Metamodel based Code Generation in DPF Editor

Anders Sandven

Master's Thesis in Informatics – Program Development

Department of Informatics
University of Bergen

HØGSKOLEN I BERGEN
Department of Computer Engineering
Bergen University College

March 2012

Contents

List of Figures

Preface

Foreword

This is submitted as my master's thesis in the Master's Degree programme in Informatics – Program Development, at the University of Bergen and Bergen University College.

Through the work on this thesis I have had the opportunity to learn something completely new. The first years as an IT student introduced me to software modelling with UML; class, activity, sequence, and use case diagrams presented as a single language. It was confusing and hard to see the real use for these diagrams beyond serving as documentation.

The introduction of model-driven engineering introduced me to software modelling in a way that one easily can understand the practical applications. Creating languages fitted for specific domains was an idea that really sparked an interest in me. To me, it is a whole new way of thinking programming.

The DPF project has introduced me to a lot of skilled and interesting people who really cares about what they do. Working with people like this makes it a lot more fun and interesting for myself, as one always have someone to discuss a problem with.

Acknowledgements

This thesis has been conducted by myself, but it would not be possible without the help of several people. The feedback and support from my supervisor Yngve Lamo has been invaluable. A special thanks goes to Florian Mantz for providing feedback and help on the technical side of things. I would also like to thank Adrian Rutle who (together with Florian) helped me figuring out the direction of my project. Øyvind Bech has also deserved a big thank you for always being helpful and providing me with LaTeX sources for this thesis. I would like to thank the rest of the DPF project: Suneetha Sekhar, Xiaoliang Wang, and Alessandro Rossini.

Lastly, I want to thank my parents and my good friend and cousin, Håkon Botnen, who has helped me proofreading this thesis.

Bergen, 23 March 2012

Chapter 1

Introduction

1.1 Motivation

Modelling languages has been used since the 1960's when the Entity-Relationship (ER) model was conceived [7]. Around the same time, the programming language Simula [10] was launched as the first object-oriented language which triggered the interest in object-oriented design and analysis. In the 1980's CASE tools were the next big thing, and was by some predicted to completely replace textual programming. The use of these tools never really took off, but we see the legacy of these tools today through software such as intergrated development environments (IDE). With the adoption of UML [40] 1.1 in 1997 as a standard by the Object Management Group (OMG) [37], there was an increased interest in modelling.

Model-driven engineering (MDE) is influenced by lessons learned from previous efforts. What modelling languages and CASE tools tried to achieve in the 80-90's was systems based entirely on models, which would generate runnable code. This is a difficult task, and is probably even harder today due to the increased complexity in software. MDE addresses this complexity with raising the level of abstraction.

The primary artefact in MDE is models. These models can be graphical and textual, but they serve the same purpose; creating an abstraction of the system which can be standardized and easily communicated with non-programmers. The MDE field has in the recent years gained new popularity, with ideas like *domain-specific languages* (DSL), *domain-specific modelling languages* (DSML) and meta-modelling as important concepts. MDE encourages a narrow and clearly defined problem domain which can be expressed with concise and expressive languages based on models.

When using a diagrammatic approach to MDE, one create models which usually has no meaning to a computer (semantics). To address this issue, we need a transformation engine or generator that can help us transform our model into something usable that the computer understands. A common so-

lution to this problem is model-to-text transformation; generating code to a target language, which in its turn is interpreted, or compiled and executed.

This thesis will focus on facilitating model-to-text transformations in DPF Editor, the reference implementation of *Diagram Predicate Framework (DPF)*. DPF is a formal diagrammatic approach to MDE. DPF Editor has at this point roughly implemented the metamodelling aspects of DPF, but lacks any form of model transformation support. To tackle this problem, one needs a general approach which can be used to facilitate code generation for all the languages created, not a specific solution targeting one DSML. This thesis will try to answer the research question: *is it possible to create a general code generation facility for modelling languages specified in DPF?* Through the thesis there will be introduced a general solution for creating code generators in the DPF Editor based on the Xpand framework [54]. In the end the tool will be demonstrated with the building of a code generator for web applications, based on the Play Framework [45].

1.2 Structure of Thesis

The thesis is structured in the following way:

Chapter 2 – Model-based Development
This chapter gives an introduction to Model-driven Engineering, Domain-specific languages, Diagram Predicate Framework and more. We will investigate the concept of *language workbenches* and take a look at some of the existing model-based solutions which we find on the market today. We will finish up with a presentation of the DPF Editor, and compare it with the reviewed tools.

Chapter 3 – Code Generation
We will introduce code generation as a general activity, and look at why it is interesting. The problem which this thesis tries to solve will be presented. At the end we will compare different code generation solutions and introduce the chosen framework.

Chapter 4 – Design and Development
As the chapter title suggests, this chapter will describe the development of the code generation tool. We will take a closer look at how the chosen code generation framework works on the inside, and how this applies to our solution. In the end of the chapter there will be an overview covering what has been achieved, as well as the shortcomings of the tool.

Chapter 5 – Demonstrating the Tool
This chapter will demonstrate the tool with the creation of Java code from a simple model. We will investigate the developed tool's features as well as the framework it is based on.

Chapter 6 – Conclusion

We will conclude the thesis with a summary of what the developed solution is capable of. At the very end there is suggestions for further work, both in the direction of code generation and as a *language workbench*.

Chapter 2

Model-based Development

This chapter will give an introduction to some of the terms and ideas surrounding MDE. We will give an introduction to the concept of *language workbenches* and demonstrate a few of the leading products on the market today. In the end we show what features the DPF Editor has to offer, and perform a comparison between all the products.

2.1 Model-driven Engineering

In the last decades we have seen a dramatic increase in software complexity. This complexity has been handled by more expressive and improved general programming languages, but has failed to keep up with the vast increase in functionality. The consequence of this is that developers are struggling to learn to new platforms, and often only learning a subset of what it has to offer. Another problem is that the developer misses the big picture, and forgets system-wide issues such as performance. MDE can help fight these issues with a higher level of abstraction. This abstraction yields advantages such as increased productivity, code quality/consistency and improved communication with domain experts, as well as programmers [47][21].

Model-driven engineering is a model-centric approach to software development, using models as first-class entities. This is in contrast to the classic code-centric approach, where models sometimes are used for describing design and is implemented by code, or the code *is* the design and the implementation (agile principle). The MDE process defines models which capture the concepts of certain parts in a computer system and some kind of transformation which makes it understandable by a computer. The models need to have a narrow focus on the problem domain to reach its full potential. If the models are too general, it will become very hard to create an expressive model transformation.

Models can be categorized as *prescriptive* or *descriptive*. A prescriptive model gives a description of the system before it is produced and works

as a blueprint, while a descriptive model works as documentation for the system that is created [22]. In MDE, models act as both descriptive and prescriptive in the sense that the model is a replacement for the traditional code, and it also acts as a guide for understanding the system at a higher level.

2.1.1 Domain-Specific (Modelling) Languages

Domain-Specific Languages

There are different ways to create a model in MDE, but in common is the need for a language in which you can specify models. A popular approach, which is often not regarded as MDE, is the creation of textual *domain-specific languages* (DSL). These languages have the trait of focusing on a particular domain, where their source files acts as the model. DSLs has been used for a number of years, and are used in everything from software build tools (e.g. Ant [2]) to HTML and SQL. Fowler [21] defines a few key elements for a DSL:

Programming language: A DSL is not a *general programming language*, but it shares a lot of properties; it should be formal to be understood by a computer, but also be intuitive for a human being.

Language nature: A DSL should not consist of separate expressions, but have a syntax that can be composed by several expressions put together.

Language expressiveness/focus: The language should only support the absolute minium of features needed for a limited part of a system. The expressiveness of a good DSL comes from a clear focus on a limited domain.

A DSL usually consists of a grammar (that defines the language), a parser (which conforms to the grammar) and a *semantic model*. Fowler defines the semantic model as "the model that is populated by a DSL", i.e. one directly populate a framework or API.

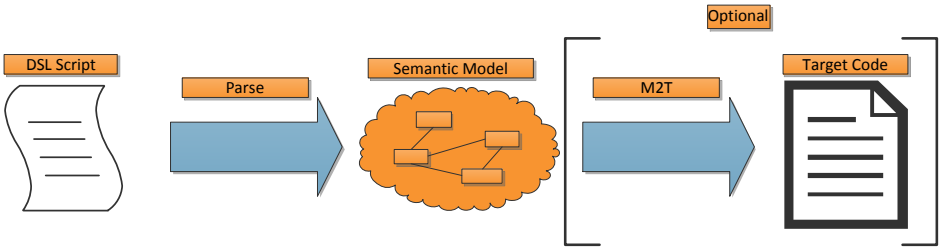

Figure 2.1: Figure shows how a DSL works.

Figure 2.1 shows how a typical DSL works. Usually a DSL script is parsed and used to populate a semantic model which defines the semantics behind the script. As shown, model-to-text transformations are an optional part of DSLs. This is because DSLs are often used to allievate the configuration of a piece of software, and can be regarded as a thin wrapper around an API. Such DSLs are not necessarily focused on raising the level of abstraction, but rather to create a more convenient syntax for an API. In a MDE context, we want to raise the level of abstraction and provide the concrete semantics through model transformations.

Fowler [21] lists four main types of domain-specific languages:

Internal DSLs Internal DSLs are languages which are created within the host programming language. The host language is formed in a way that create a fluent syntax.

```
builder
  .specification()
    .graph("m1")
      .node("DomainClass")
      .node("Type")
      .arrow("Attribute", "DomainClass", "Type")
      .arrow("Reference", "DomainClass", "DomainClass")
    .endgraph()

  .specification()
    .typeGraph("m1")
      .graph("instance")
        .node("Post", "DomainClass")
        .node("User", "DomainClass")
      .endgraph()
```

Listing 2.1: Simple internal DSL for creating specifications in DPF

Listing 2.1 shows a simple internal DSL created in Java for populating a DPF model. This particular example uses *method chaining*, a technique where a method (e.g. node()) returns its class instance when called.

External DSLs External DSLs are the most common DSLs. These languages use external DSL scripts which are not bound be the host language's syntax and thus have a lot more syntactic freedom. Text files are parsed and used to either populate a semantic model, or do a model transformation (see figure 2.1). Examples of such languages are SQL, HTML, XAML, Make etc.

```
mm:=TGraph<DPF>{
  z:Node-e0:Arrow->b:Node,
  a:Node-e1:Arrow->b:Node-e2:Arrow->c:Node-t:*->String,
}

m:=TGraph<mm>{
  b:c-y@1:t->["Hallo"],
  b:c-y@2:t->["Test"],
  f:a-l:e1->o:b,
}

ecore(mm)
```

Listing 2.2: External DSL for creating specifications in DPF

Listing 2.2 shows an external DSL for defining *DPF specifications*[1]. This example shows model `m` being typed by a metamodel `mm`, and then serializing the model. DPF is explained in section 2.6.

Fragmentary DSLs These are DSLs which are embedded in the host programming language, such as *regular expressions*.

Language workbenches Discussed in section 2.4.

Domain-Specific Modelling Languages

DSMLs serve the same purpose as a DSL, but is usually presented in a visual manner. These languages are specified according to a *metamodel*, sometimes defined as a "model of models" [47]. The DSMLs are usually defined by domain experts and experienced developers to ensure that the domain concepts are properly described. The languages are then used by other developers as a programming tool. These languages are dependent on surrounding functionality and tooling, e.g. graphical editors, code generators and model validation. Workflow and tooling are further discussed in section 2.4, Language Workbenches.

The main difference between DSLs and DSMLs are the inner workings of the modelling environment. Both DSLs and DSMLs needs a parser which can translate the source file to an internal representation. The DSL's parser usually produce an abstract syntax tree or it populates a semantic model.

Figure 2.2 shows how the equivalent of the DSL script is the internal representation in a DSML-tool. The internal representation of the model is

[1] The textual DPF tool is part of Florian Mantz' Ph.D. work.

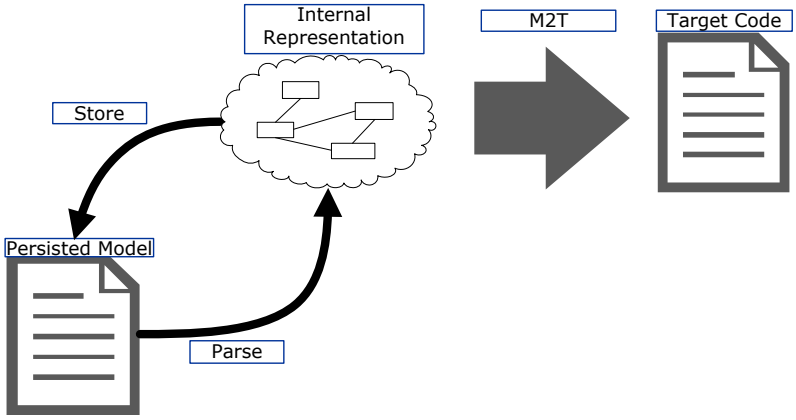

Figure 2.2: An abstract view of a Domain Specific Modelling Language tool.

the model; the persisted model are only a means for the computer to understand the model. The model-to-text transformation (M2T) is not marked as optional as in figure 2.1, because DSMLs require some kind of model transformation to have any meaning to a computer.

The DSML is dependent on tooling, e.g. a visual editor which has an internal representation of the model. Usually this model also has an external representation in the form of a file. A popular format for this is the XML Metadata Interchange (XMI), a XML-based format that is tailored for representing models. Furthermore DSMLs are purely a MDE concept, and are rarely used to populate semantic models. Model transformations with DSLs and DSMLs are discussed further in chapter 3.

2.1.2 Model-driven Architecture

Model-driven Architechture is the Object Management Group's (OMG) [42] model-driven development initiative and can be regarded as a predecessor to MDE. It was launched as a standard in 2001. Even though the concepts of domain-specific modelling was known, OMGs MDA created new interest in the subject. Its core principle is to abstract away technology details of a specified system with the use of platform-independent models (PIM). PIMs contain business functionality and behaviour, and are reusable across any operating system/architecture. Before generating any code from a MDA project to a target environment, a platform-specific model (PSM) is created through *model-to-model transformations*. A PSM is generated for each target environment.

> "The primary goals of MDA is portability, interopability and reusability through architectural separation of concern. [42]"

MDA meets a lot of critique for different reasons. MDA uses model-to-model transformations to generate more detailed models as you go, i.e. from a PIM to a PSM resulting in the need to maintain more models than necessary. Another problem arise when you edit the PSM; how is this reflected in the PIM?

The MDA spec is centered around OMG's own standards. The modelling language used is very often UML, but it can be any *Meta-Object Facility (MOF)* compliant modelling language. MOF is a closed metamodel hierarchy with four levels (see more in next section).

2.2 Metamodelling

Metamodelling is an important concept in MDE, yet its definition is highly debated [47] [46]. A metamodel is sometimes referred to as a "model of models"; Cook and Kent [8] claims this is a bad definition and should rather be "a model of the concepts expressed by a modelling language". A suitable definition could also be "a model of a modelling language".

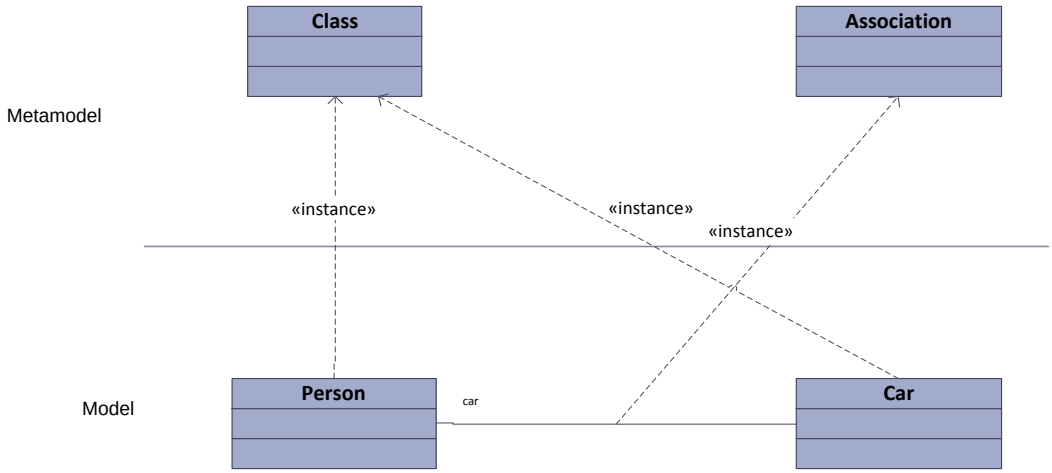

Figure 2.3: A simple metamodel/model example.

Figure 2.3 shows a simple metamodelling example where each of the modelling constructs is an instance of a concept from the metalevel. A metamodel specifies the abstract syntax of a modelling language. This syntax defines the concepts, attributes, their relationships and how they fit together to create a valid model [47]. A model's available features is defined by what features its metamodel defines. Metamodels has their own *meta-metamodel* which define the syntax they must adhere to as well. The metalayer on top of the hierarchy is usually defined by a *reflexive* modelling language, a language which is defined in itself.

In 1997 OMG adopted MOF 1.1 as a standard, around the same time as UML 1.1 was adopted. The goal of MOF is to create a common way of describing model metadata (data about models). This is done through a number of specifications such as MOF Core, MOF IDL Mapping, MOF XMI Mapping, MOF Model to text (MOFM2T) among others.

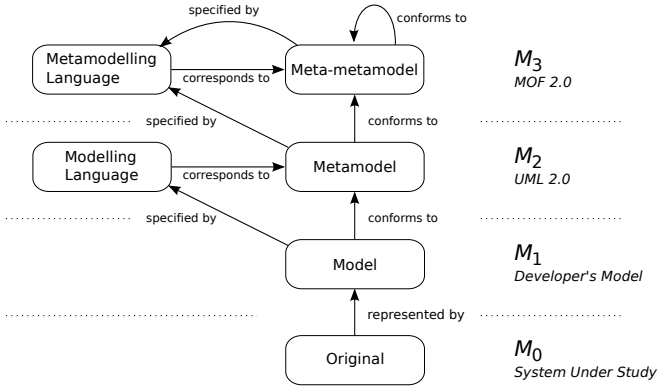

Figure 2.4: Figure shows the four layer hierarchy defined in MOF.

MOF defines a metamodel hierarchy with four layers (M_3 to M_0) where each level conforms to the level above it. MOF's goal is to create a "grammar" for how the models should be formed. MOF has two variants; Complete MOF (CMOF) and Essential MOF (EMOF). EMOF contains only the most basic constructs, which in broad strokes is classes, relations, types, packages and operations. Even though an EMOF compliant modelling language is enough for a lot of use cases, the language is fixed and thus falls short compared to the expressiveness of a DSML created without an EMOF compliant modelling language.

Figure 2.4 shows that the M_3 level, the meta-metamodel, is defined in itself (*reflexive layer*). This is done through constructs which the MOF specification provides. This layer is the metamodel for the UML language. The M_2 level is the instance of M_3, and is the model which implements UML, adhering to the abstract syntax of the layer above. The M_1 layer is where UML concepts such as Class, Atttribute and Operations are used to create a comprehensible model of the system which is then instantiated by level M_0, the *system under study (SUS)*. M_0 contains the original, real world object, e.g. "Ford" would be instance of a M_1 class Car.

MOF is often under criticism in the literature for beeing, rigid because of a fixed number of metalevels (MOF is sometimes referred to as the four-layered hierarchy). This criticism is somewhat incorrect as the MOF specification does not restrict the number of layers [40].

2.3 Constraints

A constraint helps us to specify complex requirements beyond what basic modelling constructs can offer. In UML we have the option to apply constraints to our models. Unfortunately only simple constraints like multiplicity and uniqueness can be applied directly in the metamodel. These constraints are called *structural constraints*. If we want to create a constraint which cannot be declared in the model, we need to create these externally.

The "external" constraints are called *attached constraints*, and are created using a declarative language like *Object Constraint Language (OCL)* [39]. A solution like this is problematic; a UML model is a graph based model and OCL is textual language, making it harder to reason about the models. Not only for the developer, but also for the domain expert(s). There are also issues with reflecting changes in the model against the OCL constraints, creating synchronization issues [47]. OCL is an OMG standard and is the recommended attached constraints facility used in combination with UML and MDA.

2.4 Language Workbenches

Language workbench is a term coined by Martin Fowler [21] that describes an environment for creating DSML/DSLs and corresponding tools. The workbenches should provide an IDE-like environment for creating DSML/DSLs, and in addition to generate code it should generate tooling for the specified language. Language workbenches are a relatively new concept which has been increasingly popular the last few years, and is under heavy research and changes. When working with language workbenches, the development process are divided into two phases. Figure 2.5 shows how the DSML is created along with relevant tooling, such as editors and code generators. This activity should be performed by domain experts, as well as developers with experience. Using experienced developers will ensure that the tooling for the language is tailored to the users (which are developers). The figure also illustrates that developers utilize the domain-specific environment created from the DSML.

Martin Fowler describes a few common elements in a language workbench [21]:

***Semantic Model* schema** defines the structure of the *Semantic Model*, based on a metamodel. In language workbenches the semantic model is the metamodel used for the model. Fowler uses the term *schema* for the metamodel, or DSML if you like. The base model of the tool is called a *schema definition language*.

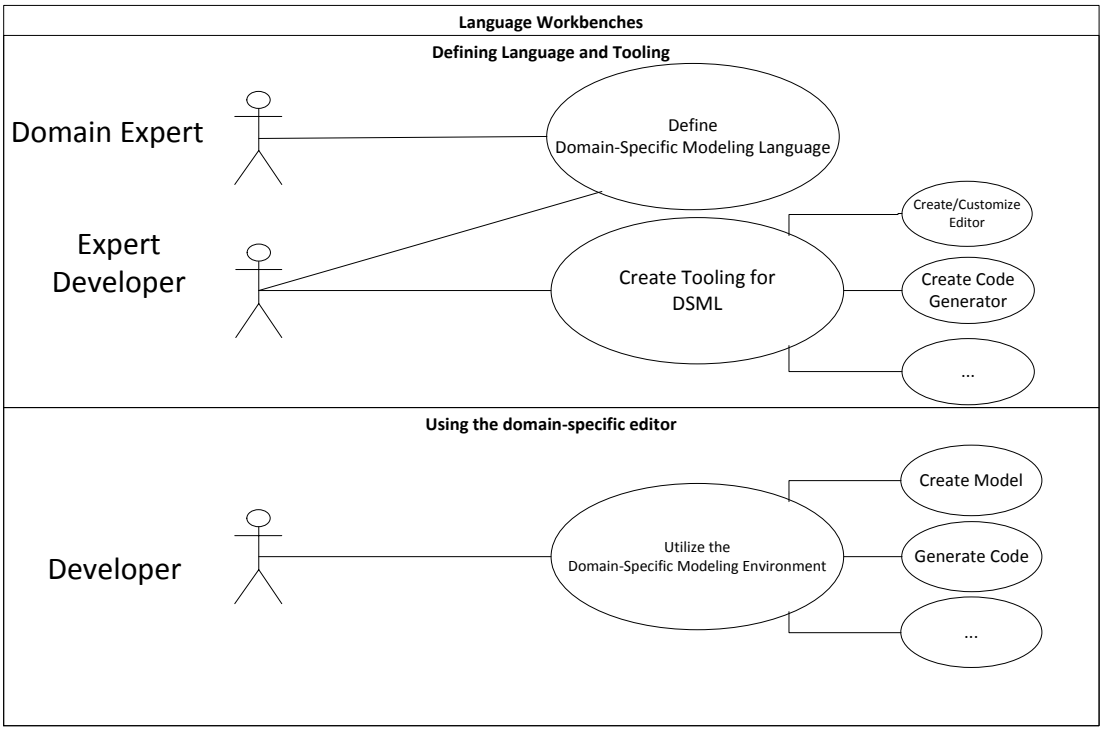

Figure 2.5: Figure shows the intended workflow for a language workbench.

DSML editing environment defines an IDE-like experience when editing a DSML/DSL, often through *projectional editing*. Projectional editing provides some kind of interface where we edit our models. Besides the diagrammatic approach, there are also schemas, forms, hierarchies etc. The counterpart of projectional editing is source editing, which is traditional programming using text files. Some language workbench solutions gives the opportunity of using multiple projections for your model, i.e. you can get a schema representation of a diagram based model, or a text representation.

Semantic Model behaviour defines the semantics of the language specified. Usually this is achieved through code generation. Although the model is stored in the language workbench, the computer does not understand the intent of the model initially. The semantics of a DSML is defined through model transformations. The model can configure an API, create an interpreter, do a model-to-model transformation, but most often generating executable code (model-to-text transformation) is the chosen solution.

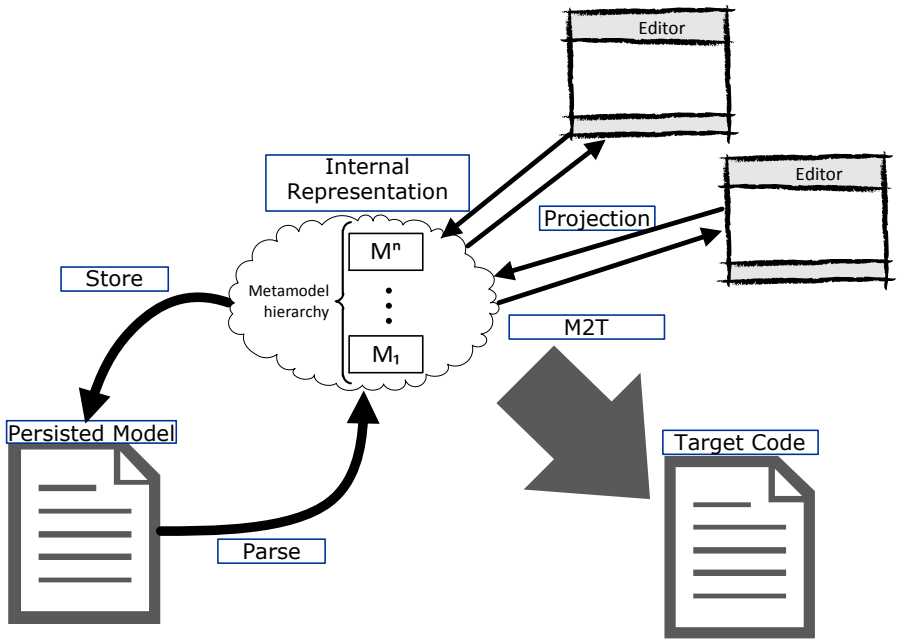

Figure 2.6: Figure shows a conceptual view of a language workbench.

Figure 2.6 shows how a language workbench work conceptually. At the heart of the language workbench, we find an internal representation of any models we operate on. This might be a single model, or a complete metamodel hierarchy. It forms the basis for the tools built around it. The tool-

ing in a language workbench is defined as anything from the concept of a schema-definition language to code generators and constraint validation. The figure shows the most basic parts needed for a functional language workbench. As discussed in section 2.1.1 the internal representation of the model is the model. A persistence facility is needed for reuse and storing of the models. The persisted version of the models are not meant for hand editing, this should be done through the editing environment.

2.5 Existing Language Workbench Solutions

This section gives a brief overview of some of the more popular projection based language workbench solutions on the market today.

2.5.1 Eclipse and Eclipse Modeling Framework

The Eclipse Foundation [16] is a non-profit organization which was established in 2004 with the goal of creating a vendor-neutral, transparent and open community around the Eclipse projects. The Eclipse eco-system consists of a vast amount of projects, everything from development tooling to SOA and web related projects. A common misconception is that when people think of Eclipse, they have Eclipse Java Development Tools (JDT) in mind. Eclipse JDT is a combination of different Eclipse projects, and provides a feature rich Integrated Development Environment for the Java programming language. This means Eclipse JDT is a collection of (in)dependent plug-ins which ultimately results in the IDE. The philosophy behind Eclipse is modularity; at the heart of Eclipse lies Equinox, an implementation of the OSGi core framework specification, which facilitates the modular nature of the *Eclipse Platform* [9]. All functionality in Eclipse is provided through plug-ins.

EMF

EMF (or rather EMF Core) is the project at the center of Eclipse's modelling technologies. The framework aims to unite Java, XML and UML; models in EMF can be expressed either way, where one can generate one representation from the other [6]. EMF is a modelling tool which can be found in the middle of a pure modelling tool and Java code. Instead of following the "raising abstraction" mantra of pure modelling tools, EMF recognizes the need for a tool where the programmer can easily understand the mapping between model and code. EMF provides a modelling environment based on its own metamodel Ecore (which is a reflexive model). Ecore contains a basic set of modelling constructs that corresponds to EMOF. EMF also provide persistence and a code generation facility that can generate Java code from your metamodel, unit-tests for said code, and the possiblity to generate a

simple hierarchy based editor. EMF does not facilitate code generation from the model instance level. This is handled by other modelling components, such as tools from the M2T (model-to-text) project.

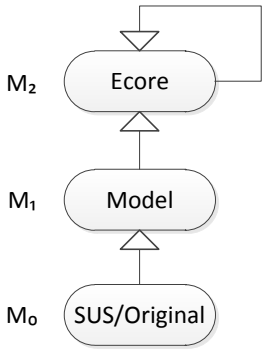

Figure 2.7: Figure shows a how Ecore fit into the MOF hierarchy.

Figure 2.7 shows how Ecore fit into the MOF hierarchy. As the figure shows, EMF does not give you the possiblity of multiple metalevels. Although this restriction strengthens the intent of a comprehensible mapping between model and code, it makes it hard to create a DSML that can be substituted for code. An important note about Ecore's MOF hierarchy is that the levels vary based on what you try to achieve. With the use of the UML2 support, one have four levels as usual, but if you use Ecore to model a system directly, you only have three levels.

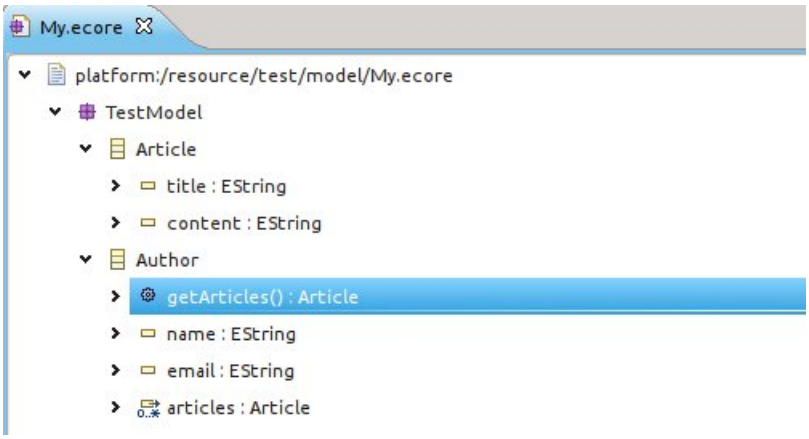

Figure 2.8: Figure shows editing an Ecore file using the hierarchy view.

Figure 2.8 and 2.9 shows how the creation of a metamodel in Ecore. You

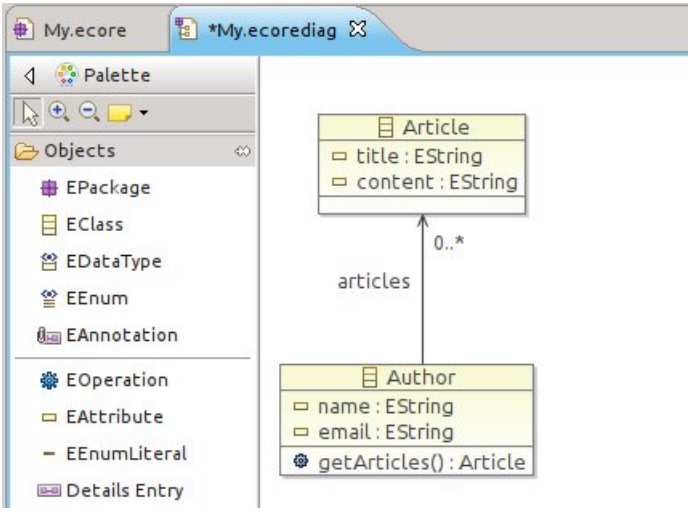

Figure 2.9: Figure shows editing an Ecore file using the graph based view.

can choose between the hierarchy solution, or a projectional view which requires the generation of an Ecore Diagram file.

EMOF defines a limited amount of modelling constructs, among them classes, attributes, references, packages and operations. These concepts are easy to understand for any programmer with a background in object oriented languages, and thus creating a clear understanding of the mapping between model and code. As the MOF standard [38] defines, constraints beyond cardinality and uniqueness are defined externally. In EMF one can define *invariants* which are defined as a boolean method in the model, or a constraint which is defined as a method on a validator class, not in the model itself. Either way, hand editing is required to define what the constraint/invariant actually constrains. The implementations can use OCL if desired.

2.5.2 Obeo Designer

Obeo Designer is the product of a french company named Obeo. Obeo is a consulting company which has a strong focus on MDE and MDA. The company is an Eclipse Foundation Strategic Member and a contributor on many Eclipse Modelling projects. Obeo Designer is based on Eclipse EMF, which means it uses *Ecore* as its modelling language. The tool's main focus is on creating visualizations for your model, as well as generators.

Figure 2.10 shows the editor for creating custom visualizations for your Ecore model. There is no graphical editor for creating graphics, one have to choose from a set of pre-defined ones or use an external image.

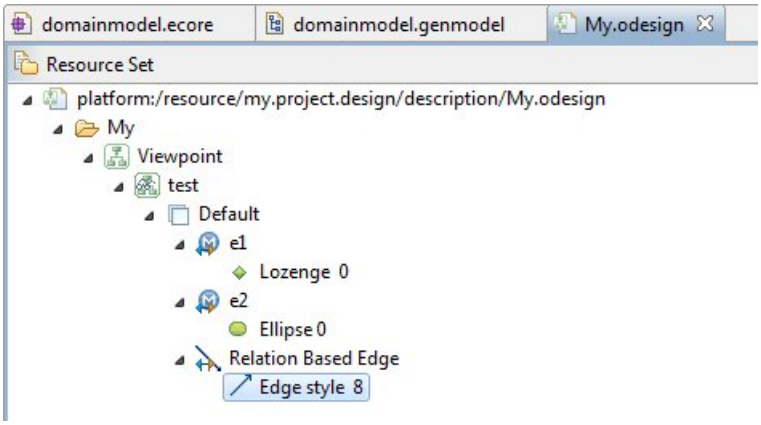

Figure 2.10: Example of creating a custom visualization for an Ecore model in Obeo Designer.

generate.mtl generateJava.mtl

```
[comment encoding = UTF-8 /]
[module generateJava('http://www.obeodesigner.com/lwc2011')/]

[template public generateJava(domainModel : DomainModel)]
    [for (entity : Entity | domainModel.eAllContents()->filter(Entity))]

    [file (entity.name.concat('.java'), false, 'UTF-8')]
public class [entity.name/] {

            [for (property : Property | entity.properties)]
                [property.declaration()/]

            [/for]
            [for (property : Property | entity.properties)]
                [property.getter()/]

            [/for]
            [for (property : Property | entity.properties)]
                [property.setter()/]

            [/for]
}
    [/file]

    [/for]
[/template]
```

Figure 2.11: Creating Acceleo templates for code generation.

One of the promoted features of Obeo Designer is the option of creating layers in your graph visualizations. This means you can create visualizations which fulfill a certain criteria. Take a family tree (a simple hierarchy) DSML as example; if we detect that two people are cousins, we can create a visualization that shows this by placing an arrow between them or similar.

Figure 2.11 shows Obeo's own code generation facility Acceleo [1]. Acceleo is further reviewed in section 3.6.2.

Obeo Designer is split up in three products, based on what you need:

Architect Edition Full set of features. Suitable for creating editors and generators.

Developer Edition Not possible to create editors. Intended for code generation template developers.

Standard Edition Not possible to create either editor nor templates. This is edition is intended for *using* the editors and generators.

2.5.3 MetaCase MetaEdit+

MetaCase's MetaEdit+ [35] products are one of the most mature solutions on the market today. The company started out in 1991, having its roots in work from the University of Jyväskylä in Finland. The schema-definition language is based on the *Object-Property-Role-Relationship (OPRR)* model. A MetaEdit+ metamodel consists of three parts; a GOPRR model, a code generator (specified with a custom DSL), and a graphical notation for the OPRR model. MetaEdit+ is split up in two different products, MetaEdit+ Workbench and MetaEdit+ Modeler. The workbench is used by the "metamodeler" to create a language, generator and graphical notation. The MetaEdit+ Modeler is used by developers to utilize the languages created in the Workbench, or with a pre-defined metamodel as basis. This follows the workflow shown in figure 2.5.

Figure 2.14 shows the editor where metamodels are created using *GOPRR* which consists of the following modelling concepts [36]:

Graph Specifies a modelling language.

Objects This is the main type which is the node in the graphs. Used to model e.g. classes and states.

Object set A collection of objects.

Property This defines a property on a object. E.g name and ID.

Relationship Connection between two or more objects. The relationships are connected to objects with roles. This is equivalent to an association in UML.

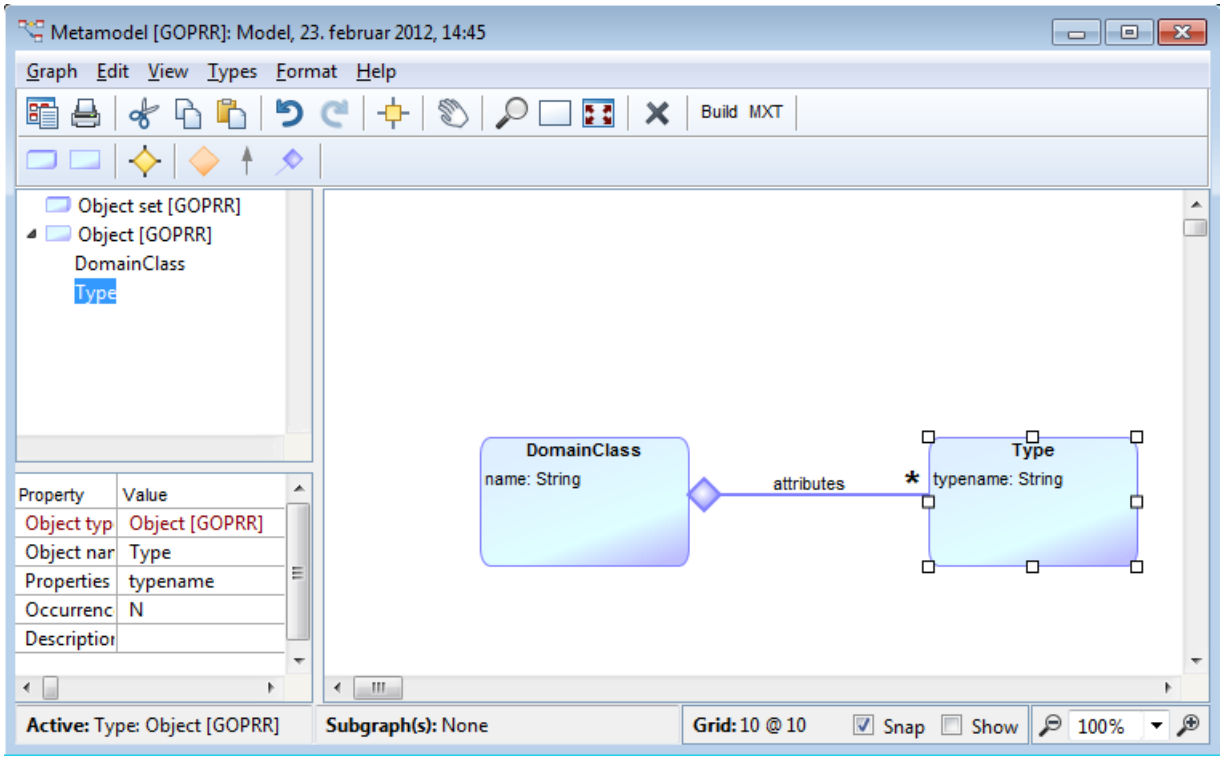

Figure 2.12: Figure shows the editor for metamodelling.

Role A role defines how an object behaves in a relationship. Each object in the relationship has a defined role. E.g. in an inheritance relationship, you have two roles: ancestors and descendants.

GOPRR follows the four level hierarchy the same way Ecore does. You may model your system directly and "skip" a level, or specify another language. MetaCase includes a lot of languages for some more or less popular domains for reference and use. Among them we find DSMLs for insurance, UML, state machines, family trees, home automation etc.

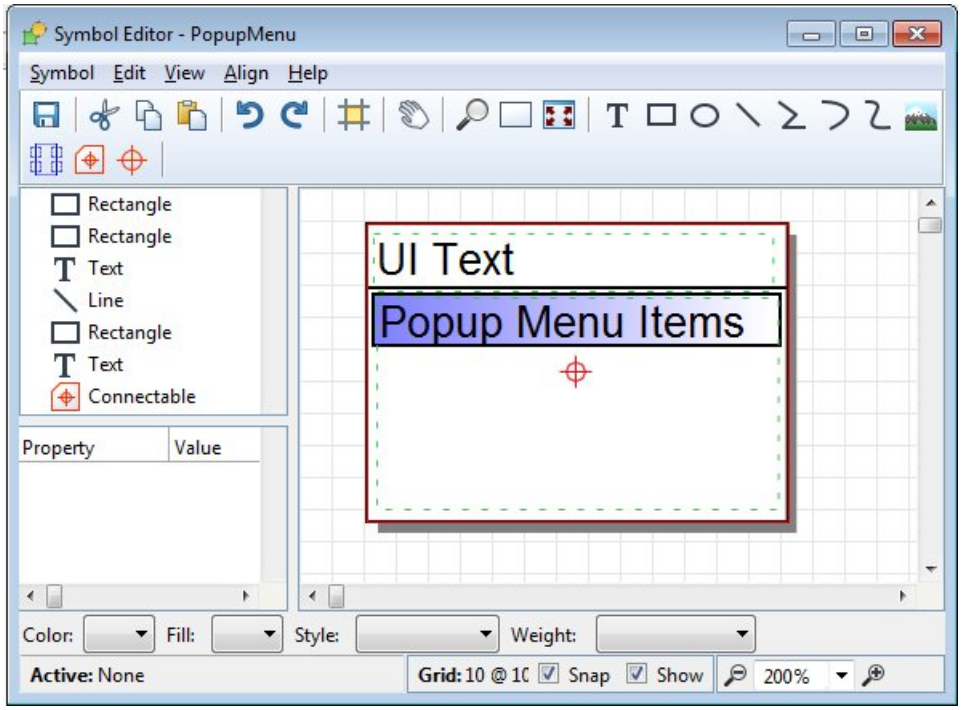

Figure 2.13: Figure shows the editor for creating custom symbols.

One of the biggest features in MetaEdit+ is the ability to create custom symbols for the metamodel. These symbols can greatly improve the modelling experience for users of the DSML. Figure 2.13 shows the symbol editor where it is possible to draw a symbol or import graphics. This particular example shows a menu entry with a list of items in a Symbian Series 60 application DSML. The symbols can utilize particular aspects of the object entities.

Figure 2.14 shows the creation of a Series 60 mobile application using the DSML defined in MetaEdit+.

Figure 2.15 shows the code generation template editor for the Series 60 example in 2.14. A metamodel can have several code generators associated with it.

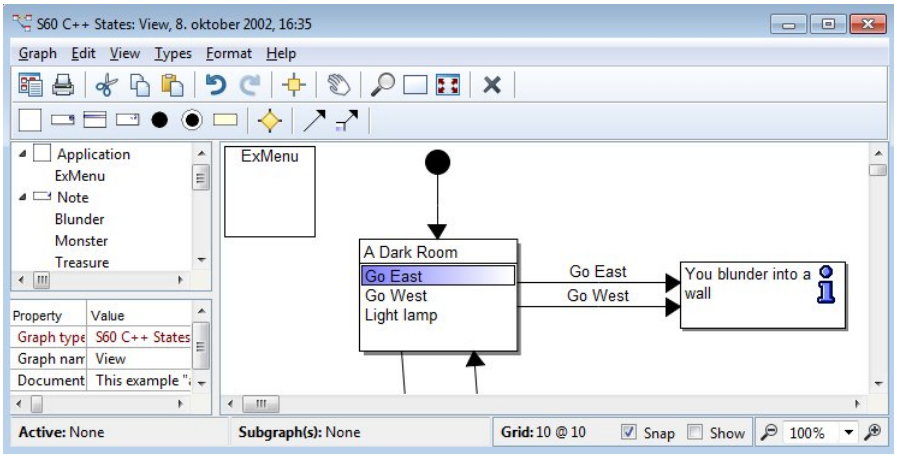

Figure 2.14: Editing an instance model of the Series 60 language in MetaEdit+.

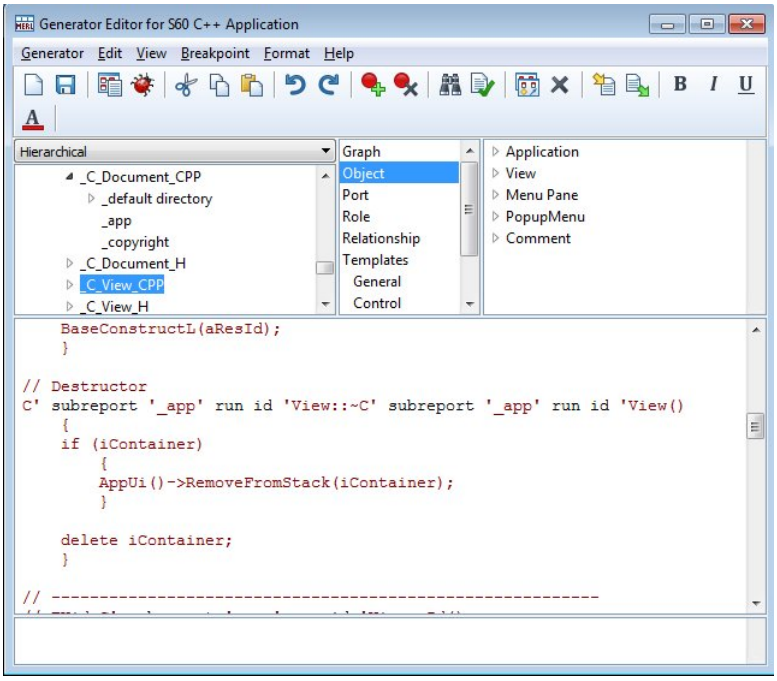

Figure 2.15: Template editing in MetaEdit+.

Constraints in MetaEdit+ are created inside the model, which means that there is no support for *attached constraints* (e.g. OCL). Unfortunately all constraints are pre-defined and there is no possiblity of creating custom constraints with corresponding symbol. All constraints are validated when modelling. A few of the pre-defined constraints include:

Multiplicity

Uniqueness

Connectivity Constraints

Explosions and Decomposition
A state in state diagram may be expanded into a separate diagram.

The MetaEdit+ products are stand-alone proprietary software and is not built on Eclipse or similar platforms. It does however provide plug-ins for both Visual Studio and Eclipse with its new 5.0 release. MetaEdit+ supports Windows, Linux and Mac, but the latest beta release (which the illustrations are from) only supports Windows.

2.6 Diagram Predicate Framework

Diagram Predicate Framework (DPF) is an ongoing project at Bergen University College and the University of Bergen. The project started in 2006 with the aim to create a formal approach to MDE, with features as metamodelling, model transformations and model management. The framework is based on the Generalised Sketches formalism by Zinovy Diskin [11], and relies heavliy on mathematical concepts like category theory and graph transformations.

DPF facilitates a completely diagrammatic approach to MDE. DPF provides a multi-layer diagrammatic metamodelling hierarchy, where the need for attached constraints are removed. One of DPF's strengths is its general nature; it can be used as pattern to model other modelling language like UML, petri-nets and ER diagrams [48].

- A DPF model consists of a *specification* $\mathfrak{S}$ which consists of an underlying graph S, along with a set $C^{\mathfrak{S}}$ of *atomic constraints* (π, δ).
- A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicate symbols Π^{Σ}.
- The constraints are instances of predicates Π^{Σ}.

A *predicate* consists of a symbol, shape graph, visualization and a semantic interpretation. Predicates are defined within a *signature*. Listing 2.3 shows the XML based definition of a multiplicity predicate. It defines a

```
<predicates symbol="[mult(m,n)]">
  <shape id="41921737-3a3a-4404-8d2a-a4ffe39da992" name="Default name">
    <nodes id="0867fde8-4eb6-4d42-be2d-715540a584d6" name="n_1"/>
    <nodes id="c4cce51b-3ab0-4e96-970a-18849131cbb3" name="n_2"/>
    <arrows id="554183df-2788-4469-ba63-84973bbb1d17"
        target="//@predicates.10/@shape/@nodes.1"
        source="//@predicates.10/@shape/@nodes.0" name="a_1"/>
  </shape>
  <semanticsValidator xsi:type="no.hib.dpf.core:MultiplicitySemantics"/>
</predicates>
```

Listing 2.3: A predicate for a multiplicity constraint.

shape graph with two nodes and an arrow, a textual symbol `[mult(m,n)]`, and the semantic interpretation which is a Java based validator.

A central concept in DPF is graphs and graph homomorphisms. A graph homomorphism φ is a mapping from a graph G to another graph H, where the mapping preserves the source and target of each arrow. An atomic constraint (π, δ) in DPF consists of a predicate symbol as well as a graph homomorphism which defines what parts of the graph it constrains.

DPF defines two types of conformance relations which is *typed by* and *conforms to*. A meta-level $\mathfrak{S}_i$ is typed by the meta-level above $\mathfrak{S}_{i+1}$ if there is a graph homomorphism (also called *typing morphism*) between the two meta-levels' underlying graphs: $\iota[i] : S[i] \rightarrow S[i+1]$. A specification $\mathfrak{S}_i$ at metalevel i is said to conform to a specification $\mathfrak{S}_{i+1}$ at metalevel $i+1$ if there exists a typing morphism $\iota[i] : S[i] \rightarrow S[i+1]$ such that $(S[i], \iota[i])$ is a valid instance of $\mathfrak{S}_{i+1}$; i.e. $\iota[i]$ satisfies the atomic constraints $C^{\mathfrak{S}_{i+1}}$ [44] [47]. See [47] [46] for further discussions on DPF.

2.7 DPF Editor

The DPF Editor is the reference implementation of DPF and its concepts [13]. There have been a few attempts the last years to implement earlier versions of DPF. The first attempt was performed by Ørjan Hatland in 2006 [24], a tool based on Microsoft .NET technology. This implementation was never completed and was not considered to be a good foundation for further development. In 2008, Stian Skjerveggen began the work on an Eclipse based solution [49]. The core technology in this project was Graphical Modelling Framework (GMF) [52], a framework that facilitates generation of graphical editors and tooling, based on EMF and Graphical Editing Framework (GEF). GEF facilitates creating rich graphical editors in Eclipse. This technology is somewhat "low-level", and requires a bit of work to achieve the same level of functionality which GMF can generate. GMF was at the end of the project deemed unsuitable for an implementation of DPF.

In 2010, Øyvind Bech and Dag Viggo Lokøen started the work on the

current implementation of DPF, the DPF Editor [4]. It is written from scratch using EMF and GEF and supports the most essential features concerning metamodelling:

- Graph based projectional editing of models
- Storing/loading the models as XMI
- Arbitrary number of metalevels
- Checks typing between metalevels if there exists a graph homomorphism
- Checks constraints between metalevels
- Creation of predicates and corresponding Java validators
- Different simple symbols on nodes (circle, ellipse, rectangle etc.)[2]

None of the previous attempts to create DPF based software has reached a level of maturity where facilitating code generation was considered.

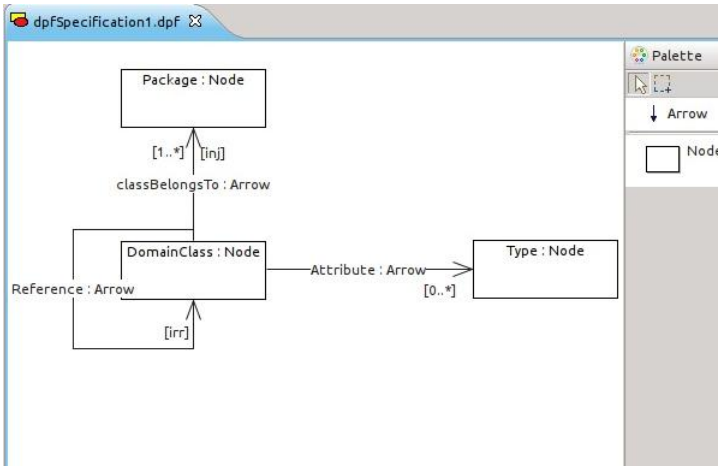

Figure 2.16: Creating a DSML in the DPF Editor.

Figure 2.16 shows the editor view in the DPF Editor. We start off with only `Node` and `Arrow` when creating a DSML. The figure depicts a simple language for creating domain classes in web application. This model forms the basis for the tool demonstration i chapter 5 and will then be explained further. Figure 2.17 shows a subset of the DSML in figure 2.16 and how the different metalayers are typed by each other. The dotted lines shows conformance between the types levels.

[2]The creation of predicates and visualizations are later work by Ph.D. student Xiaoliang Wang.

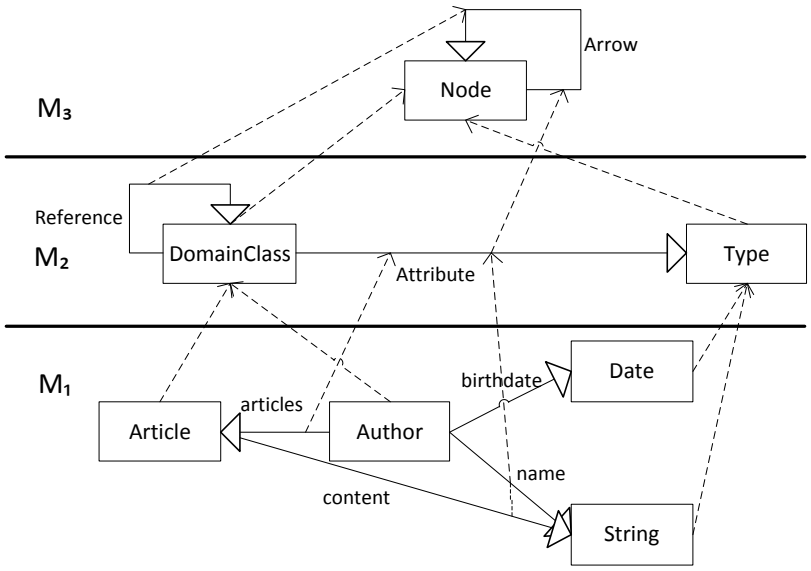

Figure 2.17: Shows how the different DPF layers of a simple domain class DSML conforms to each other.

2.8 Comparison

The evaluated tools has its strengths and weaknesses from a language workbench point of view. The concept of a language workbench is a pragmatic one, and this emphasizes usability of the tools. Even though EMF, Obeo Designer and MetaEdit+ are all mature products with a lot of users, they do not necessarily do everything right. A common problem is messy unintuitive user interfaces which significantly raises the bar for learning the software.

The most complete (and most mature) tool, is MetaCase's MetaEdit+. It supports a wide variety of pre-defined popular modelling languages, but also has its own schema definition language, GOPRR. Obeo Designer improves on the shortcomings of plain EMF, with the possiblity of creating custom visualizations and support for code generation on the model level with Acceleo. Table 2.1 shows that a common trait between the three evaluated solutions; all support different forms of projections. The *Projection Types* row use capital letters for describing a projection type: D for diagrammatic, H for hierarchy, M for matrix and T for table.

Of the reviewed solutions we see that only the DPF Editor provides multi-level metamodelling, while the other adheres to the levels of MOF. As described, all three reviewed tools have a fixed number of metalevels. The metalevels may vary, because one can define e.g. UML through MOF's modelling facilities as shown in figure 2.4. When it comes to constraints, we see

that EMF only have support for *structural constraints* inside the model and is dependent on external functionality for more advanced constraints. Since Obeo Designer is built around EMF, the same applies. MetaEdit+ has, like the DPF Editor, support for more advanced constraints within the model. There is however no option for defining constraints like in the DPF Editor, as they are fixed.

The DPF project can learn from other tools as there is a lot of room for improvement. The lessons learned are both good and bad; the tools studied all have usability issues, mostly related to complicated user interfaces. Having to use OCL for constraints also complicates the modelling process. Ideas like the symbol editor and multiple ways of projecting the model are some of the interesting features which the other tools more or less provide.

A "projection" which none of the tools provide, are textual representation of the models (beyond serializing). The *Language Workbench Competition (LWC)* which is a part of the *Code Generation* [31] conference, refers to a language workbench that supports both diagrammatic and textual representation of a model as "the one language workbench to rule them all"[3].

Tool	**EMF**	**Obeo Designer**	**MetaCase MetaEdit+**	**DPF**
Schema Definition Language	Ecore	Ecore	GOPRR	DPF
Symbol Editor	No	No	Yes	No
Projection Types	D/H[4]	D/H/M/T	D/H/M/T	D
Code Generation Facility	Yes[5]	Yes (Acceleo)	Yes	No
Multi-level metamodelling	No	No	No	Yes
Constraints	OCL	OCL	Model-based/Fixed	Model-based/Custom

Table 2.1: Language workbench features

[3] Florian Mantz has created a textual version of DPF in his Ph.D. thesis work. This work is currently not integrated with the DPF Editor.

[4] EMF provides a diagrammatic view for the metamodel, but not for the models.

[5] EMF generates editors and code for your metamodel, but does not facilitate code generation from the instance model.

Chapter 3

Code Generation

This chapter will give an understanding of the concept code generation, as well as the motivation behind it. The section Framework Analysis (3.6) will evaluate a few chosen code generation solutions, with a comparison at the end. Lastly, the Xpand code generation framework will be introduced.

3.1 What is Code Generation?

Code generation, also called *automatic programming*, is the process of automatically generate source code from some kind of specification (e.g. models). The user specifies *what* to do, the computer generates source code which perform the task. Generating code is a common practice in todays software development, where modern IDEs in particular take advantage of this technique. The popularity of large middleware platforms, such as Java EE and Spring, has shown the advantages of code generation with its extensive need for configuration. These configuration files are often written in XML and are tedious and time consuming to write by hand. Another example is project wizards in IDEs which generate the needed configuration and templates for your project. The focus in this thesis will not be on generating specific configuration files for an arbitrary framework, instead we will focus on a general solution for creating production code from domain-specific models. These models can indeed result in build scripts or XML, however we want to generate production grade code for certain aspects of a system.

Generating code is done through a model transformation, more specifically model-to-text transformations. Through a set of transformation rules, we generate text which results in usable code. A transformation rule could be an expression in a template, or simply plain Java code that outputs a string based on some satisfied statement. In language workbenches code generation is very common, as the tool itself usually do not run the DSML which is created. DSLs on the other hand, is very often used as a wrapper around an API.

3.2 Why use Code Generation?

Using code generation yields a lot of advantages not only in the context of MDE, but also as a general technique in the programmers toolbelt. Herrington [25] lists a few:

Consistency The generated code has a consistent feel, using the same naming conventions and style. This results in familiar interfaces for the users.

Quality Having many programmers creating handwritten code may result in an inconsistent code base that must be updated manually on API changes. If this code can be generated, the generator has the advantage of generating all the classes at once which reflects the changes made.

A single point of knowledge A change within e.g. a database, can have a chain reaction in your project; code, documentation and build scripts needs to be changed. This can be avoided using code generation.

More time for development After spending the initial time-cost of creating proper models and generators, one can save a lot of development time. This is time that can be spent creating new features, and/or fixing existing ones.

Abstraction The power of domain-specific modelling enables us to express more functionality in a consise manner.

Even though the benefits are big, there are always concerns about the code being generated, such as quality and efficiency. In the CASE-tools era, the fixed general modelling languages made it difficult to generate anything other than static code/schema definitions. With domain-specific modelling we are now able to generate both the static and functional code. This contrast may be the source to some of the skepticism we see today. The efficiency concern is about generating code for e.g. an embedded system with very limited resources, where handwritten code often is optimized with different workarounds and hacks.

These concerns are as old as computing; when programming languages with a higher abstraction level came to existence, there was always concern that these languages could never generate code as good as hand written machinecode. History has shown that if given enough time, compilers become so advanced and effective that it is not feasible to hand write low level code for the sake of efficiency. Compilers do a lot of optimizing on its own, and any particular hacks used could easily be included in the generator, and thus creating as efficient code as the handwritten counterpart. The expressiveness of models gives us the power to create functional code which is safe and efficient; it really comes down to how the model and generator is defined.

Tolvanen and Kelly [29] claims boldly:

> "When code is produced by a generator made by an experienced developer, it will always produce better code than the average programmer writes manually."

3.3 Creating a Code Generator

In a MDE environment, we need our models understood by a computer. The usual way of doing this is to either generate code, generate an interpreter or populate a semantic model. The last option is more likely to be used with textual DSLs rather than a DSML created in a language workbench, and is less relevant in the context of the DPF Editor. When creating a code generator, there are two things that is needed; the input (DSML) and a sample output of what you want to generate. After all, code generation is a model transformation. Through our generator, we express rules that maps our input to the output. If you attempt to create the generator first, you can end up creating a language that is to general for your domain. When you have both the output and the model in place, the generator creation will be a much simpler task.

Transformer Generation This method uses a programmatic approach, where a generator traverses an input model directly in the programming language, and creates code statements. This approach is suitable when targeting one specific environment (although it can be generalized) and most of the code is generated from the model, i.e. low amount of static code.

```
public String buildClass(Node n) {
    StringBuffer buf = new StringBuffer();

    buf.append("public class " + n.getName() + " {");

    for(Attribute a : n.getAttributes()) {
        buf.append("private " + a.getType() + " " + a.getName() ";");
    }

    buf.append("public " n.getName() + "{");
    buf.append("}");
    buf.append("}");

    return buf.toString();
}
```

Listing 3.1: Example showing a Java method building code.

Listing 3.1 shows a simple Java example where we build a class with private instance variables and a default constructor. Although easy

to understand, these classes can grow very big when generating advanced functionality. Another issue is formatting the output; one needs to either format it as you create the code, or after. Formatting as you go, requires a way to keep track of indents and newlines. Formatting after the code is generated can be achieved by exploiting an existing formatter, like what Eclipse Java Development Tools provides, instead of creating one from scratch.

Templated Generation Templated generation consists of a *template engine*, a template language and sometimes a standard library that provides simple functionality for use within the templates. A template engine uses external files, called the templates, which uses a template language. This language uses markers to identify where certain parts of the code should be insterted. A template based solution is recommended when you have a lot of static code and/or configuration code, or you want to target several environments. This approach is the one chosen for this thesis and is investigated more in section 3.6.

Another issue that needs to be addressed is what you want to generate with your generator. Most of the time, there is no need to generate a complete runnable application. A common practice is to generate code for a *domain framework* in your target environment. If no such framework exists, it might be a good idea to investigate the possibility of creating one. Such frameworks can greatly simplify the generated code by abstracting away repeated concepts.

Model-to-model Transformations (M2M)

Creating textual output from a generator is not always the best solution. Sometimes generating a new model might fit the development workflow better. A motivation for such a scenario is that the model is not descriptive enough to directly generate code. A real world example of this is the MDA workflow, where PIMs are transformed into PSMs as a step on the path to textual output.

3.4 Editing Generated Code

Ideally the generated code should work on its own, without any editing. This is a consequence of the effort put in when creating the DSML and associated tools. According to Kelly and Tolvanen [29], a proper DSML should be analogous to compiling general programming language code. The reality is somewhat different as the need for mixing generated and handwritten code is not seldom. The general rules are not to modify generated code and keeping generated code separated from handwritten code. To achieve these goals there are a few techniques [21]:

Keep generated code in a separate folder
This reduces the risk of editing code and losing the changes upon re-generation.

Partial classes
Partial classes provides the option of splitting classes into separate files. This is supported by Smalltalk and C#. This is however not supported in Java, C++ etc.

Generation gap pattern
Generation gap pattern is probably the most general solution for this problem. It uses inheritance to extend functionality; a generated class acts as a superclass which you inherit with a handwritten class. The only drawback is the need for relaxing visibility rules in the superclass.

Annotations
This approach lets you mix handwritten and generated code using annotations (Java) or attributes (C#). In some cases this might be the best solution. E.g. when generating code from an EMF model, it is often necessary to hand code some of the functionality in declared operations. It is then possible to use annotations to mark which methods should be generated or not with the `@generated NOT` annotation.

Protected Regions
With protected regions you place a marker, often a start marker and an end marker, in the source code where you want the generator to not re-generate. This can be seen as a general approach to annotations based protection.

3.5 Metamodels and Code Generation

So far the discussion has revolved around general principles and techniques concerning code generation. When using template based solutions (see 3.6) one operate directly on objects in the host language. The template engines were probably not built with MDE in mind and are thus more general. One of the biggest uses are rendering web pages. In the context of models, one would work directly on the model's internal representation, like listing 3.2 shows. There is one big drawback of this approach, and when our solu-

```
<% Node n = (Node)argument;%>
class <%=n.getName()%>
```

Listing 3.2: Simplified Java Emitter Templates (JET) example of using an object directly.

tion is supposed to facilitate code generation for any arbitrary model, it is quite essential: there is no way of expressing domain concepts in a simple elegant manner. Giving the user a template engine and a clean template for

their DSML would work, but do not address the problem at all. We need to express the most essential part of MDE, the *domain concepts*, in such a way that it is intuitive and easy to create code generators.

To express the domain concepts we need to look at the DSML, not the model (instance). The goal is to express an arbitrary DSML and its concepts in a chosen template language. Creating such a solution from the beginning would entail a lot more work than the time-frame of this thesis would allow. Luckily, the Eclipse M2T project has addressed this issue with a few solutions described in the next section.

3.6 Framework Analysis

This section gives a brief overview of the different code generation solutions that was considered in this project. The chosen solution, Xpand, is introduced with a short overview of features. This will be discussed in detail in section 3.7.

3.6.1 Template Engines

Both JET and Velocity are simple template engines, which provide nothing more than an engine, a template language and a "standard library" with functions to alleviate the use of Java code in the templates. Choosing these solutions would entail the creation of an API that exposed the desired functionality; the domain concepts of the DSML and associated operations/methods. Creating an API like this would be a messy complicated solution.

Creating editor support which exposed the DSML properly would not be an easy task to achieve as these engines have no interpreter which could interpret the models during runtime. Figure 3.1 depicts the idea behind a template engine. You have some kind of data source (e.g. a model) which get matched towards a template that have markers on where to insert data. After the compilation, the engine outputs what is defined in the template.

JET

Java Emitter Templates is a template engine in the M2T (model-to-text) project within the Eclipse ecosystem. JET provides a well known syntax based on JSP, which has a standard tag library included, that can be extended with custom tag libraries. It comes with editor support and integration with the Eclipse UI.

Listing 3.3 shows a very basic example on how a JET file could look like. We also see that objects are used directly, and not interpreted as the generator frameworks.

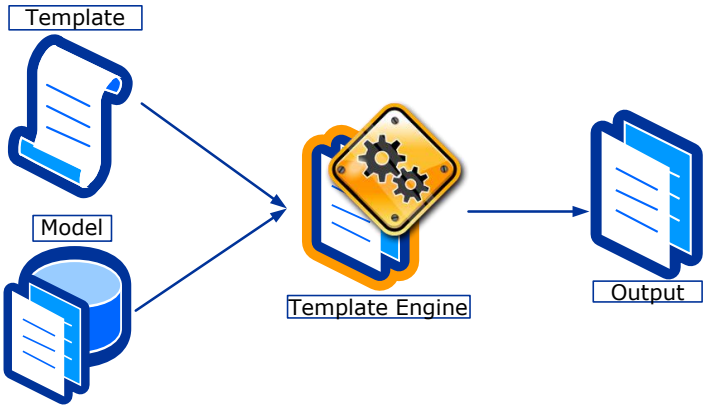

Figure 3.1: Figure shows a conceptual view of a template engine.

```
<%@ jet package="generator.dpf"
  class="DpfGen"
  imports ="generator.dpf.blog.Article no.hib.dpf.core;"
%>
<% Article art = (Article)argument;%>
public class <%=art.getName()%> {
    public <%=art.getName()%>() {

    }
}
```

Listing 3.3: Simple example showing the JSP like syntax in JET

Apache Velocity

Velocity shares a lot of the same properties that JET possesses. The main difference is the template language, where Velocity provides its own. The language is simple, yet powerful, and provides the same functionality that JET can offer. If the standard functionality is not enough, there are subprojects like VelocityTools that solves problems like date and number formatting, math operations and more. There is a strong focus on separating code from the templates, thus enforcing patterns like MVC. There is no editor support for Eclipse out of the box, this is achieved using third party plug-ins.

```
<HTML>
<BODY>
Hello $customer.Name!
<table>
#foreach( $mud in $mudsOnSpecial )
   #if ( $customer.hasPurchased($mud) )
      <tr>
        <td>
          $flogger.getPromo( $mud )
        </td>
      </tr>
   #end
#end
</table>
```

Listing 3.4: Example showing the Velocity template language

3.6.2 Code Generation Frameworks

A code generation framework is more than a simple template engine. The biggest difference lies with the additional tooling around the engine, like debugging, editor support and profiling of templates. The problem identified in section 3.5 is addressed in these tools through a built-in interpreter which interprets the DSML, and creates a template editing environment where the editor support reflects the domain concepts of the DSML.

Acceleo

Acceleo [1] is the product of Obeo, the same company that is behind Obeo Designer. The project has been in development since 2006, and was incorporated into the Eclipse M2T project in 2009. Acceleo is the reference implementation of Object Management Group's MOF model to text transformation language (MOFM2T) [41]. It has full integration with EMF, which

means you can use Ecore, UML2, etc. The framework has support for debugging and profiling templates, as well as editor support with auto completion and content assist. The debugger has its own Eclipse view where you can debug the templates directly using breakpoints. You can also step over/into/return functions like Eclipse JDT provides for Java.

Another notable feature is the generator module system. A generator module is a plug-in with pre-defined generator templates for a particular domain. This enables the users of Acceleo to take advantage of available generators for popular domains. The drawback of this system is that the generators are (of course) bound to a specific type of metamodel, and might not be tailored to the specified requirements.

```
[template public generate(aClass : Class)]
  [file (aClass.name.concat('.java'), false)]
    public class [aClass.name.toUpperFirst()/] {
    [for (p: Property | aClass.attribute) separator('\n')]
      private [p.type.name/] [p.name/];
    [/for]
  [/file]
[/template]
```

Listing 3.5: Example showing the MOFM2T language used in Acceleo

Xpand

Xpand [54] is a framework under the Eclipse M2T project that facilitates MDE activities such as code generation, model validation and model transformations. The company behind Xpand is Itemis [27], a german consulting company focusing on utilizing MDE in projects. Xpand has a wide variety of features which makes it a good fit for this projects requirements. Most notably, you can create editors (with code completion and syntax highlighting) from metamodels specified in EMF Ecore, UML2, XSD and JavaBeans. Besides the predefined metamodels, it is possible to define a custom metamodel that is based on an arbitrary modelling language. Xpand is discussed in detail in section 3.7

Comparison

Table 3.1 gives an overview of the different features in the evaluated solutions:

Template Language The language used in the templates.

Editor Support All the considered solutions provide some kind of editor support with syntax highlighting and template validation. The difference from the template engines to generator frameworks lies with how

this editor support is provided. Xpand and Acceleo interprets the input models and provide an environment for creating templates, extensions and validation based on the metamodel. In a DPF context this means you operate on the concepts of the DSML, rather than the instance model's actual Java objects. This is discussed in detail in section 3.7.

Custom Metamodel Xpand and Acceleo provide support for EMF and UML models out of the box. This means an UML model will act as a metamodel, i.e. expose its domain concepts in the editor environment. This is discussed in detail in section 3.7.

Profiler/Debugger Profilers are used to optimize your templates and extensions, and gives an overview of which parts of the system that is slow. Both Acceleo and Xpand has a profiler. A debugger is useful for finding bugs inside the templates. Acceleo allows you to debug templates like Java classes in Eclipse, with a debug view, breakpoints and stepping over/into/return functions.

Tool	Template Language	Editor Support	Custom Metamodel	Profiler/ Debugger
Apache Velocity	VTL	No	No	No
Eclipse JET	JSP	No	No	No
Acceleo	MOFM2T	Yes	No	Yes
Xpand	Xpand	Yes	Yes	No

Table 3.1: Feature comparison in template engines

The reason for choosing Xpand is together with the rich tooling, the possibility of creating a custom metamodel. The included profiler is a welcomed feature, but was not critical to the choice of solution. The choice of template language was not crucial either, but a language with its foundation in a standard is a positive trait. Acceleo is providing the same functionality as Xpand with some minor differences in the tooling. A simpler choice would have been either JET or Velocity, but this would be a poor solution for the end-users because of the lack of tooling. Both Acceleo and Xpand provides a "sandboxed" editor environment, which means the template coder only has access to the DSML concepts which is defined through a Xpand/Acceleo metamodel.

3.7 Xpand

Xpand is created and maintained by the german consulting company Itemis [27]. Before the project was included into the Eclipse M2T project, it was called *openArchitectureware (oAW)* [43]. The functionality is the same, but the project has been split up into different Eclipse projects. Xpand is under the M2T project, while the *Modeling Workflow Engine (MWE)* is under *Eclipse*

Modeling Framework Technology (EMFT). The workflow engine connects the different components in Xpand and executes them. Even though the components have a clear separation, we will use Xpand for all of the included technologies, and explain them when needed.

The biggest difference from a simple template engine, is the built-in interpreter that interprets a DSML and provides functionality based on what a Xpand metamodel dictates. In Xpand, the metamodel is the internal mapping from types in the DSML to types that Xpand understands. Xpand comes with metamodels for the most popular modelling facilities today; EMF, UML2, XSD and plain Java classes. Although these metamodels covers the most used languages, it lacks support for DSMLs specified in anything besides what Xpand has to offer. There are a lot of proprietary DSMLs out there which is not modeled in any of the mentioned languages. Acceleo solves this by saying you need to specify your DSML in EMF/EMOF. This is not a generic solution, and requires the users to create a new EMF/EMOF implementation for each new language that is created. A solution to this problem is to create a model-to-model transformation which can transform your DSML to EMF. The disadvantage is that you are still restricted to the functionality that the Acceleo metamodel defines (which is the EMF concepts), when you perhaps would want to define your own.

DPF is a framework for creating DSMLs, and as discussed in chapter 2, DPF Editor falls under the language workbench category. The use case for the DPF project is to generate tooling based on a specified DSML; we want a generic way to define generators for an arbitrary DSML. This use case is a bit narrow, and is probably why Acceleo has not facilitated custom metamodels. Xpand fits our criteria nicely with the ability to create custom metamodels with custom attributes and operations. Xpand lets us create a separate API used only in the templates and extensions, on top of the existing functionality that resides in the DPF Core API. The metamodel defines a mapping from DPF types to Xpand types which is reflected in the template/extension editor.

Xpand is a feature rich framework with features as *aspect-oriented programming (AOP)*, functional extensions, model-to-model transformations, model-to-text transformations and model validation. The framework provides three different languages that has separate functionality; Xpand, Xtend and Check.

Xpand

Xpand is the template language that controls the output of the generator. Apart from the usual control flow statements, it supports lazy evaluation, `let` statements, aspect oriented programming and extensions created using Xtend and/or Java.

```
«IMPORT dpf
«EXTENSION org::eclipse::xtend::util::stdlib::io»

«DEFINE main FOR dpf::Specification»
  «EXPAND graph FOR this.graph»
«ENDDEFINE»

«DEFINE graph FOR dpf::Graph»
  «EXPAND domainclasses FOREACH this.getDomainClasses()»
«ENDDEFINE»

«DEFINE domainclasses FOR dpf::DomainClass»
  «syserr(this.name)»
«ENDDEFINE»
```

Listing 3.6: Example showing the Xpand syntax

This listing shows a simple template where we traverse our DPF Specification and print out the name of a DSML type called *DomainClass*. We also see how the inclusion of extensions are performed. These extensions are defined in Xtend and/or Java.

Xtend

Xtend is used as an extension language. It follows the functional paradigm and provides features like type inference, recursion, caching of methods and calling external Java extensions. It also provides ways to perform model-to-model transformations. This language helps to enforce the separation between program logic and template code; it is strongly encouraged to perform logic in an extension, and call the extension from the template. Such extensions are called *embedment helpers* [21].

```
importDate(dpf::DomainClass this) :
  if this.getAAttributes().exists(e | e.target.name == "Date") then
        "import java.util.Date;";

String paramList(List[dpf::DomainClass] this) :
  JAVA no.hib.dpf.codegen.generator.extensions.
        StringUtil.getAttributeList(java.util.List);
```

Listing 3.7: Example showing Xtend extensions

Listing 3.7 shows the Xtend language, with two Xtend methods. `importDate` shows how we detect a *Date* type defined in our DPF model. If such a type exists, we return a string that contains an import statement. This example demonstrates the type inference by not declaring a return value. It also showcases some of the functional syntax with the `exists` statement, which returns a boolean value based on the expression within. The last example shows how a Java extension is called from Xtend. Type inference from these extensions are not supported,

so we explicitly need to declare the return type. We also observe the need to use fully qualified method names for the extension class. This method in particular returns a `String` that has been constructed from a list of `DomainClass` types.

Check

The Check language handles constraint checking on the model.

```
import dpf;

context Attribute ERROR
  "Name of " + name + "too short." : name.length > 1;
```

Listing 3.8: Example showing a simple constraint check

Listing 3.8 shows a simple constraint check. The first line imports the metamodel. The second and third line checks the length of property *name* in `Attribute`, and returns an `ERROR` if the test fails. Denoting the Check statement with `ERROR` will abort the workflow and print the error to a console. Alternatively, one can use `WARNING`, which will print the warning and continue the workflow. In the DPF Editor the constraint checking is done within the DPF model, but all validation could happen in the generation phase using Check if desired.

These languages are built upon the same type system, and share a lot of the syntax. The shared syntax is called the *expression sub-language* and provides the basic flow control constructs of all the three languages (e.g. if-conditions, switch statements, casting). The expression sub-language syntax is a mix between Java and OCL.

Workflow

The Modeling Workflow Engine (MWE) provides a way to execute different Eclipse modelling components in a sequential manner. The execution environment can be inside Eclipse or standalone. Modelling components do not have to be part of the Xpand framework; a component needs to adhere to an interface which MWE provides.

A workflow is defined in a XML file, where each component has its own configuration. Some of the components in Xpand are shown in listing 3.9. A workflow do not have to include all of the components shown; a typical workflow for generating code only consists of a reader and generator component.

Listing 3.9 shows a workflow file where an Ecore model is used for input:

1. The first lines declares properties for model location and the output folder for the results.
2. `StandaloneSetup` is used to initiate EMF specific functionality, like `platform:/resource` URIs.
3. Initiation of the Xpand metamodel. The handle `mm_emf` is used to refer the metamodel throughout the workflow definition.
4. The `Reader` component is used to read the input model's Ecore, and then create a *slot* which stores the model in a named attribute.
5. This is the configuration of the `Check` component. We define the EMF metamodel to operate on, as well as the name of the Check file.
6. The `Xtend` component performs a model-to-model transformation based on the rules specified in the Xtend file.
7. The generator component is the one generating code. We start off by defining the EMF metamodel, then we define the entry point in the templates, and also refer the model that the template should be run on. Finally, we define an outlet path where the resulting code should be written to. We also apply a postprocessor to our output, in this case a `JavaBeautifier`. The `JavaBeautifier` is a class which format the generator output to our needs. There are postprocessors included in Xpand for C++, Java and XML.

It is important to point out that the order of the components matter. Before using a Xpand metamodel it is necessary to use its corresponding reader component first (if it has one). Using a metamodel dependent component with an uninitalized metamodel will result in errors.

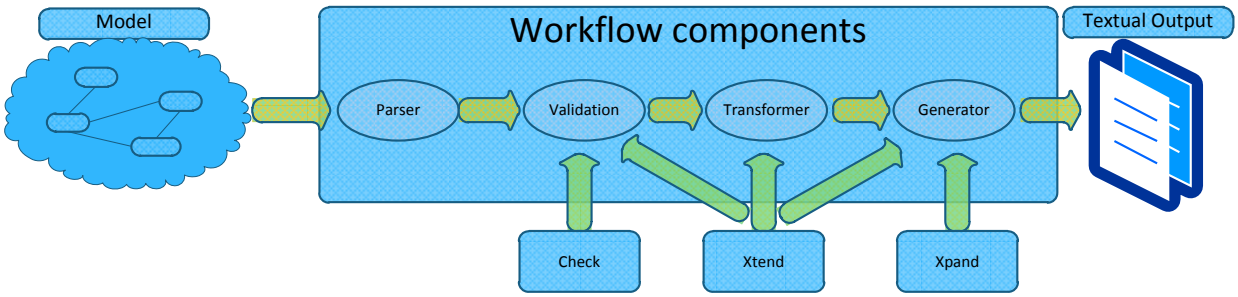

Figure 3.2: Figure shows how the workflow engine works.

Figure 3.2 depicts a workflow in Xpand. Although a usual workflow, all of the components are optional, and could be replaced by something else.

```
    <?xml version="1.0"?>
    <workflow>
(1)         <property name="model" value="path/to/model/Model.xmi" />
            <property name="src-gen" value="src-gen" />

(2)         <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup" >
                    <platformUri value=".."/>
            </bean>

(3)         <bean id="mm_emf" class="org.eclipse.xtend.typesystem.emf.
    EmfRegistryMetaModel"/>

(4)         <component class="org.eclipse.emf.mwe.utils.Reader">
                    <uri value="platform:/resource/${model}" />
                    <modelSlot value="model" />
            </component>

(5)         <component class="org.eclipse.xtend.check.CheckComponent">
                    <metaModel idRef="mm_emf"/>
                    <checkFile value="metamodel::Checks" />
                    <emfAllChildrenSlot value="model" />
            </component>

(6)         <component class="org.eclipse.xtend.XtendComponent">
                    <metaModel class="org.eclipse.xtend.typesystem.emf
                        .EmfRegistryMetaModel">
                          <metaModelFile value="${model}"/>
                    </metaModel>
                    <invoke value="test::Trafo::duplicate(rootElement)
                        "/>
                    <outputSlot value="newModel"/>
            </component>

(7)         <component class="org.eclipse.xpand2.Generator">
                    <metaModel idRef="mm_emf"/>
                    <expand
                            value="template::Template::main FOR model"
                                 />
                    <outlet path="${src-gen}" >
                            <postprocessor class="org.eclipse.xpand2.
                                output.JavaBeautifier" />
                    </outlet>
            </component>
    </workflow>
```

Listing 3.9: An example MWE workflow file

3.7.1 Xtend 2

In December 2011, version 2.2 of Xtend [17] was released. This is Xpand's successor, and is the main focus for Itemis from now on. Xpand will be maintained for a while, but will probably be dropped in favour of Xtend 2 in the longterm.

The naming is a bit confusing, as the Xpand framework has a separate language called Xtend (see listing 3.7). Xtend 2 is a complete rewrite and is not compatible with older versions of Xpand. It is based on Xtext [18], another Eclipse project created by Itemis. Xtext is a language workbench for textual languages where one can create editors and generators from a grammar. Currently Xtend 2 is a part of the Xtext project, but will in time become a separate project. The reason for this re-implementation is mentioned in the lead developer Sven Efftinge's blog [50]; performance issues, poor tooling and shortcuts in the language.

Xtend 2 is a fully-fledged programming language, which compiles to readable Java code. The language aims to reduce the verbosity of Java, and support features like type inference, closures and operator overloading. Generating code is done through what is called *template expressions*, which is the syntax from Xpand embedded into the Xtend 2 language. Instead of the interpreter based approach of Xpand, the template expressions are compiled to Java code. At the time of writing, Xtend 2 do not provide the same functionality as Xpand as it has no support for loading custom models.

Xtend 2 can work on a grammar defined in Xtext, which in its turn has its foundation in EMF Ecore. More information on this works can be found in the Xtext/Xtend documentation [19]. Further information on Xtend 2 can be found at Xtend web site, and Sven Efftinge's blog as well [17] [50].

Chapter 4

Design and Development

This chapter gives an in-depth look at how the implementation of the code generation tool works, and how it fits together with the DPF Editor and the Xpand framework. The different components and their function will be explained. In the end of the chapter there will be an overview of what has been achieved throughout the development, and what the tool's shortcomings are.

In the Xpand documentation, the mapping between a DSML and the Xpand types are referred to as the metamodel. Hence we will use *metamodel* when describing the *DPF Xpand type mapping*. When referring to the Ecore metamodel in which DPF is specified, we will describe it as *DPF Ecore metamodel*.

4.1 Development Process

Development Methodology

This project has chosen to use *Agile* development methodologies. The methodology chosen for this project is Extreme Programming (XP), although following it completely by the book has not been possible. XP was created by Kent Beck [5] which is one of the pioneers in the Agile movement [51]. The reason for choosing XP over other agile methodologies is that it is tought in the MOD251 class at Bergen University College. There are a few constraints which hindered following XP properly; the project had no clearly defined requirements, lack of time and no one else working on the same problem (pair-programming).

In the development phase of the project, the aim was to deliver working software every two weeks coinciding with DPF project meetings. The meetings gave a chance to review what had been done, and the focus area for the next iteration.

Coding Convention

XP dictates that one should use a predefined coding convention before starting the development. Such standards aim to result in code that is consistent from developer to developer. It defines how the code looks, but can also include guidelines on which patterns to use and avoid. Although this project was developed by one developer, it needs to be maintained by someone else in the future.

The code convention used in this project is Eclipse Naming Conventions [15]. This convention defines how Eclipse specific elements should be named (e.g. plug-ins and package names). The code convention in general uses Oracle's own guidelines [28]. The reason for choosing these conventions are that the DPF Editor itself uses them throughout, as well as beeing de-facto standard in the Java ecosystem. Having a completely consistent codebase will improve readability and thus make it easier for new project participants to get started.

Tools

The development process has been aided by different tools to create an environment which enhances productivity and provides structure. An important tool, even when programming alone, is a bug tracker. Using a bug tracker helps structuring ideas, as well as keeping track of defects. The tool chosen for this task was Trac [20], a lightweight Python based bug tracker with wiki functionality, integration with version control systems (VCS) and the possibility of creating milestones from feature requests and bugs. Along with Trac, Mercurial [34] was chosen as the VCS.

The development process was aided by the Eclipse Modeling Tools (Eclipse Java Development Tools with modelling components), using the latest version *Indigo* (3.7) [16]. The metamodel is based on the latest released Xpand SDK (1.1.1) which can be found in the modelling components installer inside Eclipse. Logging is used throughout the project for debugging purposes. The logging facility used is called Apache Log4j [3].

4.2 Project Overview

Naming Components

This project uses the same naming conventions which the DPF Editor code base uses. This means all packages starts with `no.hib.dpf` to denote it is a part of the DPF project. As the sub-project name, `codegen` is chosen. In time there might be other code generation solutions based on other frameworks than Xpand, and it is suitable to put such projects under the same name. Furthermore, as a component name, `xpand` is chosen. This is to denote

that the particular solution is based on Xpand. `metamodel` is chosen as sub-component name following the convention used in the Xpand project.

The following plug-in projects are defined:

`no.hib.dpf.codegen.xpand.metamodel`

`no.hib.dpf.codegen.xpand.metamodel.test`

`no.hib.dpf.codegen.xpand.metamodel.ui`

4.3 Problem Description

As discussed in section 3.5, the DPF Editor is in need for a general solution to creating code generators. Using regular template engines will result in a code generator that would not convey the domain concepts of a DSML. An ideal solution would have the following traits:

Clear expression of domain concepts
: The concepts of the DSML should form the basis for the constructs used in creating templates.

Integration with Eclipse
: Editor support is an important feature which makes the template creation process more user-friendly and intuitive. Features like template debugging and profiling are features which also improves the user experience with the tool.

Standalone generator
: A generator which does not have to many dependencies are more portable and reusable. With dependencies directly on Eclipse, one would make the solution hard to use in other contexts.

These goals are more or less already achieved in Acceleo and Xpand, but only for the predefined model types which are EMF, UML2 and XSD. This chapter will describe a solution that facilitates the use of DPF models in a custom Xpand metamodel.

Listing 4.4 shows an example where we use the DPF Ecore metamodel with the EMF metamodel. For each `DEFINE` block in the templates we need to iterate over *all* nodes of the DPF model, and not the particular collection of nodes conforming to a metatype. This example shows the creation of Java *get* methods from a DSML which contains a `DomainClass` entity. What we want to achieve is a `DEFINE` block where we iterate over *only* the nodes that conforms to the `DomainClass` metatype.

```
«DEFINE gettersAndSetters FOR core::Node»
        «IF this.getTypeName() != "DomainClass"»
                public «this.getTypeName()» «getter(this)»() {
                        return «this.name.toFirstLower()»;
                }
        «ELSE»
                public «this.name.toFirstUpper()» «getter(this)»() {
                        return «this.name.toFirstLower()»;
                }
        «ENDIF»
«ENDDEFINE»
```

Listing 4.1: An example of a Xpand template using the DPF Ecore metamodel as basis.

4.4 Metamodels in Xpand

The introduction of this chapter mentions that metamodels in Xpand are a mapping from types in an actual model language (like Ecore) to the Xpand type system. This naming is somewhat confusing, but one can think of it as a metamodel for Xpand's understanding of a model. The metamodel dictates how the input model should be mapped to the Xpand type system, and what kind of mapping this is.

As stated in section 3.7, Xpand supports a number of predefined metamodels, namely UML2, EMF, XSD and Java beans. Besides these, we have the built-in Xpand types which also can be regarded as a metamodel. In a MWE workflow (from now referred to as workflow, see 3.7) one can take advantage of multiple metamodels at the same time, making each metamodel handle different aspects of the model.

An important note is that the order of the metamodels is crucial. As an example we can define a workflow which uses three different metamodels (from the Xpand documentation [54]): Let us assume that our model element is an instance of the Java type org.eclipse.emf.ecore.EObject and it is a dynamic instance of an EMF EClass Car.

- Built-in metamodel (always first)
- Java beans metamodel
- EMF metamodel

If we want to match `Car` against `EObject` in the EMF metamodel we bump into a problem: the first match we get is from the built-in metamodel where we get the Xpand type `Object`. This is now our *best-fit* as every metatype has to extend `Object`. We then proceed to the Java beans metamodel which will return a `org::eclipse::emf::ecore::EObject` which is a specialized version of `Object`. The EMF metamodel would have returned *Car*, but was unable as we got a match before the metamodel was queried. This example illustrates how important the proper order of metamodels is. If we had

changed the order of the Java beans metamodel and EMF metamodel, the proper value would have been resolved. Note that using e.g. a UML2 and EMF metamodel to handle different parts of a model is a rare scenario. The example shown is relevant because most metamodels take advantage of the JavaBeans metamodel to provide functionality in a Java class without the need for any hand coding.

Figure 4.1 shows how Xpand works in principle. The figure shows how the DSML initializes one or more metamodels which creates custom Xpand types (henceforth referred to as types). More specifically, the DSML are *interpreted* at runtime by Xpand and mapped to types defined in the metamodel. When the workflow is executed, Xpand will parse and validate the templates, this is done with the help from the metamodel which decides which types should be applicable where. When the code generation phase starts, Xpand will match a defined input model against the template. This is done through calls against the metamodel as well.

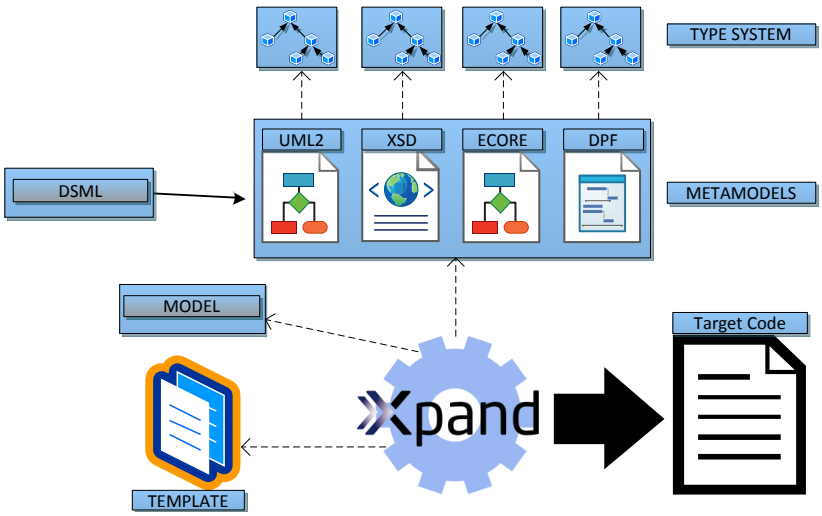

Figure 4.1: Figure shows how the Xpand metamodels work.

A metamodel in Xpand must implement the `MetaModel` interface. The most important methods that needs implementing are the following:

`getKnownTypes()` Returns a set of types which represents all of the types the system know of.

`getType(Object type)` Returns a corresponding type which the parameter object is matched against. The objects which is matched against `getType` are the model objects from either the DSML or instance model. We have to check the object's DPF type to decide how to handle it using the Java operator `instanceof`.

`getTypeForName(String typeName)` When processing a template, there are type identifiers in the different statements. An example of this is:

```
«DEFINE test FOR my::namespace::StateType»
```

The type identifier `my::namespace::StateType` would be queried against the metamodel to retrieve its designated type, or create a new type if it do not exist.

`getNamespaces()` Returns a set of strings which defines what namespaces the metamodel should handle.

4.4.1 The Xpand Type System

The purpose of the type system is to create a common representation of the input models, which can be used as a basis for creating tooling. I.e. rather than creating specialized solutions for a particular model type, we have a general "model" to adhere to. Creating types is also a way to extend the existing functionality in a model; the type system is a *reflection*[1] layer which can be extended with the implementation of metamodels. A type in Xpand contains a name, properties and operations, as well as information about inheritance. The type system in Xpand provides access to types based on the metamodels which are used.

Names have the option of using namespaces, which are delimited using "::". In the EMF metamodel, the package name(s) in a model is used as the namespace(s). E.g. with the DPF Ecore metamodel one would have `no::hib::dpf::core` as the namespace prefix to a type. When using these types in the templates one can import the namespace so that the namespace prefix can be omitted.

When creating a metamodel for Xpand, the types may contain primitives, like string, integer or floats. These types can be wrapped in a tailored solution, e.g. creating a type for floats with operations focused on a particular domain like finance. Often the existing functionality is the best fit to handle primitives as Xpand contains additional functionality for the primitives string, boolean, real and integer. The string class in particular provides methods that is useful in a code generation context, like the "+" operator for string concatenation, regular expressions and more. In addition to the primitives, Xpand has defined a few collection types: Collection, List and Set which corresponds to their Java counterpart (`java.util.*`). To implement a general approach for the collection types, Xpand uses the concept of *parameterized types*[2].

As mentioned a Xpand type contains properties and operations based on reflection. Along with static properties, they are what is called *features*.

[1] Reflection is the process of dynamically load classes and functionality at runtime. [26]

[2] A parameterized type in Xpand is a type which has an inner type. The inner type defines what type a collection can contain and must be a Xpand type.

The features are defined as following:

Attribute
An attribute contains a name and a return value which is another type.

Operation
An operation is structurally identical to Attributes with the addition of parameters.

Static Property
A static property provides the same functionality as enums or constants. It has no parameters.

In Ecore, every attribute can have a specified type (like `EString`), and the `EClasses` can inherit a super-type. When mapping types which have its foundation in a Java class, we can take advantage of the JavaBeans metamodel for providing the functionality it contains. Instead of defining the JavaBeans metamodel within the workflow, it is possible to use it directly in our own metamodel. E.g. the EMF metamodel relies on the JavaBeans metamodel for handling types which is external to the Ecore model.

4.5 DPF Xpand Metamodel

As the problem description (4.3) suggests, using the DPF Ecore with the Xpand EMF metamodel will result in a tedious and unintuitive solution that does not convey the domain concepts of a DSML. As the predefined metamodels do, the DPF metamodel must implement the `MetaModel` interface. The DPF metamodel behaves like any other metamodel, and can be used in the same manner. Although not properly tested, the metamodel should work seamlessly with all of the predefined components in Xpand.

4.5.1 Considered Approaches

Before initiating the work on creating a Xpand metamodel, a different approach was considered due to the high learning curve of Xpand's internals. Since EMF is supported "out of the box", the first thought was to create a model-to-model transformation from DPF to EMF. This approach is valid, but as we discussed in section 3.7 it would restrict the functionality provided to what was defined in the EMF metamodel. A model-to-model transformation from DPF to EMF is probably work worth a master's thesis by itself, and was thus not considered to be a viable solution.

Another approach considered (see figure 4.2) was a simplified model-to-model transformation where an Ecore model was built programmatically from a DPF specification focusing on the basic constructs like nodes and arrows. To make this work, we would have to create an (dynamic) Ecore

model on the fly and generate a .ecore file which would provide editor support. To make things worse, dynamic EMF does not support creating custom operations like regular EMF does. The biggest problem with this approach however, is that the operations which would be provided through Xpand, would be the generic EMF API. This means we would be back to comparing `EClass` entities with their name to find a specific DSML node and thus provide no benefit at all.

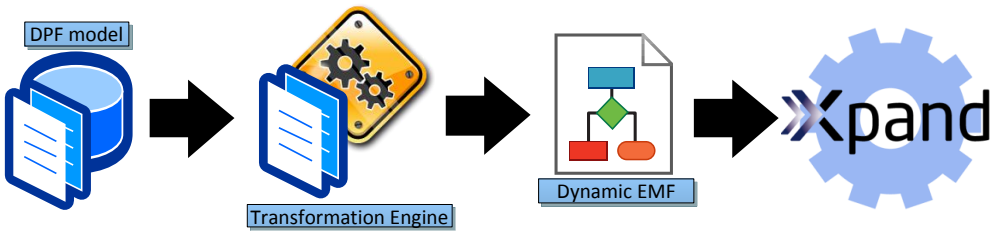

Figure 4.2: Figure shows the idea behind one of the considered solutions.

4.5.2 Packages

The basic structure of the `no.hib.dpf.codegen.xpand.metamodel` project contains four different packages.

no.hib.dpf.codegen.xpand.metamodel
Contains the metamodel class which perform and store all the mappings towards our custom types. It also contains an interface which defines the names of the entities in DPF (e.g. Node, Constraint, Arrow etc.).

no.hib.dpf.codegen.xpand.metamodel.typesystem
The type system package contains utility functionality which are used by the type classes and other parts of the system.

no.hib.dpf.codegen.xpand.metamodel.typesystem.types
This subpackage contains classes for all the custom types. This package is internal, and is not exposed to other plug-ins.

no.hib.dpf.codegen.xpand.metamodel.workflow
This package contains the workflow component `DpfReader` which handles the initialization of the metamodel, i.e. it loads the DSML's and the model's specification.

4.5.3 Structure of the Metamodel

A way of looking at the metamodel is a black box where you insert the DSML and its instance, and are then able to query it with objects or names which then returns a corresponding type.

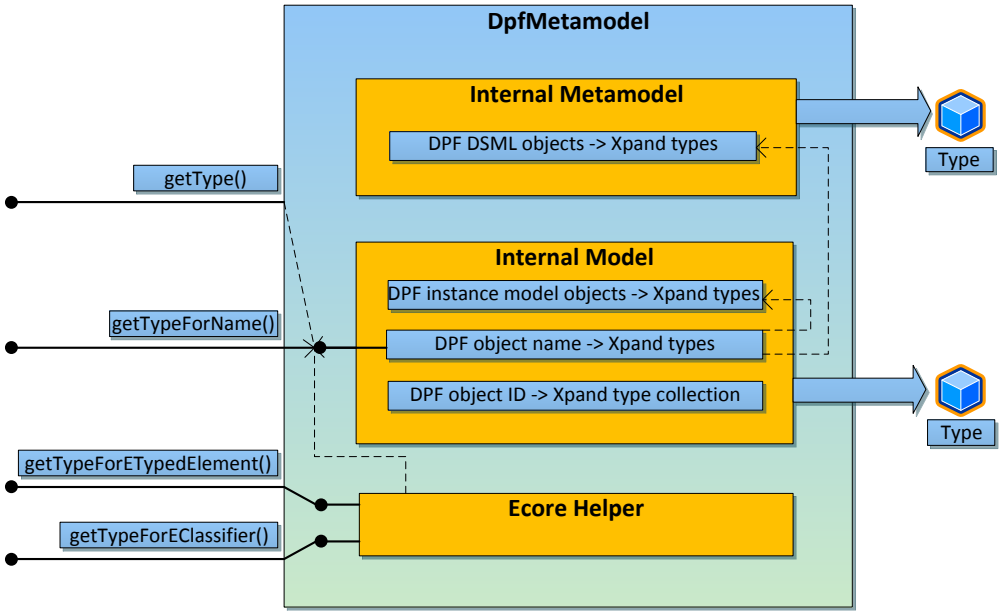

Figure 4.3: Figure shows the internals of a DPF metamodel.

The metamodel class has three internal classes with their own responsibility. Figure 4.3 shows a *internal metamodel*, an *internal model* and what is called the *Ecore helper*. The predefined metamodels in Xpand usually match an object to its metaobject and return that, but in DPF we have a lot of idiosyncrasies which makes it easier to have a representation of the model as well. An important observation is that every call to the metamodel will ultimately go through `getTypeForName`, and return a result from the internal metamodel *or* the internal model.

There is no validation of the models implemented in the metamodel. The DPF Editor enforce the consistency and validity of the DSML's typing through checking for graph homomorphisms [4]. Even though we reflect both the DSML and instance model in the metamodel, we do not check for graph homomorphisms between them. When a DSML or instance model is used within the metamodel it is assumed to be valid, both concerning constraints and typing.

Internal Metamodel

The internal metamodel represents the DPF DSML and its concepts. When the method `addDpfMetaModel(Specification)` is called, the `Specification` object representing the DSML is iterated over, and each DPF type gets matched through a map structure called `metaModelCache`. This particular mapping is mapped with the DPF model objects from the DSML as keys, and a corresponding type as value. If the type does not exist, a new one will be created with the DSML entity's name as name. In addition to the DSML types, we add two "dummy types", namely `Node` and `Arrow` for reasons explained in the next section. Each DSML entity gets stored with its namespace prefixed, although it is hardcoded for DPF models at this point (see 4.7).

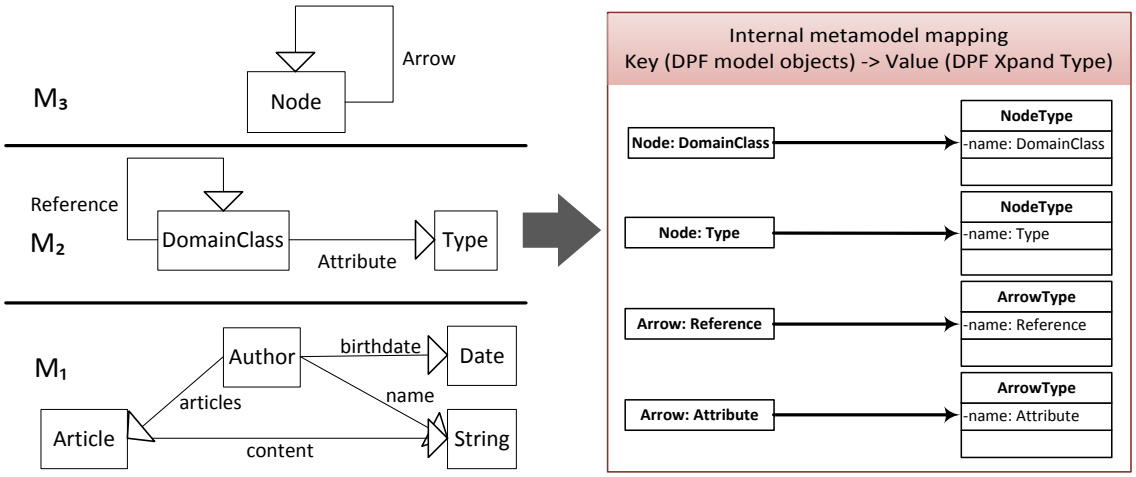

Figure 4.4: Figure shows a simple mapping between a DSML and Xpand types.

Figure 4.4 shows how a simple DSML is mapped to corresponding types within the type system.

Internal Model

An internal model represents the DPF instance model of the DSML. This class is initiated when `addDpfModel(Specification)` is called. As with the internal metamodel, we create a object to type mapping, but this time with the instance DPF specification as the basis. The reason for mapping the instance model is that we need to retrieve the correct object when traversing the graph. When chaining method calls, we would have returned the

DSMLs objects rather than the instance, thus looking up the wrong objects in the metamodel.

Next we create a DPF type ID[3] to list of DPF objects mapping. The list of objects is built using all the objects in the instance model which has the same metatype (ID). This is not necessary, but retrieving all instances of a DSML type is a common operation, and a collection simplifies that process.

The last and most important mapping is the type name to type mapping, which is used for resolving every query to the metamodel. This mapping points to both the internal metamodel mapping *and* the internal model's DPF object to type mapping. As figure 4.3 shows, the Ecore helper also uses the name to type mapping. Using names to identify objects is unfortunate due to its ambiguity, but necessary as all type definitions in the templates are simple names. An example issue is when nodes and arrows, or more than one node or arrow has the same name: which type is the correct one? The only way to deal with this is to check if the object retrieved from a mapping corresponds with what is queried. The type names are stored with their respective namespace prefix.

Ecore Helper

For convenience purposes we take advantage of the functionality specified in the DPF Ecore metamodel. From the types created for the metamodel this helper class gets called to resolve attributes and operations on the `EClass` in which any DSML/model type is an instance of. It is very useful to get operations like `get/setName` or `get/setGraph` for "free", i.e. without having to define them manually in the types. The names of getters are shortened to "name" instead of "getName" for convenience.

4.5.4 Type System

The types created for our metamodel reflects the types which DPF defines. The purpose of the types is to encapsulate the model object and expose its functionality through the Xpand type system. As stated in section 4.4.1 we can also define our own functionality through the Xpand API. This proves to be quite useful, as the DPF API is not tailored for graph traversal and code generation. As an example, we can take outgoing arrows from a node in the DPF Ecore metamodel where the only possibility is to retrieve *all* the outgoing arrows. When writing templates it is convenient to not iterate over *all* the arrows and decide its type within the template itself, as this would create a lot of complexity. A possible solution could be to create an extension that built collections of arrows upon generator execution, but this would be a lot of manual labour and unnecessary because it can be automated. This is where our custom types come to the rescue; we simply

[3] Each node, arrow, constraint and graph has a unique id within the `Specification` object.

define new methods that can be used within the template environment. This particular example problem is tackled by creating a mapping from a DSML type ID to a collection of instance model nodes that has the same metatype ID (see previous section).

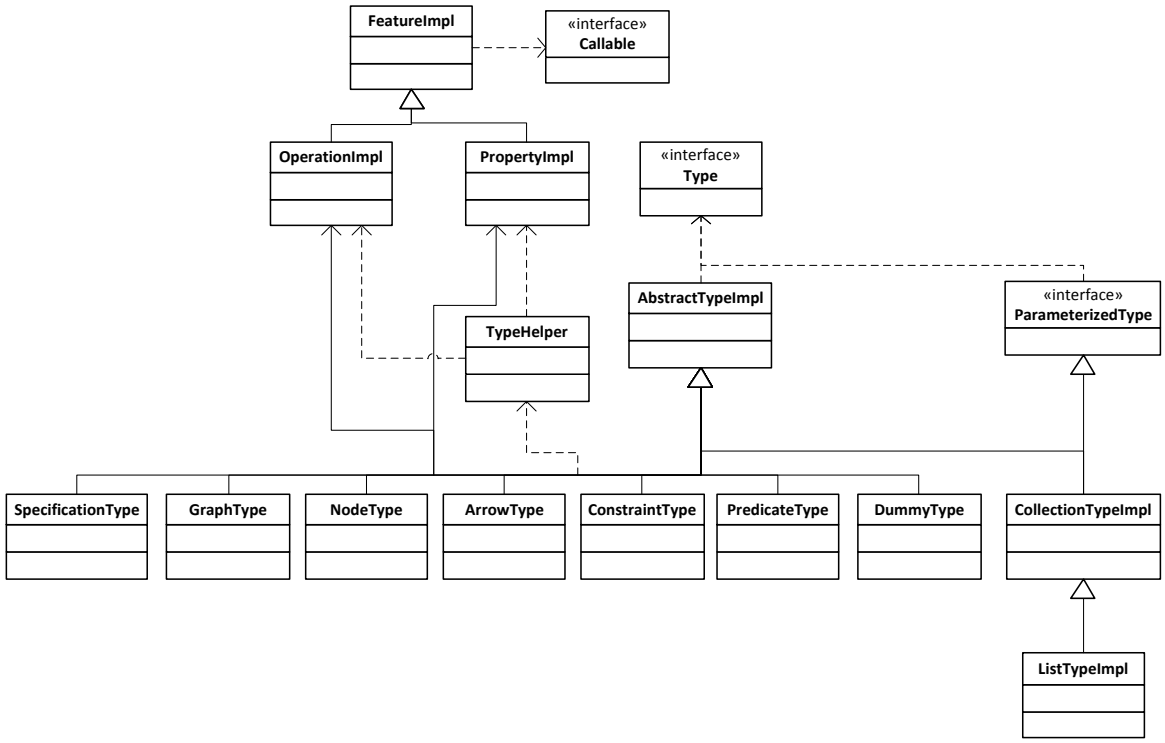

Figure 4.5: Figure shows the type system in the DPF Xpand metamodel.

Figure 4.5 shows an overview of the DPF metamodel's typesystem. For each DPF concept, we have a corresponding type. All of the types inherits `AbstractTypeImpl` which in its turn implements the `Type` interface. We also see that the `CollectionTypeImpl` implements the `ParameterizedType` interface which defines an inner type of the collection, which is how parameterized types are declared in Xpand. As in the `java.util` package, the list

and set is a specialized case of a collection.

We also see that we have referenced attributes, operations and static properties from our types. `AbstractTypeImpl` does caching of all the *features* within a type automatically. The `TypeHelper` class is responsible for creating all the Ecore features for each type based on the DPF type. I.e. an instance of `Arrow` will provide getters and setters for names, retrieving its graph, target and source etc.

Not all of the custom types provide any alterations to their corresponding DPF entity's type. `ConstraintType`, `PredicateType` and `SpecificationType` are only defined through the functionality in the DPF Ecore metamodel. Due to recent API changes there has not been a priority to define custom behaviour as the DPF Ecore functionality is satisfactory.

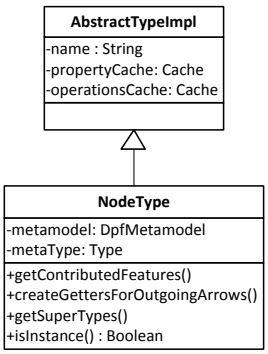

Figure 4.6: A class diagram showing a `NodeType` and its parent. This graphic only shows selected entities within the classes for simplicity.

Figure 4.6 represents a the "anatomy" of a `NodeType`. All of the specified types for the DPF metamodel inherits `AbstractTypeImpl`. We see `AbstractTypeImpl`'s data fields, which contains `Cache`[4] objects that store all the *features* of the metamodel. That is, when the method `getAllFeatures()` is called (which is defined through the Xpand interface `Type`), the class calls `getContributedFeatures()` which is an abstract method implemented in the subclass. This is an example of the *template pattern* [33].

The Xpand type system supports inheritance within the types. The Java objects which represent the Xpand types has no inheritance in between each other. This inheritance is defined through the method `getSuperTypes()`, where a type returns a set of Xpand types which the particular type should inherit. In the DPF metamodel each type from the model instance inherits its metatypes's type. Figure 4.5 shows a type called `DummyType`, which is a type that only contains its name, no operations and attributes are defined. The reason for implementing this type is to use it as supertypes for

[4] A `Cache` object is a wrapper around `java.util.HashMap` with some additional functionality.

the DSML's and instance model's types, for the purpose of defining general methods on e.g. nodes like `getOutgoingArrows`. Although that particular method is implemented through the DPF Ecore, it will not work due to the DSML and instance model types beeing different from eachother (e.g. a `NodeType` called `Process` is not the same type as a `NodeType` called `Control`).

The `createGettersForOutgoingArrows()` is a method which returns custom operations for the node. In short, the method creates a "get" method for each type of arrows which is outgoing from the node. This is the implementation of the scenario mentioned in the beginning of this section.

4.5.5 Reader and Workflow

For the metamodel to work in a MWE workflow, we need a reader component which initates the metamodel with data. In the DPF metamodel's case, we want the reader to load both a DSML and model. A component in Xpand is a class which implements the interface `WorkflowComponent`.

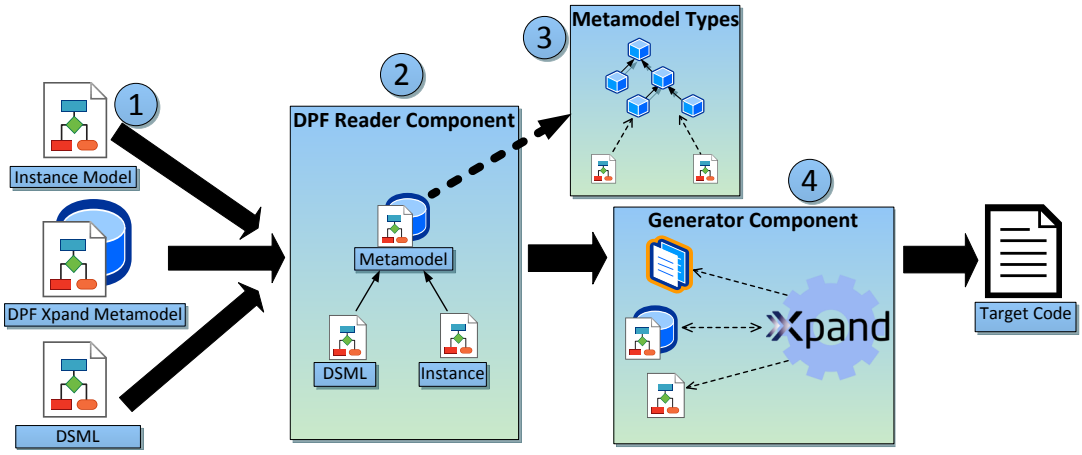

Figure 4.7: Figure shows how the Xpand DPF metamodel works in a workflow.

Figure 4.7 shows how the workflow proceeds from start to end. This visualization only contains two components, the DPF Reader and the Xpand Generator component.

1. The first step in a workflow is to declare properties. We assign a name to e.g. a path or value to simplify the appearance of the workflow file. This is optional as you can use paths directly in the components, the assigned ID is merely an alias. What is important is instantiating the metamodel and assigning it an ID, making it available across components within the workflow. See the three first lines of listing 4.2 for example.

2. The DPF Reader component takes two paths or URIs to the serialized versions of both a DPF DSML and its instance. The purpose of the reader is to initialize the metamodel. The component can be declared more than once with different metamodels and DSMLs.
3. This step illustrates the mapping from DPF types (from both the DSML and instance model) to Xpand types within the metamodel.
4. The generator component shows how the Xpand generator utilizes the metamodel by querying it for suitable types based on type definitions in the template, as well as the input model from the reader.

```
<property name="dpf_model" value="/path/to/resource/model.xmi" />
<property name="dpf_metamodel" value="/path/to/resource/metamodel.xmi"
     />
<bean id="mm_dpf" class="no.hib.dpf.codegen.xpand.metamodel.
    DpfMetamodel"/>

<component class="no.hib.dpf.codegen.xpand.metamodel.workflow.
    DpfReader">
  <dpfMetaModel value="${dpf_metamodel}"/>
  <dpfModel value="${dpf_model}" />

  <metaModel idRef="mm_dpf"/>
  <modelSlot value="dpf" />
</component>
```

Listing 4.2: A MWE workflow that depicts the DPF Reader component.

Listing 4.2 shows the DPF Reader component with properties declared for clarity. As previously mentioned, the reader component is needed for initializing the metamodel with the data from models. This happens through the `<dpfMetaModel>` and `<dpfModel>` tags. The `<metaModel>` specifies the DPF Xpand metamodel. Internally, we load the specified XMI files with the DPF Core API and call `addDpfMetaModel` and `addDpfModel` on the metamodel. The `<modelSlot>` tag stores the `Specification` object of the instance model for use in the `<expand>` statement in the generator component (see listing 3.9).

4.5.6 Dependencies

Using a framework like Xpand instead of a simpler solution, often entails some dependencies. The goal of creating a standalone code generation facility is somewhat hard to achieve. If we look past the dependencies of the Xpand framework, the `no.hib.dpf.codegen.xpand.metamodel` contains a very limited amount of dependencies:

no.hib.dpf.core The DPF core API.

org.eclipse.emf.mwe.core Used for the workflow specific code, i.e. the DPF Reader component.

org.eclipse.xtend Contains the type system specific functionality.

org.apache.log4j The logging facility used for debug statements within the metamodel and its types.

In a language workbench setting, the generators will always be used with the DSML. Using the Xpand framework for the code generation is then not a problem because the dependencies are fulfilled through the language workbench. A scenario could be that one would want to use Xpand without Eclipse; this is possible as Xpand can run standalone.

An interesting scenario could be where you did not want the whole language workbench tooling, but a lightweight textual solution based on e.g. Florian Mantz' textual DPF tool. In such a scenario it could be interesting to generate a code generator for a DSML that takes textual input (instance models) and outputs code. I.e. generate a standalone generator for a particular DSML. This is called a *generator generator* [29].

4.5.7 Testing the Metamodel

Creating tests for the code generation tool is important, as there is a lot of corner cases that is hard to predict when creating a metamodel. Unfortunately, test-driven development (TDD) was not an option because of time constraints. Another problem is the lack of expert domain knowledge; learning the ins and outs of DPF was not required to create a usable solution. With the adoption of the tool, identifying corner cases will be a lot easier. The most basic tests as loading models into the Xpand metamodel, and retrieving a type based on its name can be achieved using plain JUnit test-cases. When we want to test for a more intricate bug, creating tests programmatically is a lot of manual labour. A solution to this is creating a test fixture that can load models, and expect a particular output from the Xpand framework.

Even if we decide to create a test programmatically we need a model to operate on. Besides beeing time consuming work, this poses a difficult problem which needs to be resolved: migration from one model to another.

Every time we see a change in the DPF Ecore metamodel, all models created previously are unusable. What is needed is a migration strategy when these changes occur, so that the model based tests will not break on API change.

4.5.8 Documenting the Metamodel

As a lot of projects in the Eclipse ecosystem, the Xpand framework has limited documentation. There exists a user guide which covers most aspects of the tools, although some are more explained than others. Developer documentation is next to non-existing and the code base is almost without any comments. This makes the learning curve very steep when trying to make sense of how Xpand works. The best bet of getting an explanation for any concept or piece of code is through the Eclipse Community Forums [14]. The response on the *M2T forum* are excellent, with very helpful representatives from Itemis [27].

To mitigate the poor existing documentation, the project will be well documented. Hopefully, this project will serve as a foundation for further work, and must thus have as low threshold for learning the code as possible. Besides this report, the final code will be properly commented, with an example implemented (see chapter 5 to demonstrate the tool in use. The metamodel will especially be well commented, as the idea behind it might be hard to grasp when diving into the code base.

4.6 Integration with Eclipse

An important aspect of the language workbench is the integrated tooling. We want an IDE like experience when defining our generators, with the best possible support and tools for writing templates. Xpand provides rich editor support through the metamodel, but it is not ready for use until we have created a *metamodel contributor*.

4.6.1 Packages

no.hib.dpf.codegen.xpand.ui
: Root package which contains the plug-in activator and `DpfMetaModelContributor` (see next section).

no.hib.dpf.codegen.xpand.ui.nature
: Contains support for a project nature. A nature is a project-type specific environment, which can be used to load project type specific functionality like a specific editor or a property dialog. There is also support for project specific properties (workspace scope) through a generic get/set property class.

no.hib.dpf.codegen.xpand.ui.wizards

The wizards package provides UI classes for the project creation wizard. It also contains a XML parser which parses the workflow XML files and gives the option to alter attributes.

4.6.2 Editor Support

Editor support for metamodels are not quite supported out of the box. The `MetamodelContributor` interface needs to be implemented and then registered through an *extension point*[5] within Xpand. The metamodel contributor has one objective; return the DSMLs which should be associated with the editor. The editor support in Xpand provides syntax checking and a dynamic code assistance which provides auto completion and suggestions when writing templates.

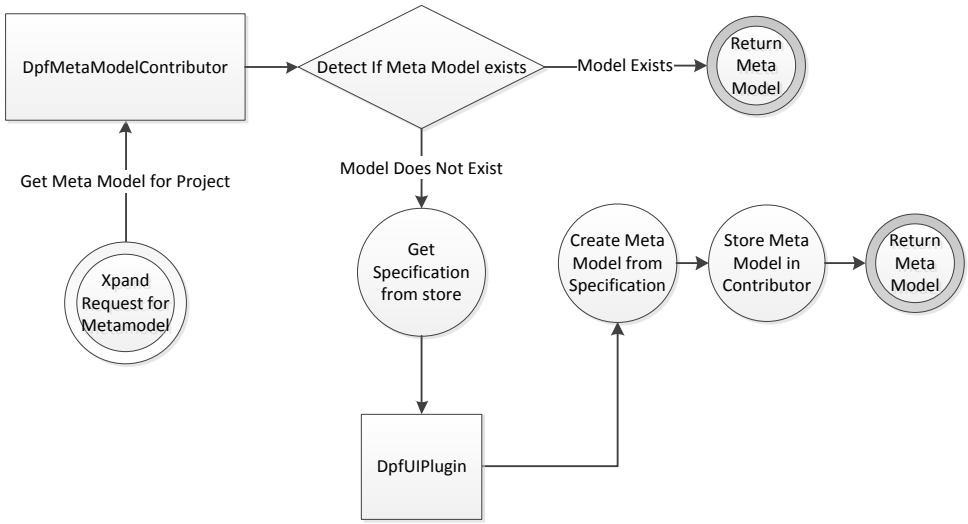

Figure 4.8: Figure shows the process of retrieving a metamodel for a project.

Figure 4.8 describes the process of retrieving a metamodel. Xpand will request a metamodel for a project, in which the metamodel contributor will try to look up if it has a metamodel associated with the requested project or not. If not, the metamodel contributor will request that the `DpfResourceDeltaVisitor` scans the workspace to retrieve any specifications found in either a .prefs file, or a workflow file. The specification found will then be loaded using EMF, and returned to the metamodel contributor where a metamodel object is created with the returned specification as argument. The metamodel contributor then stores the metamodel with the

[5] An extension point is provided by a plug-in for enabling other plug-ins to contribute functionality.

requested project associated for later use and returns the metamodel.

The Eclipse integration and the Xpand generator component both rely heavily on the DPF metamodel. When creating a new DPF Generator Project we specify the location of our DSML through the wizard or the workflow, which then loads the DSML into a metamodel that is associated with the current project. This metamodel is never used for generating code, but only for providing the DSML concepts and its types to the template editor. When generating code, the metamodel and types are instantiated in the workflow, and only exists until the workflow is finished. The template editor's metamodel never has an instance model associated with it, as it is irrelevant.

4.6.3 Project Structure

For the users convenience, we generate a project structure which is ready for use. Listing 4.3 shows the project structure which is generated for the user. The `src` and `src-gen` are source code folders where `src-gen` will contain any generated code. This path is not hardcoded, and can be customized through the workflow file (see listing 3.9). The project is built on top of a Xpand project. This means the project is an Eclipse plug-in which can be used in the same manner as any other Eclipse plug-in project, and we get all the correct dependencies for a Xpand project through the `MANIFEST.MF` file. The `.project` file contains details about the project, such as which natures (see next subsection) are activated. This file is essential to let Eclipse know what kind of project we are dealing with. Lastly the `.settings` folder contains two preference files which are read when the project is recognized as both an Xpand project, and a DPF Generator project. The files define which metamodel is active on the project and where to find the DSML.

```
no.hib.dpf.test/
|-- .settings/
|  |-- no.hib.dpf.codegen.xpand.ui.prefs
|  |-- org.eclipse.xtend.shared.ui.prefs
|-- META-INF/
|  |-- MANIFEST.MF
|-- src/
|  |-- template/
|  |   |-- templ.xpt
|  |-- workflow/
|      |-- workflow.mwe
|-- src-gen/
|-- .project
```

Listing 4.3: Listing shows the generated project structure for a DPF Generator Project

A template and workflow file are also added. The workflow file contains a simple standard setup, with the DPF Reader component properly defined.

The template file contains the "entry" definition of a template, more precisely a `DEFINE` block for a DPF specification.

4.6.4 Project Nature

In Eclipse the concept of a project nature is to indicate that your project are of a certain type, which uses certain tools. The nature can configure the UI by contributing menu selections, views or any other Eclipse artefact. Natures can also handle project specific properties.

In the earlier iterations of the codebase, all paths to models was stored in project specific settings. Unfortunately these settings was also workspace specific, which means that if a user checked his project into a VCS and another checked it out, the second user would not have access to the first users settings as to where the models was stored. This problem is avoided by writing a `.prefs` file in the `.settings` folder in the project.

This means that the project nature facility in the plug-in is now largely unused, besides always being enabled for new DPF Generator projects. The reason for leaving the code in the plug-in is that it will almost certainly be used in the future if the tool gets further developed.

4.7 Shortcomings in the Tool

Namespaces and Multiple DSMLs

In its current state namespaces are not supported. The prefix `dpf` is hard-coded and used for all types within the metamodel. This poses a problem when one wants to have more than one DSML in the project. It is also not possible to have multiple instance models on one DSML. A solution to this is to run the generator workflow for each DSML and corresponding instance.

This feature was a low priority in this project, and was unfortunately not finished. It might be a usability improvement to manage multiple DSMLs in a single workflow, as well as multiple DSML instances. Multiple DSML instances on a single language requires the implementation of namespaces, as name collisions can occur, and thus returning the wrong node or arrow.

An implementation of this feature is relatively easy. One suggestion is to prefix every entity from the DSML/model with a custom name. Another solution could be to store a map of `InternalMetamodel`'s and `InternalModel`'s with the prefix as key.

Decoupling from `no.hib.dpf.core`

At this point, there is a very tight coupling between the tool and `no.hib.dpf.core`. Although a natural dependency, it is problematic that whenever the DPF API

is changed, the metamodel might break and its types needs to be updated manually. The nature of the DPF project, where each student works on a separate aspect of the tool, makes it desirable to have each "module" as self sustainable as possible. Learning a new codebase from scratch is a lot of work which can be alleviated by good self-documenting and annotated code, but it is still extra work.

A solution to this problem can be to create an API, beyond the one generated from the DPF Ecore, which is maintained by whoever is responsible for `no.hib.dpf.core`. Tolvanen and Kelly [29] mentions data-based and message-based APIs to decouple generators from models. A data-based API is basically what we have today, an API which returns parts of the actual model, or a copy. To avoid breaking the API one should mark obsolete methods and classes as `@Deprecated` rather than removing them. The message-based API is an approach where only proxies for the model are sent back and forth. This means you operate on proxy objects that resolves to a primitive like string or integer at its most basic form.

Test Coverage

The current test coverage of the tool is poor. The metamodel and type classes are the only components which have some tests written. The coverage for these components are very limited as well. Besides the lack of time, the issue addressed in section 4.5.7 is the main reason for not creating more tests for the metamodel itself. Other components like the workflow reader and UI related code can be tested programmatically.

Implementing Constraints and Predicates Using Types

When this tool was developed, the signature file (see section 2.6) was hard-coded along with its predicates. It was not a priority to do anything about these types, as the Ecore helper within our metamodel exposed the needed functionality. The latest version of DPF Editor gives the opportunity to define custom predicates which opens up new possibilities. There was unfortunately no time to implement this.

Chapter 5 will present a way to simplify the handling of constraints using extensions.

4.8 Feature Overview

We wrap up with an overview of what has been achieved through the development in this project:

Xpand metamodel
: The Xpand metamodel for DPF is the core functionality that lies at the

heart of what has been developed in this project. The metamodel is a mapping from the DPF model types to our custom Xpand types. The types exists only in the metamodel, and is available through queries.

Type system
We have defined a type system which Xpand can understand for each of the modelling constructs in DPF. This enables us to easily define new functionality for each modelling constructs, such as DSML specific getters and setters, or utilizing the functionality already defined in the DPF Ecore metamodel.

Workflow integration
With the implementation of the DPF Reader component, we have integrated the metamodel with the Modeling Workflow Engine (MWE). This results in a seamless use of the metamodel between the different components that Xpand offers.

Eclipse integration
With the implementation of a *metamodel contributor*, we can take advantage of all the features which Xpand has to offer. We get editor support for template editing with Xpand, extension editing with Xtend and constraints checking with Check. The editor support contains code completion, syntax coloring, error highlighting, source navigation and refactoring.

Project environment
As part of the Eclipse integration, we have implemented a project environment that defines a wizard for generating a project structure suitable for code generation. There is also a facilitated a *project nature*.

The time invested in creating this functionality is time well spent, as using any of the alternative approaches (section 4.5.1) would have been an inferior solution technologically.

Lastly, let us take a look at our problem description in section 4.3. We stated that an ideal solution would solve a few requirements:

Clear expression of domain concepts
Through the interpreted nature of Xpand, we are able to create an environment for code generation based on the concepts of a DSML rather than an instance model.

Integration with Eclipse
The metamodel and the metamodel contributor give us access to all the features Xpand has to offer.

Standalone generator
This requirement is the one which is not completely fulfilled. A framework with the size of Xpand is bound to have some dependencies. The bright side is that a generator can be executed without Eclipse, with only Xpand, log4j and `no.hib.dpf.core` as dependencies.

Chapter 5

Demonstrating the Tool

This chapter will demonstrate how to use the Xpand framework together with the DPF metamodel and its Eclipse integration. We will generate code for the Play [45] web framework based on a simple DSML. Generating code for a web framework is an excellent use case, because of the similar traits from one application to another.

5.1 Components/Packages

`no.hib.dpf.codegen.examples.dpfplay`
: Contains the generator implementation.

`no.hib.dpf.codegen.examples.dpfplay.ui`
: This plug-in contains the Eclipse integration for the generator.

The component name chosen is `codegen` like the DPF Xpand metamodel. The sub component is called examples, as a common subcomponent for all code generation related example projects. The name chosen for the project is "dpfplay".

> *NOTE: The plug-ins are part of the reference example project. This chapter will not explain the Eclipse integration found in the `no.hib.dpf.codegen.examples.dpfplay.ui` plug-in.*

5.2 Choosing a Framework

The following is a short evaluation of the candidates chosen for this tool demonstration. The criterias used was simple; a modern framework with as

little configuration and boilerplate[1] code as possible. Most popular frameworks today is based on the *Model View Controller (MVC)* pattern, which give them similar properties. If one defines two web applications in the same framework, they will most likely share the same structure. This makes web applications a popular domain for MDE and code generation [53] [30].

The model part of an MVC framework is usually the most generic and simple to generate code for. *Controllers* and *View* vary a lot more in how they function.

After a short evaluation the Play Framework was chosen. In addition to fitting the criterias, it is made for Java (and Scala) which was ideal considering the time left in the project. As we will discover, Play enables us to get results with very little code.

5.2.1 Lift

Lift [32] is a web framework based on the Scala programming language, which focuses on security, scalability and ease of use. The framework tries to fill the same need as other web frameworks such as Ruby-on-Rails and Spring, but tries to improve on the shortcomings other frameworks exhibits. Lift follows the MVC pattern to enforce separation of concern; unlike JSP, Lift does not allow code in the templates. The framework also benefits from Scala's Actor model which provides concurrency in a safe and robust manner.

5.2.2 Django

Django [12] is another web framework based on the MVC paradigm which uses the Python programming language. Its focus is to hide the boilerplate code and focus on the functionality. Among the wide range of features you find Object-Relational Mapping (ORM), caching framework, template engine, a standalone web-server and automatically generated CRUD interface for your model classes.

5.2.3 Grails

Grails [23] uses the Groovy programming language which is a dynamic language running on the JVM. As with Django and Lift, Grails uses the MVC pattern. Under the hood it uses a lot of the functionality inside Spring, but stays clear of any XML configuration.

[1] Boilerplate code is a term for code which appears in many places with little to none alterations.

5.2.4 Play Framework

The Play Framework [45] is a MVC web framework for Java and Scala. It aims to be a more effective alternative to other Java frameworks like Spring and Java EE. Play comes with its own runtime that loads code changes directly into the JVM, and removes the need to restart the application server each time a change is made. Another notable feature is the template engine, which uses Groovy as expression language. The result is a concise syntax with less boilerplate code.

5.3 Problem Description

Creating a working example for the code generation tool is needed to demonstrate how it works. The process of creating a generator project is simple, yet it demands some hand coding to create the templates, which are the most important part. Any UI integration of the generator will need to be hand coded in the same manner as any other Eclipse plug-in.

Through this chapter we will demonstrate a simple code generator, which generates simple Java model classes for the Play framework using a simple DSML. We will utilize Play's built-in modules to provide a CRUD (Create, Read, Update, Delete) interface using our generated classes.

This chapter acts as a tutorial on how to use the code generation tool. The code for this example project can be found at `http://dpf.hib.no/downloads/`. It contains a more elaborate example with Eclipse integration for the generator.

This example project was developed using Eclipse Indigo (modelling tools bundle) with Xpand and the DPF Xpand metamodel installed. The version of Play framework used is 1.2.4.

5.4 Creating the Generator

5.4.1 What to Generate

As stated in the previous section, we want to generate simple model classes for a Play project. The example is very simple, and will not entail the generation of any behaviour oriented code. We will handle multiplicity constraints to give an example on how to use them in their current state. In section 3.3 we briefly discuss the creation of a code generator and what is needed for creating one; mainly a sample input and output to clearly define the generator's requirements. The templates in the generator can be regarded as transformation rules, and it is thus necessary to see what you want to transform and what it should result in.

The first thing we need to do is define a DSML for our problem. This step is not necessarily a part of the code generation activity, as the language might be pre-defined.

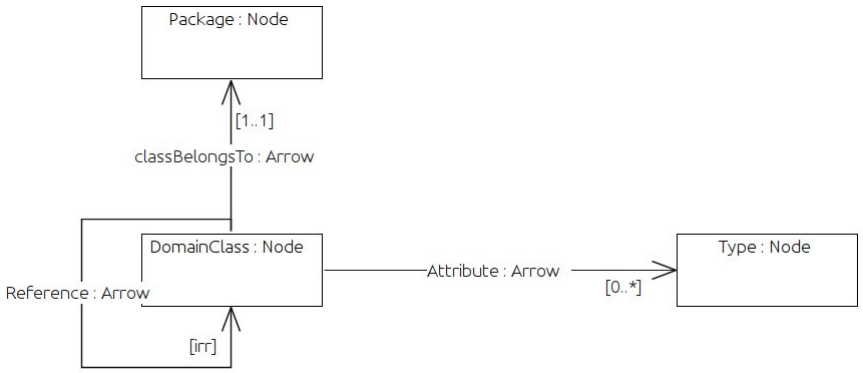

Figure 5.1: Figure depicts a simple DSML for creating domain classes.

Lets go through the DSML step by step:

1. We define three `Nodes` which have the type names `DomainClass`, `Type` and `Package`.
2. A `DomainClass` may have a *reference* to one or more `DomainClasses`.
3. A `DomainClass` can not reference itself. This is enforced by the irreflexive `[irr]` constraint.
4. A `DomainClass` has exactly one `Package` (`[1..1]` multiplicity constraint).
5. A `DomainClass` can have zero or more `Types` associated.

In short, this DSML enables us to create `DomainClass` nodes that belongs to a `Package`. The `DomainClass` may or may not have any attributes or other `DomainClasses` associated with it.

To properly understand how to create our generator, we need to create a sample input model:

The instance model in figure 5.2 shows an example with nodes `Author` and `Book` which both are typed by `DomainClass`. Each node has a few attributes shown by the arrows typed by `Attribute`. There are also a zero-to-many relation between `Author` and `Book` (an author can have zero or more books), as well as a one-to-many relation from `Book` to `Author` (one book can have many authors).

Now that we have a sample input defined, we can define what we want to achieve with the generator. This example will create very simple model classes which forms the foundation in a Play web application.

Listing 5.4.4 shows a draft of how we would like the code to look after code generation. The seasoned reader might see that we have defined our data fields as *public*, rather than *private*. This is an example of the convenience Play provides through its own runtime; all public fields will have get-

Author.java

```
package no.hib.dpf.codegen.examples.dpfplay.model;

import java.util.Date;
import java.util.ArrayList;

public class Author {
  public String name;
  public String email;
  public Date birthdate;
  public ArrayList<Book> books;

  public Author(String name, String email, Date birthdate, ArrayList<
      Book> books) {
    this.name = name;
    this.email = email;
    this.birthdate = birthdate;
    this.books = books;
  }
}
```

Book.java

```
package no.hib.dpf.codegen.examples.dpfplay.model;

import java.util.ArrayList;

public class Book {
  public String name;
  public String isbn;
  public ArrayList<Author> authors;

  public Book(String name, String email, ArrayList<Author> authors) {
    this.name = name;
    this.isbn = isbn;
    this.authors = authors;
  }
}
```

Listing 5.1: Listing shows a preliminary draft of the code we want to generate.

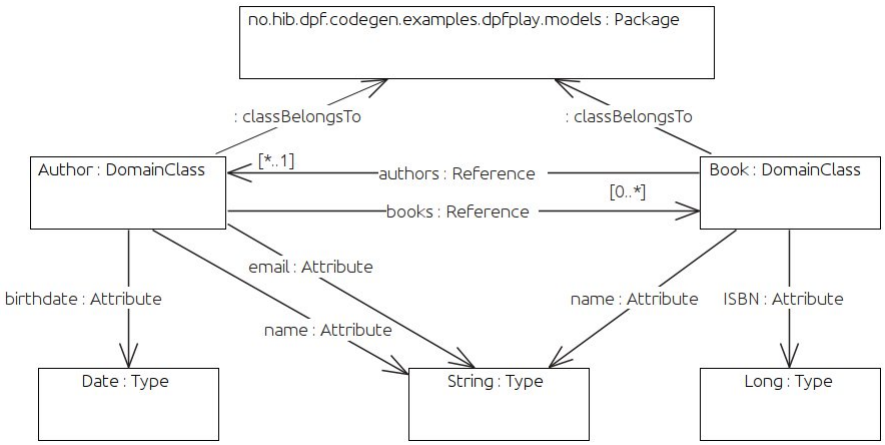

Figure 5.2: An instance model for the DSML (5.1).

ters and setters generated. Model classes in Play will typically also contain Java Persistence API (JPA) annotations for the persistence of the models. These will be added at a later point.

Now that we have defined our DSML, instance model and output sample, we can start thinking about creating a generator project and define our templates.

5.4.2 Creating a Generator Project

When selecting the *New* wizard in Eclipse, we find a new entry under the DPF category, *DPF Generator Project*. This is part of the functionality we defined in the `no.hib.dpf.codegen.xpand.ui` plug-in.

- *Select DPF Generator Project as figure 5.3 shows.*

Figure 5.4 shows the DPF Generator Wizard. As shown, there are two fields in the wizard. The project name is obligatory while the location of the DSML is optional. As discussed in section 4.6.2, the DSML is needed to enable the editor support. Even though a DSML is not defined for the project you create, one can define the location where it is supposed to be. The DSML will then load upon creation. One can also use the location of DSMLs in other projects in the workspace. We will leave the metamodel location blank for now, as we wish specify it later.

- *Set project name to `no.hib.dpf.codegen.examples.dpfplay`.*
- *Leave metamodel location blank and press "Finish".*

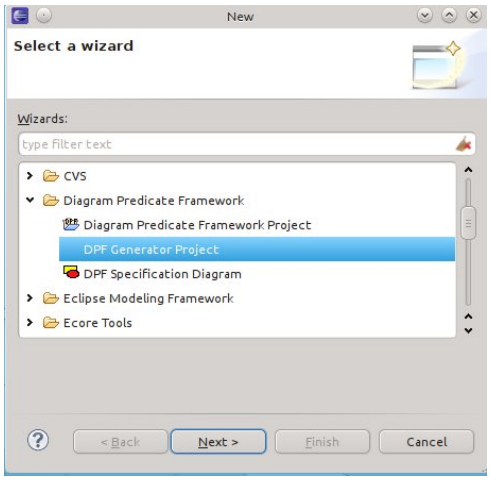

Figure 5.3: Eclipse *new* wizard showing the DPF category with DPF Generator Project.

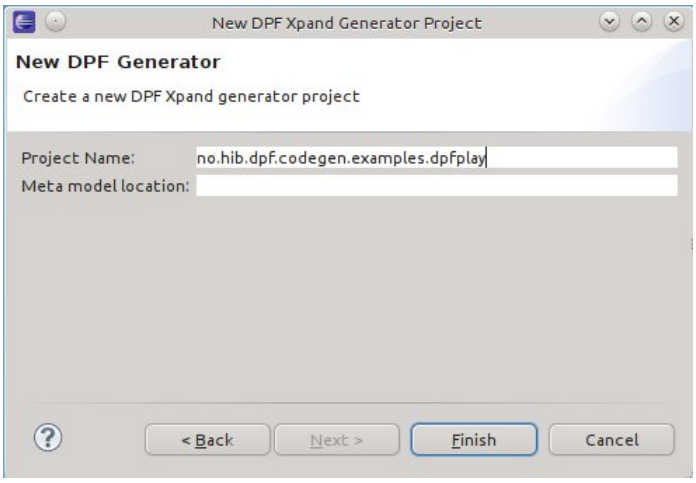

Figure 5.4: Figure shows the DPF Generator Wizard.

We have now a ready to use project structure (see 4.6.3). As the DSML location has not been defined, the generated template stub will show an error as the `dpf` namespace is not found.

Before proceeding, we want to define the DSML and instance model using the DPF Editor, so that we can get editor support in the later steps.

- *Create a new folder **models** in our project.*
- *Inside the **models** folder, create a new DPF specification with the name "metamodel".*
- *Open the specification and define the DSML in figure 5.1.*
- *Create another DPF specification with the name "author" within the **model** folder. Use "metamodel.dpf.xmi" as the typing for the specifi-*

cation.

- *Open the specifcation and define it using our model from figure 5.2.*

5.4.3 Defining the Workflow

With the project structure a workflow is generated, it contains almost everything that is needed to run it. The generated workflow should look like so:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<workflow>
  <!-- workflow properties -->
  <property name="dpf_model" value=""/>
  <property name="dpf_metamodel" value=""/>

  <property name="src-gen" value="src-gen"/>

  <!-- set up EMF, only needed when using URI's -->
  <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup">
      <platformUri value=".."/>
  </bean>

  <!-- instantiate metamodel-->
  <bean class="no.hib.dpf.codegen.xpand.metamodel.DpfMetamodel" id="
      mm_dpf"/>

  <!-- DPF component -->
  <component class="no.hib.dpf.codegen.xpand.metamodel.workflow.
      DpfReader">
      <dpfMetaModel value="${dpf_metamodel}"/>
      <dpfModel value="${dpf_model}"/>
      <metaModel idRef="mm_dpf"/>
      <modelSlot value="dpf"/>
  </component>

  <!--  generate code -->
  <component class="org.eclipse.xpand2.Generator">
      <metaModel idRef="mm_dpf"/>
      <expand value="template::templ::main FOR dpf"/>
      <outlet path="${src-gen}">
          <postprocessor class="org.eclipse.xpand2.output.
              JavaBeautifier"/>
      </outlet>
  </component>
</workflow>
```

Listing 5.2: The generated workflow file in a DPF Generator Project

As listing 5.4.4 shows, we get a workflow file with the DPF Reader component inserted (see 3.7 for an explanation of a workflow). The only attributes which must be set is the location for a DSML and an instance model.

- *Insert the DSML's path* `models/metamodel.dpf.xmi` *into the* `dpf_metamodel` *property's value attribute.*
- *Insert the instance model's path* `models/author.dpf.xmi` *into the* `dpf_model` *property's value attribute.*
- *Save the file.*

Defining the instance model at this moment, does not affect anything as it is only used when running the generator. The definition of the DSML on the other hand is more interesting; every time a resource gets changed in the project, we scan it and try to retrieve a valid path to a DSML. The first check is to see if a workflow file is defined. If so, it will look for the `dpf_metamodel` attribute and see if a value is defined. If a value is defined, it will mirror the path in the *no.hib.dpf.codegen.xpand.ui.prefs* file and load the specifcation for use in the DPF Xpand metamodel. If no such value is found in the attribute, the tool will try to find a path in the *no.hib.dpf.codegen.xpand.ui.prefs* file. If this fails as well, no metamodel will be loaded for the project. This solution is admittely not the most optimal, as it will not work with more than one DSML per project (in its current state).

5.4.4 Defining the Template

Now that we have defined a DSML in the workflow file, we get editor support. We have defined our input model as well, and created a draft of our output. We will now define a template for creating model classes and use Xtend and Java extensions where it is needed.

> *NOTE: When creating/reading templates in Xpand, make sure that the font encoding is set to UTF-8 or another encoding which support* guillemets *(« and »).*

Basic Xpand Concepts

- *Open the templ.xpt file.*

The only content of the file is:

```
«IMPORT dpf»

«DEFINE main FOR dpf::Specification»

«ENDDEFINE»
```

A `DEFINE` block for a `Specification` type is required to start with, as it is the assumed input object for the generator. `DEFINE` statements are also called *definitions* or *templates*. The `DEFINE` statements are the building blocks of the template files; each statement can invoke other `DEFINE` statements and so on.

In our example template, the `main` block engulfs all of the other `DEFINE` blocks. The `main` block is invoked through the workflow engine using `<expand value="template::templ::main FOR dpf"/>`. `dpf` is the *model slot* defined in the DPF Reader (see 4.5.5). The `Specification` object that represents a DPF model has a hierarchy which is reflected in the templates we create.

The next step in our template is to create a `DEFINE` statement for a graph.

> *NOTE: Guillemets are created using* Ctrl+< *for « and* Ctrl+Shift+< *for ».*

- *Add the following listing to templ.xpt:*

```
«DEFINE graph FOR dpf::Graph»

«ENDDEFINE»
```

To invoke the `graph` block from `main`, we need an `EXPAND` statement. The `EXPAND` statement is used for invoking other `DEFINE` blocks.

- *Inside the `main` block, add:*

```
«EXPAND graph FOR this.graph»
```

The `EXPAND` statement expands the `graph` block with the `Specification` object's internal `graph` object. The `this` handle refers to the object which the `DEFINE` block pertains to.

Our DSML defines the concept `DomainClass` which represents a model class for use in a Play web application. The next step in our template is to iterate over every `DomainClass` which is defined in the instance model.

- *Create a new `DEFINE` statement:*

```
«DEFINE domainclasses FOR dpf::DomainClass»

«ENDDEFINE»
```

- *Add the following line to the `graph` `DEFINE` block:*

```
«EXPAND domainclasses FOREACH this.getDomainClasses()»
```

We start to see how the DPF Xpand metamodel lets us express the concepts of our DSML rather than the DPF Ecore model (see section 4.3 for an example). The `getDomainClasses()` method is a custom convenience method defined in the DPF Xpand metamodel's `GraphType`. Using this method we iterate over every `DomainClass` instead of its type node, `Node`. An important observation is that we use `FOREACH` when we want to invoke a `DEFINE` statement for each element in a collection.

In Java, a class is defined in its own file. This means we need to create a Java class file for each `DomainClass` type in our instance model. This is achieved using the `FILE` statement.

- *Inside the `domainclasses` block, add:*

```
«FILE this.name.toFirstUpper() + ".java"»
«ENDFILE»
```

The `FILE` statement creates a file with a name as argument. The example shows that we retrieve the `name` of the current `DomainClass` using `this`. We also take advantage of the built-in string operation `toFirstUpper()` that returns the name with the first letter as uppercase.

- *Run the MWE workflow by right-clicking the workflow file and selecting "Run As -> MWE Workflow".*

We should now see two empty files called "Author.java" and "Book.java" in our `src-gen` folder.

The basic constructs of Xpand are now familiar and we can start defining the content of the files we generate. From within the `FILE` block we can insert the first statements.

Creating an Extension

- *Inside the `FILE` block, add:*

```
package «this.getAClassBelongsTos().get(0).target.name»;
```

- *Run the workflow.*

When executing the workflow, we generate two files which only contains a package declaration in the `src-gen` folder. Although the statement looks a bit messy, the semantics are easy to grasp. From the current `DomainClass` type, we retrieve all the outgoing `classBelongsTo` arrows. The DPF Xpand metamodel defines every getter for an outgoing arrow as a collection for consistency reasons throughout the tool, despite the use of a `[1..1]` multiplicity constraint on the particular arrow. When we have retrieved the arrow, we call the target attribute, which retrieves the node that the arrow

points to, and lastly we fetch the target node's name. We observe that Eclipse displays an error icon on our generated files, and tells us that the files are in the wrong package. This happens because we have told the generator to output everything to the `src-gen` directory. In our metamodel we have specified that each model class can have its own package which means we do not want to hardcode the package path. Fortunately we can delimit the `FILE` statement's name argument with a slash, thus defining the output file's path.

Creating a valid path for the `FILE` statement is an example on where you can use extensions, rather than perform program logic within the template. Replacing all "." with a "/" is done using a single line of code, but it will look tidier with a descriptive method name.

- *Create a new package called `extensions`.*
- *Create a new file in the `extensions` package called* ***dpfplay.ext****.*
- *Insert the following code:*

```
import dpf;
getPackage(DomainClass d):
    d.getAClassBelongsTos().get(0).target.name.replaceAll("\\.",
        "/") + "/";
```

- *Insert the following right below the `import dpf` statement in our template:*

```
«EXTENSION extensions::dpfplay»
```

- *Modify the `FILE` statement inside the `domainclasses` block:*

```
«FILE packageName(this)+this.name.toFirstUpper() + ".java"»
```

- *Run the workflow again.*

The generator should now output the files into its proper package `no.hib.dpf.codegen.examples.dpfplay.models`.

The `Type` nodes in our model can be any type as we have not intended any restrictions on which types that are allowed. We must then take care of this inside our templates (or extensions).

- *Create a new `DEFINITION` statement called imports:*

```
«DEFINE imports FOR List[dpf::Attribute]»
    «FOREACH this AS e»
        «IF e.target.name == "Date"»
            import java.util.Date;
        «ENDIF»
    «ENDFOREACH»
«ENDDEFINE»
```

- *Add an `EXPAND` statement below the package definition in `domainclasses`:*

```
«EXPAND imports FOR this.getAAttributes()»
```

We now have an import for the `Date` type if it is defined in the template. In a more advanced scenario, it would perhaps be easier to use a fully qualified name for the import to avoid long chains of `IF` statements for definition of every possible type. Clearly defining which types are allowed would also be a way to fend off unnecessary complexity.

Creating a Java Extension

Creating collections of `Reference` types is possible, and this means we need to handle this in a proper manner. We need to figure out if the `Reference` is constrained with a multiplicity constraint, and what the bounds are.

In the DPF Ecore metamodel, constraints are defined on the *graph* and not on the nodes and arrows it constrains[2]. Each constraint contains a list of the nodes and arrows it constrains, and a reference to the predicate which it is an instance of. To identify what kind of constraint we are dealing with, we need to look at its predicate. When writing templates, these are the idiosyncrasies which can be dealt with through defining operations in our DPF Xpand metamodel's type system. Unfortunately there is no solution for this in place for various reasons (see section 4.5.4). Creating an extension to simplify the process is a satisfactory solution for now.

In Xpand we can define Java extensions which are an Xtend function that calls methods in external Java classes.

- *Create a new Java file in the `extensions` package called **`TemplateHelper.java`**.*
- *Insert the following content:*

```
package extensions;

import no.hib.dpf.core.Node;

public class TemplateHelper {
    public static String printArrayImport(Node n) {
        return "import java.util.ArrayList;";
    }
}
```

[2] In the latest version of the DPF core, constraints can be retrieved on the nodes and arrows it constrains. The code generation tool is not (yet) compatible with the new API.

- *Insert into* ***dpfplay.ext****:*

```
String publicArrayImport(DomainClass d):
    JAVA extensions.TemplateHelper.printArrayImport(no.hib.dpf.
        core.Node);
```

- *Insert a call to this extension in your template below the* `imports EXPAND` *statement.*

For now we hardcode the return value from the Java extension.

Type inference in Xtend[3] does not apply to Java extensions which is why we have specified String as the return value. One must also use the fully qualified name of the class and method to invoke it.

Finishing the Template

Up until now we have been thorough with the explanations of the Xpand language and its concepts. We are now equipped with enough knowledge to speed things up.

- *Below the* `printArrayImport` *insert:*

```
public class «this.name.toFirstUpper()» {
    «getDomainClassAttrRef(this.getAAttributes(), this.
        getAReferences())»

    public «this.name»(«paramList(this.getAAttributes(), this.
        getAReferences())») {
        «constructorSetAttributes(this.getAAttributes(), this.
            getAReferences())»
    }
}
```

- *The extensions above are found listed below.*

All of the defined extensions needs to know if we have a multiplicity constraint so that it can write `ArrayList<Book>` instead of `Book`. The extensions shown in listing 5.5.1 shows both Xtend extensions as well as Java extensions. The listing below demonstrates the vast amount of code needed to handle simple constraints using the DPF API directly.

[3] Xtend is the Xpand extension language, see section 3.7.

```
package extensions;
import java.util.List;
import no.hib.dpf.core.Arrow;
import no.hib.dpf.core.Constraint;
import no.hib.dpf.core.Node;

public class TemplateHelper {
    public enum Mult {MANY_TO_ONE, ONE_TO_MANY, MANY_TO_MANY,
        ONE_TO_ONE};
    private static String MULT_CONSTRAINT = "[mult(m,n)]";
    public static Mult parseConstraint(String expr) {
        String f,l;
        Mult m;
        try {
            f = expr.substring(0, expr.indexOf(','));
            l = expr.substring(expr.indexOf(',') + 1, expr.length());

            if(Integer.parseInt(f) == 1 && Integer.parseInt(l) == -1)
                {
                m = Mult.ONE_TO_MANY;
            } else if(Integer.parseInt(f) == -1 && Integer.parseInt(l)
                 == -1) {
                m = Mult.MANY_TO_MANY;
            } else if(Integer.parseInt(f) == -1 && Integer.parseInt(l)
                 == 1) {
                m = Mult.MANY_TO_ONE;
            } else m = Mult.ONE_TO_ONE;
        } catch (StringIndexOutOfBoundsException e) {
            m = Mult.ONE_TO_ONE;
        }
        return m;
    }
    public static String printArrayImport(Node n) {
        List<Arrow> la = n.getOutgoingArrows();
        boolean ret = false;
        for(Arrow a : la) {
            ret = hasOneOrManyToOtherConstraint(a);
            if (ret) return "import java.util.ArrayList;";
        }
        return "";
    }
    public static String getAttr(Arrow a) {
        if(hasOneOrManyToOtherConstraint(a)) return "private ArrayList
            <" +
                    a.getSource().getName() + "> " + a.getName() + ";"
                        ;
        else return "public " + a.getTarget().getName() + " " + a.
            getName() + ";";
    }
```

```
    public static boolean hasOneOrManyToOtherConstraint(Arrow a) {
        for(Constraint c : a.getGraph().getConstraints()) {
            if(c.getPredicate().getSymbol() != null &&
                    c.getPredicate().getSymbol().equals(
                        MULT_CONSTRAINT)) {
                for(Arrow tmp : c.getConstrainedArrows()) {
                    if(tmp != null && a != null && tmp.getId() == a.
                        getId()
                            && !parseConstraint(c.getParameters()).
                                equals(Mult.MANY_TO_ONE)) {
                        return true;
                    }
                }
            }
        }
        return false;
    }

    public static String getParamList(List<Arrow> aa, List<Arrow> aaa)
         {
        StringBuffer ret = new StringBuffer();
        ret.append(paramList(aa));
        if (aaa.size() != 0 && ret.length() != 0)
            ret.append(", " + paramList(aaa));
        return ret.toString();
    }
    private static String paramList(List<Arrow> aa) {
        StringBuffer ret = new StringBuffer();
        for(int i = 0; i < aa.size(); ++i) {
            Arrow a = aa.get(i);
            if(hasOneOrManyToOtherConstraint(a)) ret.append("ArrayList
                <" +
                    a.getSource().getName() + "> " + a.getName());
            else ret.append(a.getTarget().getName() + " " + a.getName
                ());
            if(i < aa.size()-1) ret.append(", ");
        }
        return ret.toString();
    }
    public static String getConstructorInit(List<Arrow> aa, List<Arrow
        > aaa) {
        StringBuffer ret = new StringBuffer();
        for(int i = 0; i < aa.size(); ++i) {
            Arrow a = aa.get(i);
            ret.append("this." + a.getName() + " = " + a.getName() + "
                ;");
        }
        return ret.toString();
    }
}
```

```
import dpf;
extension org::eclipse::xtend::util::stdlib::io;
packageName(DomainClass d):
    d.getAClassBelongsTos().get(0).target.name.replaceAll("\\.", "/")
        + "/";

String printArrayImport(DomainClass d):
    JAVA extensions.TemplateHelper.printArrayImport(no.hib.dpf.core.
        Node);

getDomainClassAttrRef(List[dpf::Attribute] attr, List[dpf::Reference]
    ref):
    let attrRet = attr.collect(e|getAttr(e)) :
        attrRet.addAll(ref.collect(e|getRef(e)));

String getAttr(dpf::Attribute a):
    JAVA extensions.TemplateHelper.getAttr(no.hib.dpf.core.Arrow);
String getRef(dpf::Reference r):
    JAVA extensions.TemplateHelper.getAttr(no.hib.dpf.core.Arrow);

String paramList(List[dpf::Attribute] attr, List[dpf::Reference] ref):
    JAVA extensions.TemplateHelper.getParamList(java.util.List, java.
        util.List);

constructorSetAttributes(List[dpf::Attribute] attr, List[
    dpf::Reference] ref):
    let constructorInitList = attr.collect(e|"this." + e.name + " = "
        + e.name + ";") :
        constructorInitList.addAll(ref.collect(e|"this." + e.name + "
            = " + e.name + ";"));
```

Listing 5.3: Listing shows the Xtend extensions for our generator.

Using Java extensions to mitigate the unpractical DPF API results in verbose code which we would like to avoid. The solution entails looping through all the constraints defined for the graph, finding the ones which are instances of the multiplicity predicate, and then looping through its constrained arrows comparing each one against the outgoing reference arrows of our `DomainClass` node. This can be achieved in a slightly more consise and elegant way using the Xtend language.

When we run the workflow, Xpand should generate two Java classes which contains valid code. The classes demonstrate handling multiplicity by checking if each reference is constrained by a multiplicity constraint. The example code is very simple and uninteresting, but has potential to define a lot more with few adjustments.

To make the example a little bit more interesting, we will generate a CRUD (create,read, update, delete) interface for our model classes. Fortunately Play makes this very easy for us with its built-in CRUD module.

5.5 Creating a Play Project

The Play framework comes with a plug-in for Eclipse. The functionality is limited due to the separate runtime that is provided with the framework. The Play runtime comes with a web server for testing the web application. Everytime a resource is changed and saved, the Play runtime will update the application on the fly. Creating a new Play project is achieved through the Play runtime's command-line interface (CLI). This tool creates and administers the projects. It controls which modules that should be active, running tests and it generates IDE specific configuration files for each project.

The first step is to obtain the Play framework from the project's web site. The version used in this example is 1.2.4, and not the most recent version. Version 2 was recently released with a lot of improvements. Unfortunately this release was too late for the use in this thesis.

1. Obtain Play from `http://www.playframework.org/documentation/1.2.4/install`. Follow installation instructions.
2. Create new project using `play new dpfcrud`.
3. Create Eclipse-project configuration with `play eclipsify dpfcrud`.
4. Import project into Eclipse workspace using the import wizard.

5.5.1 Configuring Play

The next step is to enable the CRUD module in Play. This is done through the **dependencies.yml** file in the `conf` directory. We must also resolve the dependencies and generate new Eclipse project files. Each module is created as a linked resource within the project, and must therefore update the `.project` file.

- *Open **dependencies.yml** and add "`-> crud`" after "`- play`".*
- *Run `play dependencies dpfcrud`.*
- *Run `play eclipsify dpfcrud`.*
- *Refresh your project in Eclipse.*

Play needs to know which database JPA should use. In this example we will use an in memory database for simplicity.

- *Open the **application.conf** file and add:*

```
db=mem
```

The configuration file shows examples on how to use other datasources like PostgreSQL and MySQL.

The next thing to do is redirect every HTTP request to the `/admin` area through the CRUD module. Defining which part of the application that handles specific URLs is done through the **routes** file in the `conf` directory.

- *Open the **routes** file and add the following rule before the last rule:*

```
*      /admin    module:crud
```

5.5.2 Modifying our Generator

We need to alter our generator to generate JPA annotations, as well as controllers for the CRUD module. The template file `templ.xpt` should look like so:

```
«IMPORT dpf»
«EXTENSION extensions::dpfplay»
«DEFINE main FOR dpf::Specification»
  «EXPAND graph FOR this.graph»
«ENDDEFINE»

«DEFINE graph FOR dpf::Graph»
  «EXPAND domainclasses FOREACH this.getDomainClasses()»
«ENDDEFINE»

«DEFINE domainclasses FOR dpf::DomainClass»
  «FILE "models/"+this.name.toFirstUpper() + ".java"»
    package models;

    «printArrayImport(this)»
    «EXPAND imports FOR this.getAAttributes()»
    import javax.persistence.Entity;
    import play.db.jpa.Model;

    @Entity
    public class «this.name.toFirstUpper()» extends Model {
         «FOREACH getDomainClassAttrRef(this.getAAttributes(), this.
             getAReferences()) AS e»
           «e-»
         «ENDFOREACH»

          public «this.name.toFirstUpper()»(«paramList(this.
              getAAttributes(), this.getAReferences())») {
            «FOREACH constructorSetAttributes(this.getAAttributes(),
                this.getAReferences()) AS e»
              «e-»
            «ENDFOREACH»
         }
    }
  «ENDFILE»
  «FILE "controllers/"+this.name.toFirstUpper() + "s.java"»
```

```
    package controllers;
    import play.mvc.*;
    import play.*;

    public class «this.name.toFirstUpper()+"s"» extends CRUD {

    }
  «ENDFILE»
«ENDDEFINE»

«DEFINE imports FOR List[dpf::Attribute]»
  «FOREACH this AS e»
    «IF e.target.name == "Date"»
      import java.util.Date;
    «ENDIF»
  «ENDFOREACH»
«ENDDEFINE»
```

Listing 5.4: Listing shows the updated Xpand template.

Listing 5.4 shows the updated template. As the code we are generating is still very simple, we have avoided changing the models. We have generated `@Entity` annotations for the model classes. The seasoned developer might notice that none of the datafields are generated with an `@Id` annotation which is mandatory. This happens in the `Model` class which we extend, where a generic id is created for each entity.

To generate the controller classes, we have introduced a new `FILE` statement which also generates a file for each `DomainClass`. Each model has a corresponding controller class. By convention the controllers have a plural name of the corresponding model, e.g. `Author` becomes `Authors`. To enable the CRUD functionality, we only need to extend the `CRUD` controller.

We have decided to ignore the `Package` as models and controllers always reside in their respective package (model and controller) in the `app/` source folder.

5.5.3 Running the Project

Now that everything is ready, we can run the workflow and generate our code. If desired you can hardcode the path to the Play project in the `src-gen` attribute in the workflow file. If not, you need to copy and paste the files into your Play project.

- *Run workflow.*
- *Copy and paste generated files into Play project. Model files goes in `app/models`, controllers goes in `app/controllers`.*
- *Use the Play console tool to run: `play run dpfcrud`.*
- *Point your browser to **`http://localhost:9000/admin/`**.*

If everything went according to plan, you should see a simple CRUD interface for your model classes like figure 5.5 and figure 5.6 shows.

5.5.4 Conclusion

The tutorial has shown how the process of creating a code generator works. We start off by defining our DSML and a sample instance model and corresponding textual output.

Through the chapter we have seen the basic building blocks and features of a Xpand template, everything from defining the output to using extensions for abstracting away complex code. Lastly, we have seen how to generate a simple CRUD interface using Play.

The complete example project can be found at `http://dpf.hib.no/downloads/`, which includes Eclipse integration for the project.

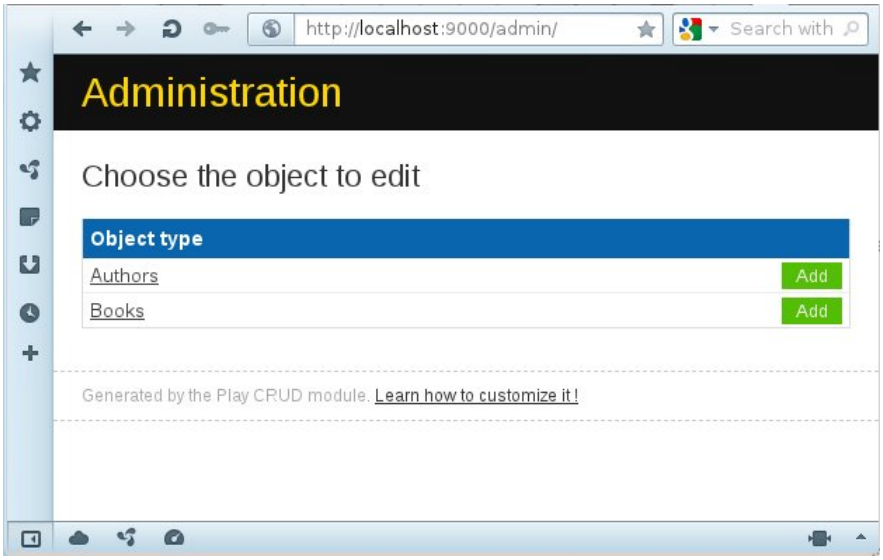

Figure 5.5: Figure depicts the CRUD interface which Play provides for simple model classes.

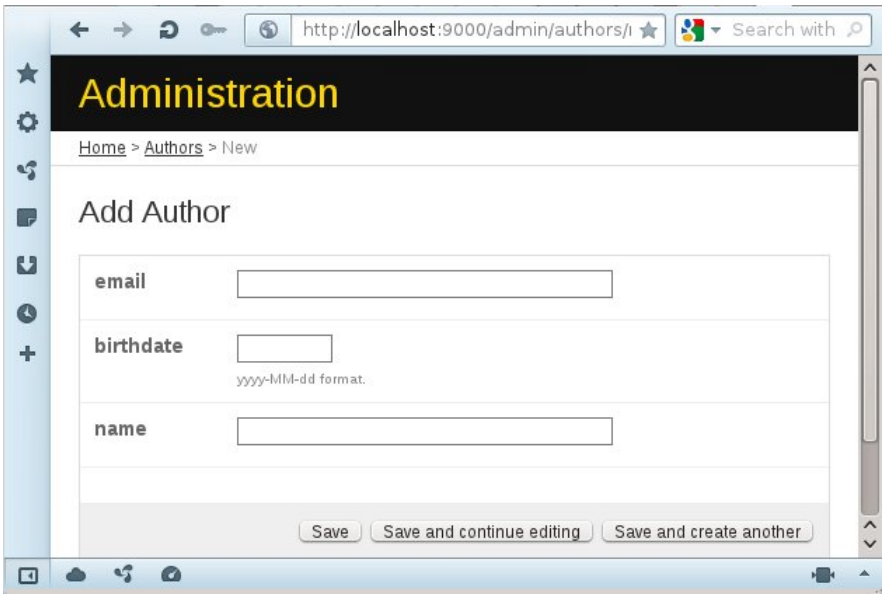

Figure 5.6: Figure depicts editing an Author object with the Play CRUD interface.

Chapter 6

Conclusion

This chapter will give an overview of what has been achieved through this project. The code generation tool which is developed in this project serves as a proof of concept solution which we will propose further work on. As the thesis has had some focus on the concept of language workbenches, we will propose some further work in this directon as well.

6.1 Summary

Previous efforts to create a DPF based tool has never reached the level of maturity needed to start serious work on code generation. With the work of Øyvind Bech and Dag Viggo Lokøen, we now have a diagrammatic editor which conveys the most important metamodelling concepts in DPF. The editor is built on top of a solid platform that enables us to create rich tooling for the DPF project. Through the chapters of this thesis we have created a solution for metamodel based code generation with the Xpand framework. Like the DPF Editor, this solution is based on Eclipse technologies and is thus well integrated.

Although the creation of a metamodel for Xpand is not a unique concept, we have yet to find any implementations of custom metamodels besides the ones provided with the framework.

Through the project we achieved the following:

General solution for creating code generators
: We have introduced a solution which can be applied to any DSML created with the DPF Editor. The solution provides a functional environment for creating code generators. Creating a metamodel for Xpand has enabled us to use the domain concepts of a DSML when defining templates, extensions and constraints checking. This is possible because the metamodel is a mapping from DPF specifications to types which Xpand understands. We have defined a

type system which represents each of the modelling constructs in DPF (i.e. node, arrow, constraint etc.). The types expose the functionality that is defined in the DPF Ecore metamodel, together with additional functionality that is defined to enhance the process of creating code generators.

Eclipse integration
With the implementation of a Xpand metamodel, we get a lot of functionality (almost) for free. Xpand provides a rich editing experience for templates, extensions and constraint checking based on the DPF Xpand metamodel. The editors feature code completion, syntax coloring, refactoring and error highlighting among others.

Moved DPF Editor towards becoming a language workbench
A DSML without any tooling around it is worthless. One of the most essential activities in a language workbench is model transformations, where model-to-text transformations might be the most pragmatic solution. Creating a usable language workbench without the support of creating code is arguably not a language workbench. Although the code generation activity can be seen as separate from creating the language itself, it is indeed part of a language workbench.

6.2 Further Work

This section will propose further work for the code generation tool, as well as some suggestions for features that takes the DPF Editor further towards becoming a language workbench.

6.2.1 Code Generation

The code generation tool should be regarded as a prototype as it have not been properly tested. The following section will explore some of the aspects which can be worked on.

Work on Shortcomings

Section 4.7 contains a small list of functionality which is not implemented at the time of writing. The tasks are relatively simple to implement, but might be time consuming.

Consistent API: This is not a shortcoming in the tool itself, but a project-wide problem. When relying on the DPF core API, one is very prone to errors due to modifications in the DPF Ecore metamodel (e.g. changing names, methods etc.). However, the biggest problem

with this is rendering all "older" DPF models invalid. A migration strategy is needed, both for the API and models.

Constraints and predicates as Xpand types: Predicates and constraints are not properly implemented using their designated types (`PredicateType` and `ConstraintType`) for various reasons. The current functionality might be sufficient for now, but hopefully adoption of the tool will help to identify where the need for alterations lies.

Test coverage: The DPF Xpand metamodel has not been properly tested. Creating tests is also a challenge, and hard without using DPF models directly. Spending development time on creating model based tests is a waste of time if the models become invalid after the first change to the DPF Ecore metamodel.

Namespaces: Implementing the support for more than one DSML per project was not a priority when developing the DPF Xpand metamodel. This is a relatively simple task to achieve.

Validating Constraints

The DPF Editor validates the different entities when creating a DSML. Each layer gets checked for the correct typing, and all constraints are validated. When creating a code generator the validation of constraints is left to the user. E.g. if a model has some kind of multiplicity constraint, there is no way to check if the constraint has been fulfilled in the generated code. Xpand provides a constraints checking language called *Check* (see section 3.7), but this language works on the Xpand types rather than checking if the template code has fulfilled the constraints, which is defined within the model. Check can however be used to validate the generator's input models if this is desired.

An idea can be to define a post-processor (see Xpand documentation [54]) which scans the output and checks statements against the model.

Investigate Xtend 2

In section 3.7.1, the Xtend 2 language is introduced as the replacement for Xpand. Xtend 2 is a general programming language (GPL) which compiles to Java. Unfortunately Xtend 2 is not compatible with Xpand as it is a completely different approach (Xpand uses an interpreter). Xtend 2 has not been properly investigated to draw any conclusions on if it can be used in the DPF Editor providing the same functionality as Xpand. At the time of writing there are indications which suggests that Xtend 2 is not able to fulfill the use case that Xpand fills, namely interpreting models runtime, and providing an editing environment based on the content of the model. The good news is that the template language used in Xtend 2 is Xpand,

which means any templates written in the current tool, will (largely) be compatible with Xtend 2. This does not apply to any extensions specified, as these are written in Xtend 2.

Although Xpand's future is uncertain, it will be supported a while longer. This means that bugs will be fixed, but no larger releases will occur.

A few solutions to this problem can be:

- Migrate our solution to Xtend 2
- Fork the Xpand project
- Create an in-house code generation solution
- Find another solution (e.g. Acceleo)

It is the author's opinion (based on the current knowledge) that Xpand is a better solution than Xtend 2. Xtend 2 is a GPL that uses Xpand "template expressions" within the code, thus mixing code and templates. As stated, there does not seem to be any way of interpreting DPF models, which means we are back to using the Ecore instance directly (see section 4.3).

This is not a pressing issue as Xpand is a modern solution with a very stable code base.

6.2.2 Language Workbench

A part of the DPF project's vision is to create a working tool that increases the developer's productivity, and do not exist solely for academic purposes. This thesis has introduced the term language workbench, which is a pragmatic take on the MDE methodology. In a language workbench the tooling which supports the modeler and its users are of upmost importance. This section will propose further work that will move the DPF Editor even closer towards the goal of becoming a productive language workbench.

Symbol Editor

One of the challenges the DPF Editor faces is visualizing the DSMLs. In chapter 2 we looked at MetaCase's MetaEdit+ tool, which has the ability to create custom symbols for their *object* modelling entity. This is an effective way of communicating the intent of the language and its concepts. In DPF there is no support for attributes on the nodes, there is only nodes and arrows. A DSML of only nodes and arrows can look a lot more complex than it actually is, which then becomes a usability problem.

To tackle the problem with complex models, one can create create symbols for nodes and arrow. Currently there is support for using simple pre-defined shapes, like ellipse, square, circle etc.

The most recent work by Ph.D. student Xiaoliang Wang has resulted in a signature editor where one can define custom predicates. All predicates needs a symbol, which means a symbol editor would be a nice fit for this requirement as well. The symbol editor can be a simple drawing application, with the possibility of importing graphics.

Visualization for DSMLs

Even though symbols will create a better user experience when modelling, the problem of only having nodes and arrows are not dealt with. Our models are centered around graph concepts. A solution to this problem is to create a visualization independent of the underlying model structure. One can group different nodes and arrows, and create a visualization for the group. As an example we could group the `DomainClass`, `Attribute` and `Type` concepts from the DSML in chapter 5, and create a visualization similar to an UML class.

This way DSML modelers have a greater freedom in how he/she wants to present the domain concepts of the created language. One could also ship the tool with pre-defined DSMLs, visualizations and code generators that defines e.g. UML class diagrams, a language for modelling Android apps, petri-nets etc.

A solution like this is as far as the author knows unique, and would probably give an advantage over other language workbenches.

Textual Representation of DPF models

The comparsion at the end of chapter 2 mentioned the need for a textual representation of models in addition to projectional representations in language workbenches. Of the more popular solutions on the market, no one seems to have this feature. The language workbenches are either textual or graphical. An optimal solution would be a textual, human readable representation of a DPF model which the graphical representation was based on. Any modifications in either model should be reflected in the other.

We have through Florian Mantz' Ph.D. work, an external DSL for DPF which could be integrated with the DPF Editor.

File Format

The current state of file formats in DPF are XMI serializations of the specifications, as well as the corresponding visualization[1]. A file format could be an archive file which contained all the specifications in a

[1] The serialization of the visualization is recent work by Ph.D. student Xiaoliang Wang, and was not implemented for the most part of this projects' time span

metamodelling hierarchy, together with its visualizations and signature. If a textual model representation is integrated in the DPF Editor, there needs to be room for this in the file format as well. An important aspect of this task is to create a file handling facility which can keep compatibility between different versions files, and the models within.

Version Control System for Models

Creating a VCS for DPF models would be another unique feature for the DPF Editor. This subject was the focus of Alessandro Rossini's Ph.D. thesis [46] which provides a formal approach to the problem.

6.3 Final Words

The work on this thesis has resulted in a prototype code generation tool, which provide the opportunity to create code generators for an arbitrary DSML. Code generation is an important activity in MDE, and thus an important feature in a language workbench. A usable code generation facility is critical for the success of the DPF Editor.

Even though the Xpand framework's future is uncertain, it is a state of the art tool. The author is confident that Xpand and the prototype developed in this thesis is a solid foundation for further work.

Bibliography

[1] Acceleo. *Project Web Site*. http://www.eclipse.org/acceleo/.

[2] Apache Ant. *Project Web Site*. http://ant.apache.org/.

[3] Apache log4j. Project Web Site. http://logging.apache.org/log4j/.

[4] Øyvind Bech. DPF Editor – A Multi-Layer Modelling Environment for Diagram Predicate Framework in Eclipse. Master's thesis, Department of Informatics, University of Bergen, Norway, May 2011.

[5] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[6] Frank Budinsky, Ed Merks, and David Steinberg. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2006.

[7] Peter Pin-Shan Chen. The entity-relationship model toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, March 1976.

[8] Steve Cook and Stuart Kent. The domain-specific ide. *UPGRADE*, IX:17–22, April 2008.

[9] International Business Machines Corp. Eclipse platform technical overview, 2006.

[10] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9:671–678, September 1966.

[11] Zinovy Diskin and Boris Kadish. Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling. *Data Knowl. Eng.*, 47(1):1–59, 2003.

[12] Django Project. *Web Site*. http://www.djangoproject.com/.

[13] DPF: Diagram Predicate Framework. *Project Web Site*. http://dpf.hib.no/.

[14] Eclipse. Community Forums. http://www.eclipse.org/forums/.

[15] Eclipse Naming Conventions. *Project Web Site*. http://wiki.eclipse.org/Naming_Conventions.

[16] Eclipse Platform. *Project Web Site*. http://www.eclipse.org.

[17] Eclipse Xtend. *Project Web Site*. http://www.eclipse.org/xtend/.

[18] Eclipse Xtext. Project Web Site. http://www.eclipse.org/Xtext/.

[19] Eclipse Xtext. Xtext/Xtend Documentation Code Generator Example. http://www.eclipse.org/Xtext/documentation/2_1_0/040-first-code-generator.php.

[20] Edgewall Software. Trac Project Web Site. http://trac.edgewall.org/.

[21] M. Fowler and R. Parsons. *Domain-specific languages*. Addison Wesley Signature Series. Addison-Wesley, 2010.

[22] Cesar Gonzalez-Perez and Brian Henderson-Sellers. Modelling software development methodologies: A conceptual foundation. *J. Syst. Softw.*, 80:1778–1796, November 2007.

[23] Grails. *Project Web Site*. http://grails.org/.

[24] Ørjan Hatland. Sketcher .NET – A drawing tool for generalized sketches. Master's thesis, Department of Informatics, University of Bergen, Norway, June 2006.

[25] Jack Herrington. *Code Generation in Action*. Manning Publications, revised edition, July 2003.

[26] IBM. *Reflection Tutorial*. http://www.ibm.com/developerworks/library/j-dyn0603/.

[27] Itemis. Company Web Site. http://www.itemis.com.

[28] Java Naming Conventions. *Project Web Site*. http://www.oracle.com/technetwork/java/codeconventions-135099.html.

[29] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.

[30] Christian Kroiss, Nora Koch, and Alexander Knapp. Uwe4jsf: A model-driven generation approach for web applications. In Martin Gaedke, Michael Grossniklaus, and Oscar Díaz, editors, *Web Engineering*, volume 5648 of *Lecture Notes in Computer Science*, pages 493–496. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02818-2_46.

[31] Language Workbench Competition. Competition Web Site. http://www.languageworkbenches.net/index.php?title=Main_Page.

[32] Lift Web Framework. *Web Site*. `http://www.liftweb.net/`.

[33] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[34] Mercurial. Project Web Site. `http://mercurial.selenic.com/`.

[35] MetaCase. Company Web Site. `http://www.metacase.com/`.

[36] MetaCase. *MetaEdit+ Manual*. `http://www.metacase.com/support/45/manuals/mwb/Mw.html`.

[37] Object Management Group. *Web site*. `http://www.omg.org`.

[38] Object Management Group. *Meta-Object Facility Specification*, January 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[39] Object Management Group. *Object Constraint Language Specification*, May 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

[40] Object Management Group. *Unified Modeling Language Specification*, November 2007. `http://www.omg.org/cgi-bin/doc?formal/2007-11-04`.

[41] Object Management Group. *MOF Model to Text Transformation Language Specification*, January 2008. `http://www.omg.org/spec/MOFM2T/`.

[42] OMG Model Driven Architecture. *Web Site*. `http://www.omg.org/mda/`.

[43] openArchitectureWare. *Reference Manual*, December 2008. `http://www.openarchitectureware.org/pub/documentation/4.3.1/openArchitectureWare-4.3.1-Reference.pdf`.

[44] Yngve Lamo Florian Mantz Øyvind Bech, Adrian Rutle and Xiaoliang Wang. DPF Editor: A Multi-Layer Diagrammatic (Meta)Modelling Environment. In *SPLST 2011: 12th Symposium on Programming Languages and Software Tools*, October 2011.

[45] Play Framework. *Project Web Site*. `http://www.playframework.org/`.

[46] Alessandro Rossini. *Diagram Predicate Framework meets Model Versioning and Deep Metamodelling*. PhD dissertation, Department of Informatics, University of Bergen, Norway, 2011.

[47] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD dissertation, Department of Informatics, University of Bergen, Norway, 2010.

[48] Adrian Rutle, Uwe Wolter, and Yngve Lamo. Generalized Sketches and Model-Driven Architecture. Technical Report 367, Department of Informatics, University of Bergen, Norway, February 2008.

[49] Stian Skjerveggen. (Towards an) Implementation of a Graphical Editor for Diagrammatic Predicate Logic in the Eclipse Platform. Master's thesis, Department of Informatics, University of Bergen, Norway, June 2008.

[50] Sven Efftinge. Sven Efftinge's Blog. `http://blog.efftinge.de/2010/12/xtend-2-successor-to-xpand.html`.

[51] The Agile Alliance. *Project Web Site*. `http://www.agilealliance.org/`.

[52] The Eclipse Graphical Modeling Project. *Project Web Site*. `http://www.eclipse.org/modeling/gmp/`.

[53] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008.

[54] Xpand. *Project Web Site*. `http://wiki.eclipse.org/Xpand`.