutorial]

[7]: A. Jibran Shidqie, Conversion of Octopus UML Models into Eclipse UML2 Models, 2006

[8]: The EMF Validation Framework Overview
URL: [http://www.eclipse.org/emf/docs/]

[9]: JET Tutorial Part 1 (Introduction to JET)
URL: [http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html]

[10] GMF Tutorial - Part 2
URL: [http://wiki.eclipse.org/index.php/GMF_Tutorial_Part_2]

[11] GMF Tutorial - Part 3
URL: [http://wiki.eclipse.org/index.php/GMF_Tutorial_Part_3]

[12] Eclipse Consortium, Eclipse Graphical Modeling Framework
URL: [http://www.eclipse.org/gmf/]

[13] Eclipse Consortium, Eclipse Modeling Framework
URL: [http://www.eclipse.org/emf/]

[14] Eclipse Consortium, Eclipse Graphical Editing Framework
URL: [http://www.eclipse.org/gef/]

[15] Gabriele Taentzer, Towards Generating Domain-Specific Model Editors with Complex Editing Commands
URL: [http://www.sciences.univ-nantes.fr/lina/atl/www/papers/eTX2006/18-GabrieleTaentzer.pdf]

Appendix A

Domain Model Instances of EAI Patterns

1. Aggregator Pattern

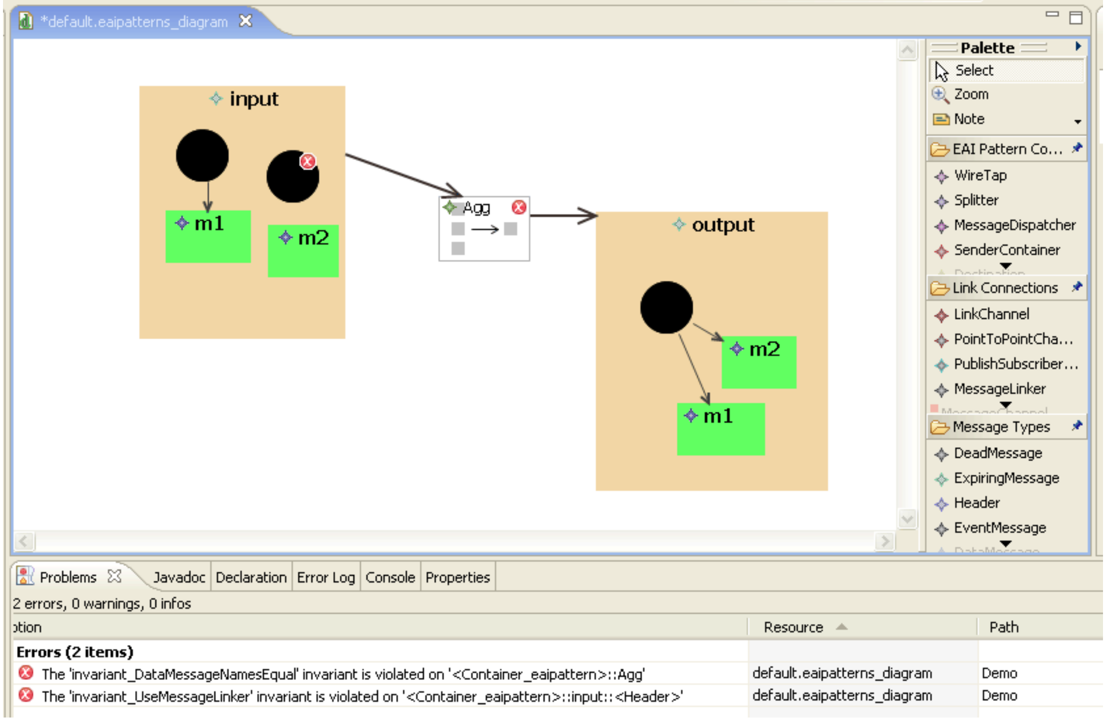


Figure 57: Domain Model Instance - Aggregator Pattern

2. ContentFilter Pattern

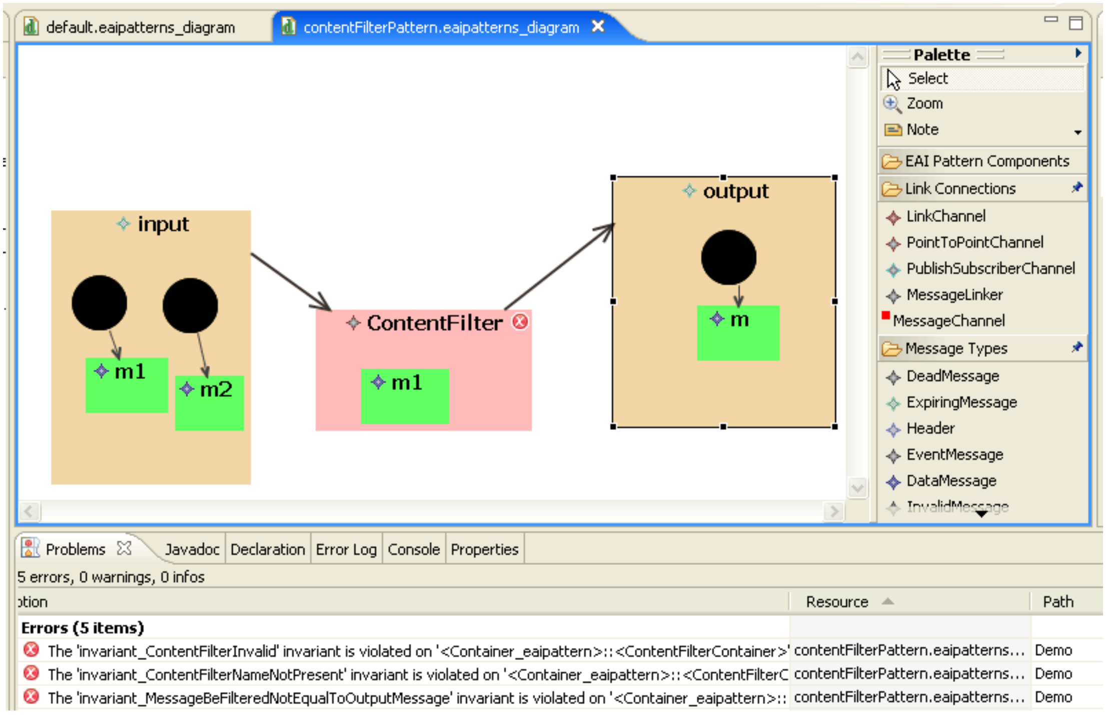


Figure 58: Domain Model Instance - Content Filter Pattern

3. Invalid MessageChannel

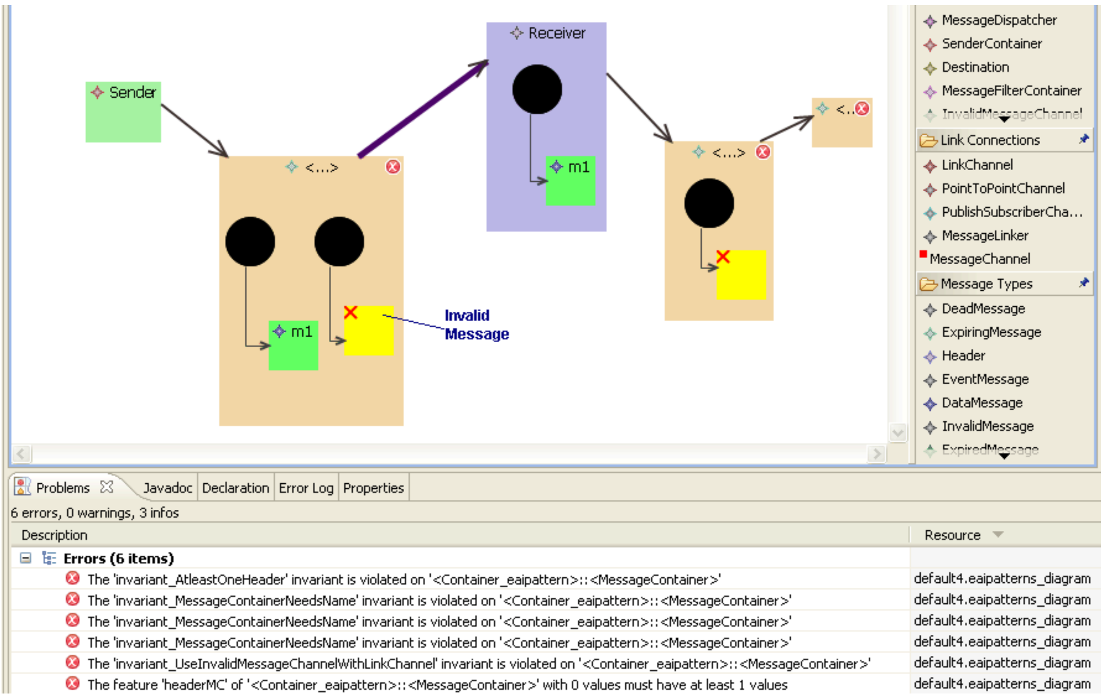


Figure 59: Domain Model Instance - Invalid Message Channel

4. Splitter Pattern

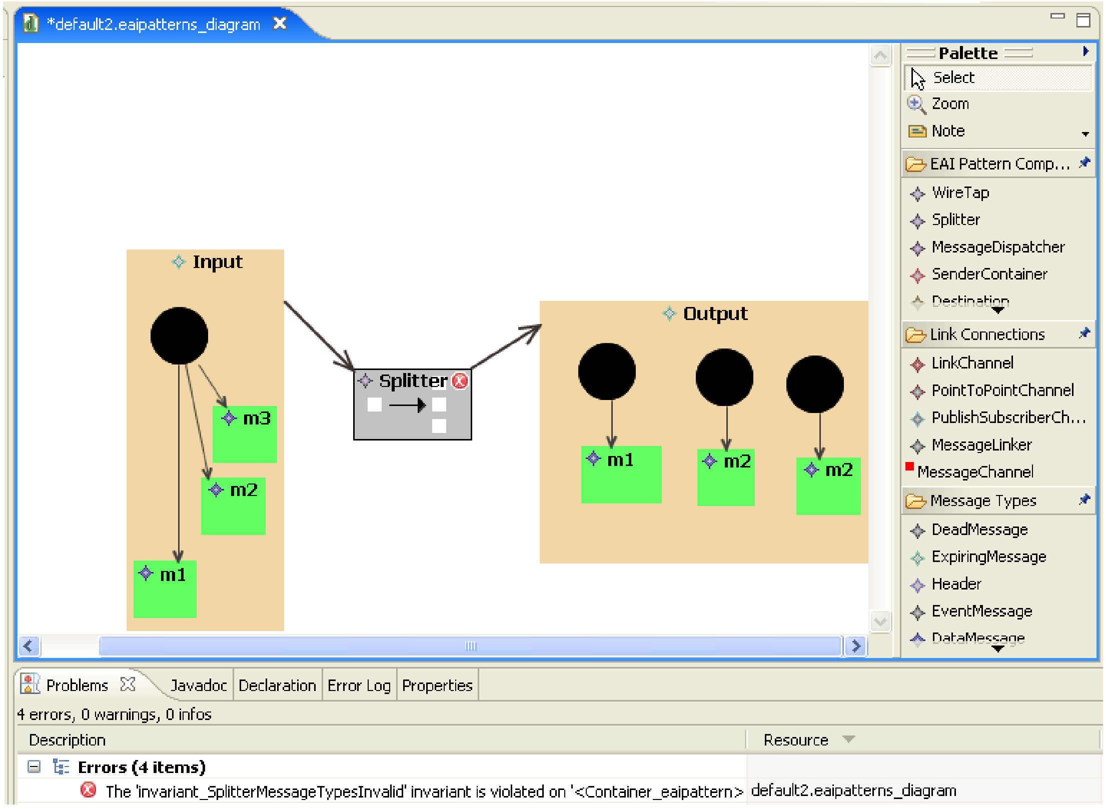


Figure 60: Domain Model Instance - Splitter Pattern

5. Message Expiration Pattern

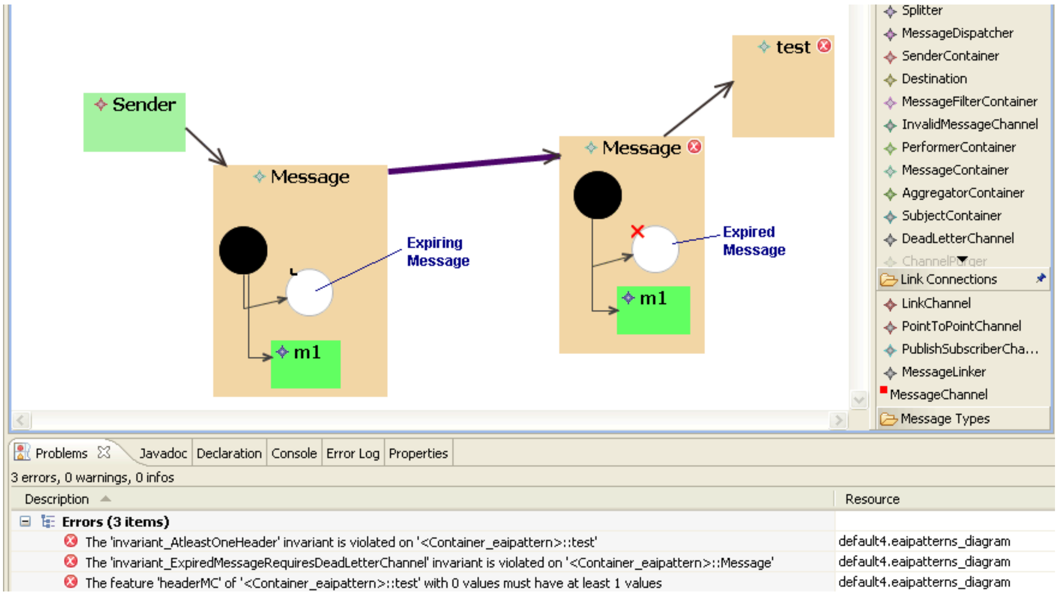


Figure 61: Domain Model Instance - Message Expiration Pattern

Appendix B

1. Emfatic Basic Type Names

Emfatic Keyword	Ecore EClassifier name	Java type name
`boolean`	EBoolean	boolean
`Boolean`	EBooleanObject	java.lang.Boolean
`byte`	EByte	byte
`Byte`	EByteObject	java.lang.Byte
`char`	EChar	char
`Character`	ECharacterObject	java.lang.Character
`double`	EDouble	double
`Double`	EDoubleObject	java.lang.Double
`float`	EFloat	float
`Float`	EFloatObject	java.lang.Float
`int`	EInt	int
`Integer`	EIntegerObject	java.lang.Integer
`long`	ELong	long
`Long`	ELongObject	java.lang.Long
`short`	EShort	short
`Short`	EShortObject	java.lang.Short
`Date`	EDate	java.util.Date
`String`	EString	java.lang.String
`Object`	EJavaObject	java.lang.Object
`Class`	EJavaClass	java.lang.Class
`EObject`	EObject	org.eclipse.emf.ecore.EObject
`EClass`	EClass	org.eclipse.emf.ecore.EClass

2. Class Feature Modifiers

modifier	means	applies to
`readonly`	EStructuralFeature.changeable = false	attribute, reference
`volatile`	EStructuralFeature.volatile = true	attribute, reference
`transient`	EStructuralFeature.transient = true	attribute, reference
`unsettable`	EStructuralFeature.unsettable = true	attribute, reference
`derived`	EStructuralFeature.derived = true	attribute, reference
`unique`	ETypedElement.unique = true	attribute, reference, operation, parameter
`ordered`	ETypedElement.ordered = true	attribute, reference, operation, parameter
`resolve`	EReference.resolveProxies = true	reference
`id`	EAttribute.iD = true	attribute

3. Multiplicities

Emfatic multiplicity expression	ETypedElement lowerBound	ETypedElement upperBound
none	0	1
`[?]`	0	1
`[]`	0	unbounded (-1)
`[*]`	0	unbounded (-1)
`[+]`	1	unbounded (-1)
`[1]`	1	1
[*n*]	*n*	*n*
`[0..4]`	0	4
[*m*..*n*]	*m*	*n*
`[5..*]`	5	unbounded (-1)
`[1..?]`	1	unspecified (-2)

4. Class Feature Kind Keywords

Emfatic keyword	introduces
`attr`	EAttribute
`op`	EOperation
`ref`	normal EReference (EReference.containment = false)
`val`	"by value" EReference (EReference.containment = true)

Appendix D

Dual Link Connections – Round Link Mapping

Consider a scenario where a link connection is to be made between two instances of a same class but in different directions. The direction of the link created is specified in the graphical level and has no direction information from the domain model.

So when an instance of link class exists before creating an another instance of link between the same class but in the opposite direction, this action results in the formation of duplicate links.

Below is the code snippet that handles the creation of link instances.

```
private void refreshConnections() {
try {
       collectAllLinks(getDiagram());
       Collection existingLinks = new LinkedList(getDiagram().getEdges());
       for (Iterator diagramLinks = existingLinks.iterator(); diagramLinks
                     .hasNext();) {
              Edge nextDiagramLink = (Edge) diagramLinks.next();
              EObject diagramLinkObject = nextDiagramLink.getElement();
              EObject diagramLinkSrc = nextDiagramLink.getSource()
                            .getElement();
              EObject diagramLinkDst = nextDiagramLink.getTarget()
                            .getElement();
              int diagramLinkVisualID = EnterVisualIDRegistry
                            .getVisualID(nextDiagramLink);
              for (Iterator modelLinkDescriptors = myLinkDescriptors
                            .iterator(); modelLinkDescriptors.hasNext();) {
                     LinkDescriptor nextLinkDescriptor = (LinkDescriptor)
                            modelLinkDescriptors.next();
                     if (diagramLinkObject == nextLinkDescriptor
                                   .getLinkElement()
                                   && diagramLinkSrc ==nextLinkDescriptor
                                          .getSource()
                                   && diagramLinkDst == nextLinkDescriptor
                                          .getDestination()
                                   && diagramLinkVisualID ==
nextLinkDescriptor
                                          .getVisualID()) {
                            diagramLinks.remove();
                            modelLinkDescriptors.remove();
                     }
              }
       }
       deleteViews(existingLinks.iterator());
       createConnections(myLinkDescriptors);
} finally {
       myLinkDescriptors.clear();
       myEObject2ViewMap.clear();
}
}
```

RefreshConnections() in Container_eaipatternCanonicalEditPolicy.java

The round about solution is to prevent the method from creating the dual link if the presence of a previous instance in the same direction is detected. The code snippet below helps to provide the effect.

```
private void refreshConnections() {
try {
	collectAllLinks(getDiagram());
	Collection existingLinks = new LinkedList(getDiagram()
			.getEdges());
	for (Iterator diagramLinks = existingLinks.iterator();
		diagramLinks.hasNext();) {
		Edge nextDiagramLink = (Edge) diagramLinks.next();
		EObject diagramLinkObject = nextDiagramLink.getElement();
		EObject diagramLinkSrc = nextDiagramLink.getSource()
			.getElement();
		EObject diagramLinkDst = nextDiagramLink.getTarget()
			.getElement();
		int diagramLinkVisualID = EnterVisualIDRegistry
			.getVisualID(nextDiagramLink);
		for (Iterator modelLinkDescriptors = myLinkDescriptors
			.iterator(); modelLinkDescriptors.hasNext();) {
			LinkDescriptor nextLinkDescriptor = (LinkDescriptor)
			modelLinkDescriptors.next();
			if (isSameLink(diagramLinkObject, diagramLinkSrc,
				diagramLinkDst, nextLinkDescriptor)) {
				diagramLinks.remove();
				modelLinkDescriptors.remove();
			}
		}
	}
	deleteViews(existingLinks.iterator());
} finally {
	myLinkDescriptors.clear();
	myEObject2ViewMap.clear();
}
}

private boolean isSameLink(EObject diagramLinkObject,
		EObject diagramLinkSrc, EObject diagramLinkDst,
					LinkDescriptor nextLinkDescriptor)
{
	boolean directLink = diagramLinkSrc ==
nextLinkDescriptor.getSource()
		&& diagramLinkDst == nextLinkDescriptor.getDestination();
	boolean reversedLink = diagramLinkDst ==
nextLinkDescriptor.getSource()
		&& diagramLinkSrc == nextLinkDescriptor.getDestination();
	return diagramLinkObject == nextLinkDescriptor.getLinkElement()
		&& (directLink || reversedLink);
}
```

RefreshConnections() in Contianer_eaipatternCanonicalEditPolicy.java