MODEL DRIVEN SOFTWARE DEVELOPMENT:
EXTENDING THE PROGRAMMING PARADIGM

ABSTRACT

Keywords: Model Driven, MDSD, openArchitectureWare, OAW, xtext, EMF

The term “model driven” has many facets in software engineering all spun around the vision of enabling agile business solutions at the push of a button. Its utilisation ranges from business modelling via architecture to coding. Its intent is unique, namely trying to achieve automation via formalization. This paper focuses on model driven software development (MDSD) and what it means to introduce it to real life projects.

Application features are usually described via domain models using specific business vocabulary like Customer, Order or Invoice. Meta-models describe a domain-models structure. They use terms like Entity, Relation or Operation. This meta-information is the key to model driven software development (MDSD). The MDSD idea is to establish an automation process based on meta-information for creating runtime behaviour. The latter can be achieved by either generating code at development time or by parsing meta-information at run time.

openArchitectureWare (www.openarchitectureware.org) as part of the Eclipse Modelling Project, amongst others, conducts a vast collection of projects simplifying the use of meta-information for that purpose.

Technologies discussed in this paper are based on the EMF (Eclipse Modelling Framework). EMF provides a modelling framework and code generation facilities for building tools and other applications based on meta-information. The paper shows what must be considered in a real life project when MDSD is applied. Necessary skills and organizational requirements are discussed. Improvements that can be made on code quality and consistency are highlighted. The concept of horizontal and vertical consistence is introduced.

An example in the paper outlines how xtext and related tools as part of the openArchitectureWare Eclipse Modelling Project can be used to create a domain specific modelling syntax for your own project. It shows how to create your own domain specific language (DSL) which can be tailored and extended for individual automation purposes. Requirements are expressed in terms of that DSL and runtime behaviour created from it at the push of a button. The paper sketches how the magic happening behind the button can be achieved. It is a challenging task but has always proved to be profitable. Requirement changes are a law of nature. You cannot appreciate enough to be able to spread those changes from one source, the DSL specification, across all your project artifacts instantly.
It is the authors’ belief that applying model driven software development (MDSD) in real life projects will change one’s view of programming. It will become a commonly used paradigm and the question is not IF but WHEN.
MDSD, EXTENDING THE PROGRAMMING PARADIGM

TABLE OF CONTENT

Abstract ................................................................................................................................. 1
1 Introduction ....................................................................................................................... 5
  1.1 Abstract....................................................................................................................... 5
  1.2 Motivation .................................................................................................................... 5
  1.3 Document Structure ................................................................................................. 6
2 General Concepts for MDSD ........................................................................................... 7
  2.1 Automation through Formalization ........................................................................... 7
  2.2 Domain Specific Language ....................................................................................... 9
  2.3 Integration of existing Frameworks ............................................................................ 9
  2.4 The Consistence Boost ............................................................................................ 11
  2.5 Template Libraries ................................................................................................... 12
     2.5.1 Template Reuse ............................................................................................... 12
     2.5.2 Template Rules ............................................................................................... 12
     2.5.3 Template Design ............................................................................................. 12
3 MDSD Projects ............................................................................................................... 14
  3.1 Technical Approach ................................................................................................. 14
     3.1.1 Learning New Languages ................................................................................. 14
     3.1.2 AOP .................................................................................................................. 18
     3.1.3 The Workflow .................................................................................................. 19
  3.2 Activities related to MDSD ...................................................................................... 20
     3.2.1 Model Driven Design Activities ...................................................................... 20
     3.2.2 Model Driven Development Activities .............................................................. 21
  3.3 Development Processes Change ................................................................................ 22
     3.3.1 Responsibilities and Roles ................................................................................ 23
  3.4 MDSD and Catalyst ................................................................................................... 25
     3.4.1 MDSD and Catalyst Guiding Principle ............................................................... 26
     3.4.2 MDSD and the Catalyst Framework Concept .................................................... 27
     3.4.3 MDSD in the Catalyst Lifecycle ....................................................................... 28
  3.5 The Remaining 20% ................................................................................................. 30
4 Summary ......................................................................................................................... 32
  4.1 Experiences ............................................................................................................... 32
  4.2 Best Practices ............................................................................................................ 32
     4.2.1 Separate Generated and Manually Written Code .............................................. 32
     4.2.2 Do Not Check in Generated Code ................................................................. 32
     4.2.3 Integrate the Generator in the Build Process .................................................. 32
     4.2.4 Generate Clean Code ..................................................................................... 32
     4.2.5 Talk “meta” ...................................................................................................... 32
     4.2.6 Develop Iteratively ......................................................................................... 32
     4.2.7 Use a Reference Model .................................................................................... 33
     4.2.8 Use a Simple Syntax ....................................................................................... 33
     4.2.9 Use Model-to-Model Transformation ............................................................. 33
     4.2.10 Do Not Reinvent, Use Existing Frameworks ............................................... 33
  4.3 Pro & Contra .............................................................................................................. 33
  4.4 The Future .................................................................................................................. 34
5 Appendix ......................................................................................................................... 35
5.1 Abbreviations .................................................................................. 35
5.2 Links .............................................................................................. 35
5.3 Books ............................................................................................ 35

Disclaimer .................................................................................................. 36

LIST OF FIGURES

Figure 1. Model Driven Approach ................................................................. 8
Figure 2. Model Drives Application ............................................................. 10
Figure 3. Horizontal and Vertical Consistency ........................................... 11
Figure 4. MDSD Iterations ........................................................................ 22
Figure 5. “meta” Skills in the MDSD Development Process ....................... 24
Figure 6. “Project” Skills in the MDSD Development Process ................... 25
1 INTRODUCTION

1.1 ABSTRACT
The term “model driven” has many facets in software engineering all spun around the vision of enabling agile business solutions based on dedicated methods and technologies. Its utilisation is ubiquitous ranging from business modelling via architecture to coding. Its intent is unique trying to achieve automation via formalization.

This paper focuses on model driven software development (MDSD) and what it means to introduce it to real life projects. It describes general concepts, and our selection of tools available from the open source community and how they can be utilized to make software development model driven.

1.2 MOTIVATION
There are many ways to benefit from model driven approaches in real world software development projects. The key issue is meta-information which is used to create runtime behaviour. Meta-information describes the model of a model. It is a relative term in that respect and called a meta-model. To get an impression of what this means in the context of this paper think of the model being expressed by elements like Customer, Book, Chair, or Table. These elements belong to a certain business domain. That’s why the corresponding model is often denoted as domain-model. Examples for elements used in the associated meta-model would be Entity, Relation or Operation. These elements represent items used for modelling the domain-model and constitute the meta-model.

Designing meta-models involves an additional level of abstraction. This needs to get used to. But “thinking meta” opens new perceptions of how software development can be performed. Domain-models describe structure and functionality of a business formalized via meta-model syntax. The idea is to use this information and syntax structure to automate software artefact creation.

It can be described in short as follows: After an artefact has been written, ask which information would have been necessary to automatically create the same runtime behaviour from it, create a meta-model that can represent this information, find means to generate the desired runtime behaviour from it and next time do not write the artefact to begin with. This can be thought of as an extension of the programming paradigm – extended by means of meta-modelling.

Runtime behaviour can be achieved in two essentially different ways, either by generating code at development time or by interpreting meta-information at runtime. Both types are used by well known open source products such as Hibernate, Spring or JSF.

The structure of meta-information needed to create specified runtime behaviour may differ significantly between projects depending on what the requirements are. Requirements depend on project domain and general modelling languages need to be extended with or replaced by domain specific features to provide sufficient information for automating software generation. These extensions or replacements are called domain specific languages (DSL).

DSL’s are a commonly used approach for purposes of model driven software development. Until not too long ago introducing automation to a project, meant that one had to self write modelling facilities as well as generation tools. Today there is a
vast selection of mature open source tools and methodologies available supporting individual DSL specification and artefact generation from it.

Most technologies discussed in this paper are based on the EMF (Eclipse Modelling Framework) and have been published as part of the Eclipse Modelling Project. EMF provides a modelling framework and code generation facilities for building tools and other applications based on a structured data model. The tools introduced can be found in the open architecture ware platform which provides a modular generator framework implemented in Java™.

It is our belief that applying model driven software development (MDSD) in real life projects will change ones view of programming. It will become a commonly used paradigm and the question is not IF MDSD will become part of the software development process but WHEN.

1.3 DOCUMENT STRUCTURE
Model Driven Software development affects all phases of a project’s lifecycle. It is based on essential principles and presumes a dedicated approach. On the other hand MDSD promises benefits and is able to keep the promises.

Chapter 2 introduces the general concepts and consequences for software development when going “model driven”.

Chapter 3 gives a detailed view of how some development activities are affected by MDSD. It also describes MDSD’s relation to other methodologies and serves as compendium for project teams that want to develop software in a model-driven way. It describes the influence of MDSD on the development cycle, new roles, responsibilities and skills required by the Model Driven Software Development.

Chapter 4 is the summary of this document. This chapter includes best practices, pros and cons and anticipates a possible future of MDSD.

In Chapter 5 you will find some useful information to get started with MDSD.
2 GENERAL CONCEPTS FOR MDSD

Models provide structured and abstracted views of real world systems and have become an indispensable part of every methodology in software engineering. The most used modelling language today is probably UML (or rather UML2) but there are others around like MOF (Meta-Object Facility), XMI (XML Metadata Interchange) or CWM (Common Warehouse Meta-model).

A majority of projects however still use a coding oriented approach when it comes to implementation. Models created in earlier project phases are used as reference to check requirements instead of using them as powerful sources for creating functionality via appropriate automation processes.

The vision of model driven software development is to create full system implementations solely from the model information. To this end existing models have to be enhanced to provide a standard project vocabulary (together with a dedicated grammar) for use during development as well as sufficient information to automatically create working software. Hence much of the effort during project realization goes into designing an appropriate domain model providing for all necessary information. This is usually more than documented by standard modelling techniques.

This chapter describes how domain modelling targeting automation can be performed. The concept of domain specific languages, templates and generation mechanisms are outlined and how they provide for automation.

2.1 AUTOMATION THROUGH FORMALIZATION

Automation through formalization is achieved by providing structured data that can be used to generate software artefacts. Structured data is split into a twofold set of models. The meta-model describing the structure and the domain-model containing the data. Data in the domain-model adhere to the structure defined in the meta-model. Artefacts are created by a specific generation mechanisms which understands the meta-model, based on this reads the structured data and applies them to a set of templates. Templates in return specify rules how to turn the structured data into code.

The concept of automation as understood in this paper is outlined in Figure 1. The meta-model is specified in terms of a domain specific language (DSL). A DSL specifies syntax rules for the grammar used to collect model information. There are many ways to define such a grammar. We will come to that later. Graphical as well as textual representations are possible. For the moment it suffices to think of it as XML data where an underlying XSD schema represents vocabulary and structure, i.e., the meta-model.

Using the meta-model many domain-models can be created each representing an instance of that particular meta-model. If we assume the domain to be the persistence layer of a web application deployed into an application container and our DSL was designed to describe data entities and their relations, a dedicated persistence layer could be specified for every data model that can be expressed in terms of our meta-model.
In Figure 1 models on the left hand side all adhere to the DSL and a dotted arrow points to that. They are instances of the meta-model represented by that DSL.

A generator and templates provide for creating project artefacts from a given model. The generator can understand models written in the given DSL syntax. It reads model information, associates it with appropriate templates and creates artefacts from it.

There is no restriction as to which artefacts can be generated. In order to find candidates one has to look for structural or behavioural patterns. They can be found for example in data model related artefacts, such as Java™ Beans, POJO’s (Plain Old Java Objects), their DAO’s (Data Access Objects) or Relations. Other candidates are Business Objects, Validations, Navigations, UI-Handling, Facades, State-Transitions, Forms, Dialogs, Wizards, not to forget Reports and much more.

Generally speaking, if you find an appropriate meta-model representation for the information required to automatically generate an artefact, then generate it. The problem is to decide what appropriate means in this context. This is related to effort, complexity and reusability benefit of the generation mechanism evolving from it.

Automation refers to creating runtime behaviour for a targeted application. This does not necessarily mean that source code is generated. Specific runtime behaviour can also be achieved via configuration. The essential difference is that code generation happens at development time, interpreting configuration information at runtime.

Property files are typical but very simple examples for configuration. Much more complex types of configuration exist. The ultimate configuration would be a DSL-based model instance from which the complete runtime behaviour is derived. This can be thought of as a virtual machine for the given meta-model or DSL.

We shall use the term Software Generation to represent both creating artefacts from meta-information at development time as well as creating functional behaviour at runtime.

Which way of software generation is best depends on project requirements. Creating code at development time is often favoured as from a developer’s point of
view writing templates is closer to writing code than designing virtual machines. Theoretically both types of software generation are equivalent; they just differ with respect to the point in time when model information is used.

In the remainder of this document we will however concentrate on code generation at development time. This is best supported by the tools we refer to in chapter 3.

2.2 DOMAIN SPECIFIC LANGUAGE
A DSL is a formal language that is dedicated to one concrete domain. It is able to describe all aspects of that domain but nothing beyond your domain. A domain in this concept has to be seen as a relative term. There is no strict definition for what a domain is. Hence it is the designer’s decision to define the coverage of the DSL.

There are typically two major classifications for domains. The motivation for defining a domain can either be founded in a business related view or in terms of a technical perspective. A business motivated domain could involve terms like account, interest and remittance to describe the domain of finance. Terms like one-to-many, nullable or PK would build the core of a technically motivated DSL.

The basis for each DSL is its abstract syntax. The DSL can only be interpreted on the basis of a set of rules that define the DSL’s structure. The abstract syntax of a DSL therefore defines its grammar and is the substantial basis for any tool to process instances of the DSL. Applying MDSD one can make use of existing abstract syntax specifications like XText, which is based on the Eclipse Modelling Framework and comes with many tools and features such as an editor, a generator, code completion and syntax highlighting.

Generally DSLs can either be graphical or textual. Both have their strength as well as limitations. Which type of DSL is most appropriate to use must be decided in the concrete context of the project. Graphical tools are usually easier to learn and use. They illustrate the domain model and are suitable for use in documents and presentations. They are designed to be used intuitively. On the other hand they are not flexible and difficult to merge in a team environment. UML2 tries to solve those limitations but new problems occur. The usage becomes more complicated and therefore shifts the difficulties.

Using tools provided as part of the openArchitectureWare framework has made textual DSL development very simple. Frameworks for textual DSLs come with all features and tools necessary for an effective practical use. These frameworks allow “one-click” generation of editors and generators supporting exactly the grammar defined for the DSL. Features like syntax highlighting and code completion come out of the box. Thus the acceptance of such frameworks has increased. Syntax checking is available within the editor as well. The generator allows for processing of concrete models that were written in the underlying DSL. Designing your own DSL has become straightforward.

2.3 INTEGRATION OF EXISTING FRAMEWORKS
Many state of the art frameworks already generate software based on meta-information to create runtime behaviour. There is Hibernate, iBATIS, Toplink, Castor, JPA, JSF or Spring to name only a few. When employing model driven techniques these frameworks should be utilizable as they are without restrictions.
In fact model driven software development will put no restrictions on using existing software components or frameworks. Quite contrarily it can provide helpful means creating necessary glue between or supplements to them.

This is illustrated by an example. If for instance a project wanted to use Hibernate with JPA annotations to create a persistence layer, a simple DSL for modelling Java Beans can be used to document a model with class names, attributes and relations and create the annotated classes from it, nothing else. Hibernate does the rest, just as if one had written the annotated classes by hand. The difference is that we have a DSL based model instance. This contains most of the information to create basic service objects for the business layer or even JSP’s for presentation. Often simple extensions to the meta-model are sufficient to create additional artefacts across all layers of a project.

Suppose for instance that JSF in combination with Spring are used, which both provide sophisticated bean management based on XML configuration, a model driven approach could still be used and create code for the managed beans and (glue-) artefacts like faces-config.xml or applicationContext.xml configuration files for it.

This constellation is outlined in Figure 2. The shaded area at the top left symbolizes the meta-model specified by the underlying DSL. It represents the syntax for documenting models. An excerpt of an instance of such a model is shown underneath it. The arrow indicates that it was created adhering to the given DSL syntax. Generators or templates are not shown in Figure 2 but from this model several artefacts could be generated using the DSL’s template and generation mechanisms. Amongst these artefacts are JSP’s, faces-config.xml, and service beans, applicationContext.xml, annotated Java™ beans, DAO’s and possibly SQL and much more.

Hibernate would then be employed to create the database and access mechanisms, Spring and JSF for their bean management.

![Figure 2. Model Drives Application](image)

Framework functionality is used as if all sources were written manually and not generated via MDSD mechanisms.
The basic principles when using existing frameworks in a model driven approach are as follows:

- Use existing framework functionality,
- If the framework requires some kind of meta-information, try to generate that meta-information from your domain model,
- If your domain model does not contain that information, put it in,
- Never try to replace framework functionality or even parts of it.

This way a perfect match between MDSD and the use of existing components or frameworks can be achieved.

2.4 THE CONSISTENCE BOOST

Consistence is a big issue in most projects. It can refer to mass or depth. Similar things should be solved following comprehensible and coherent conventions. With respect to mass for example Java™ beans should all look alike or Dao-methods should all appear in the same order. With respect to depth a column name in the database should be recognizable in the associated class, the service bean or the JSP and derived from the name in the database by a consistent convention.

Violating consistency can become annoying. Consistency management is tedious and error prone whenever more than one source exists for the same piece of information. The key is to provide unique sources of information and this is what the model driven approach strives for. If an Entity, its attributes and their types are given in one place, that's it. There should not be anywhere else this information is repeated. No matter whether an attribute name appears as a column in the database, a member variable in a Java-Bean, a parameter in a validation method or a tag-name in a JSP, it should be provided from one source, the model.

Consistency over different types of artefacts across architectural layers is often called vertical consistency. Vertical consistency always regards to different types of artefacts such as sql, java or jsp.
Corresponding to this consistency referring to the same type of artefact is called horizontal consistency. This is used to describe consistent handling of the same type of artefact across many instances. Hence if all of a project’s JSF files were generated following the same rules and patterns they would be called horizontally consistent.

The MDSD approach supports both types of consistency. Vertical consistence is based on the fact that every piece of model information has a unique source. Horizontal consistency is ensured by choosing unique templates for a given functionality.

2.5 TEMPLATE LIBRARIES
As we focus on source code generation rather than writing virtual machines for a DSL we need a possibility to generate text from the domain model. The generator needs to be aware of the DSL syntax and each DSL needs its own generator. The DSL and its generator can be considered a domain specific generator. The generator takes two types of input; one is represented by the domain model and the other by the templates that describe how to transform the domain model into suitable text. This transformation is known as M2T (model to text).

2.5.1 Template Reuse
Each template in this context is designed to serve a specific concern. There might be one template that transforms entities of the domain model into concrete Java™ Beans, one template to generate asp files and another one to produce resource files for internationalization (I18N). The generator would process all of them and produce source code for the runtime platform. Each concern needs an individual template. Templates are usually target platform specific. Templates designed for a specific target platform and concern may contribute to a template library. In a template library domain specific generator fragments are gathered for possible reuse.

2.5.2 Template Rules
Generated code should be clearly structured, comprehensive and well documented as if it was written by hand, or even better. This is a key issue to ensure readability and maintainability.

- Generate readable code: If an error occurs in the application the source code would be inspected first. Think of the maintenance activities
- Write readable templates: Source code is used to identify errors but the template is the place to fix the error.
- Modify the meta-model if necessary: If you find yourself faced to a very complicated task to define the template think of redesigning the meta-model.

2.5.3 Template Design
There will be the need for individual code that cannot be created via general purpose DSL means for almost every application. Templates and hence the application have to cater for this. To this end means must be provided to implement extensions or modifications of any generated behaviour. This could be achieved by OSGI-like extension point mechanisms, using inheritance or interfaces. We shall not disclose more details on this here. It represents, however, an important aspect of the overall application design when employing code generation.
The overall purpose of MDSD is to build pluggable components for reuse. A domain specific generator and templates designed for a distinguished concern is also denoted as a *cartridge*.

A cartridge would for example produce the DDL scripts to generate tables, columns, primary keys and constraints for a relational database based on a given domain model.
3 MDSD PROJECTS

After the first theoretical approach of MDSD and its general concepts, this chapter will give a hands-on example of how MDSD along with available tools and frameworks can be used to create runtime behaviour by means of code generation. To this end we will introduce a simple domain model and show how development concepts and frameworks are integrated with MDSD. Our aim is to encourage architects and developers to use these tools and underlying methodologies as an extension to their habitual design or development habits.

3.1 TECHNICAL APPROACH

We will describe the use of XText to define the DSL and its grammar, how AOP simplifies templates and the processing of the transformation over the openArchitectureWare Workflow. All components are based on the Eclipse Modelling Framework (EMF) and can be downloaded and used for free. The aim of this chapter is to give a short overview of how to create artefacts on the example of a persistence layer. The same approach can be used for all types of artefacts across all layers of an application.

3.1.1 Learning New Languages

There are basically three programming languages involved in creation, extension and processing of a domain specific model, XText for designing the DSL, XTend to enhance the model's features and XPand, the template language, to transform a model into text using source code generation. These languages are easy to learn but provide powerful means for MDSD. Let us have a look at them.

3.1.1.1 XText

XText can be classified as an external textual DSL. The basis for a textual DSL is its grammar. The definition is based on a set of rules. We decided to define our own DSL and therefore started creating a new XText project called CarRental:

The wizard creates three new projects CarRental.dsl, CarRental.dsl.editor and CarRental.dsl.generator. The file rentaldsl.xtext is opened. Within this single file one will define the grammar for rentaldsl.
As the DSL files allow to import different grammars we decided to swap base definitions into a separate file called `base.txt`. The whole grammar can be defined within one single file, however, in terms of reusability and clarity we decided to spread different concerns to different sources.

```plaintext
Enum BOOL:
   TRUE="true" | FALSE="false";

Enum Type:
   aString="String" | aName="Name" | aDate="Date" | aDateTime="DateTime"
   | aCurrency="Currency" | anInt="int" | aLong="long"
   | aFloat="float" | aDouble="double" | aBoolean="Boolean";

Enum Cardinality:
   one="1..1" | noneOrOne="0..1" | noneOrMany="0..*" | oneOrMany="1..*";

String QName:
   ID("ID") *;
```

The file `datamodel.txt` is based on the above grammar and extends it.

```plaintext
importGrammar "base.txt"

DataModel:
   "DataModel" name=QNAME "("
   (dataModelItems+=DataModelItem)*
   ");

DataModelItem:
   Entity | Enumeration | Relation;

Entity:
   (isAbstract="abstract") ? "Entity" name=ID ("extends" extends=[Entity]|(""
   |attributes+=Attribute) | "")";
```

The listing is just a cut-out of the complete file. The two files `base.txt` and `datamodel.txt` are now imported into the top level grammar file `rentaldsl.txt`. In this file the application model is described as a container for a data model. Typically the grammar for the rental application model would also contain a definition for a business model as well as a presentation model. As our intention is to show that the setup of an XText project is quite easy we also keep the example easy.

```plaintext
importGrammar "datamodel.txt"

ApplicationModel:
   "ApplicationModel" name=QNAME "("
   (dataModel + DataModel)?
   ");
```

The grammar of rental DSL is already defined at this moment. The next step is to generate XText artefacts. This is done by opening the context menu of the file `rentaldsl.txt` (right click) and choosing `Generate XText Artefacts`. All necessary files are built upon the grammar.

In the next step the three related projects are exported as Deployable Plug-Ins into the plugin directory of the eclipse installation. After restarting Eclipse the menu entry `New -> Other -> rentaldsl project` allows to open a project with full support for the defined DSL.
The project we are now working in can be considered a rentaldsl editor. The file model.dsl is opened in the editor. This is the place where all information of the concrete model goes. The following listing describes a valid instance of the model defined by means of the underlying meta-model.

```java
ApplicationModel CarRental {
    DataModel datamodel {
        Entity Customer {
            String customerNo (maxLength=10);
            String lastName (maxLength=20);
            String firstName (maxLength=20);
        }
        Entity Car {
            String make (maxLength=20);
            String model (maxLength=20);
            double rentalPrice;
        }
        Entity RentalAgreement {
            String agreementNo (maxLength=20);
            Date dateFrom();
            Date dateTo();
        }
        Relation AgreementCustomer {
            From RentalAgreement;
            nameFrom = rentalAgreement;
            cardFrom = 1..*;
            to Customer;
            nameTo = customer;
            cardTo = 1..1;
        }
    }
}
```

3.1.1.3  XTend

XTend is a language that allows the extension of a model's capabilities. The extensions are stored in the file Extension.ext. Extensions can be static declarations that can be executed for every component of a model or they are defined for a concrete type of the model. A typical extension for generating a java source file is the creation of the package declaration. The extensions can be used to keep the templates clean from platform dependent information.

```xtend
getPackageDefinition(String package): "package " + package + ";"
```

The example takes a String as input and returns the String package, followed by a package name and a semicolon.

XTend also allows the creation of concrete types of the model. This feature is used to do M2M – model to model transformation. Models can therefore be dynamically extended by new entities. The following expression is used to create a new entity that can be added to the dataModelItems.

```xtend
create Entity addEntity(Entity ent):
    setName("Discount");
```

We prefer to extend our model directly in the model file whenever possible as the model file should be the base information of the complete domain.

3.1.1.3  XPand

XPand is the language that is used by the generator. It allows selective access to all entities and attributes of the model. XPand with its straightforward syntax is quite easy to learn. However it is very powerful to the purpose of processing a given model. To illustrate the use of XPand within a template we will have a look at the
template persistence.xpt and describe some of the features. The following sequence is our entry for the generator template persistence.xpt.

```xml
<DEFINE main(String package) FOR applicationModel>
    <EXPAND businessObject(package)>
        FOREACH dataModel.dataModelItems.typeSelect[rentalId::Entity]>
    <ENDDEFINE>
```

The first line is the signature of the XPand definition. This definition is the entry into the template and routes the processing into another definition called businessObject. The `foreach` keyword indicates that the processing is an iteration over all `dataModelItems` in the `dataModel` that are entities. In our example it is the entities Car, Customer and RentalAgreement. The definition of `businessObject` is described below.

```xml
<DEFINE businessObject(String package) FOR Entity>
    <FILE getClassName(this)>
        <getPackageDefinition(package)>
            /**
             * DO NOT EDIT
             * This file was generated automatically
             */
            @javax.persistence.Entity
            @javax.persistence.Table (name="" + name.toUpperCase() + "")
            <EXPAND classDefinition FOR this> (public static final long serialVersionUID = 1L;)
```

The directive `FILE` instructs the generator to create a new file. By means of the extension `getClassName` the name as well as the path is defined. We have already seen the definition of `getPackageDefinition (String package)` as an extension. After the package definition we jump directly into the class definition. Imports will be done by the beautifier so we don’t care. The extension `classDefinition` will resolve to `public class + Entity name`.

The complete structure of the template shall not be described within this document but we want to show that typical activities can be generated very easy.

```xml
<DEFINE attributeSetter FOR Attributes>
    /**
     * Seter method for property <name>
     */
    public void setName(String toUpperCase) { this.<name> = toUpperCase; }
</DEFINE>
```

The setter method that is very typical for a java file is generated for each attribute of our entity. This approach really guarantees consistency. As our entities are no standalone entities without relation to the rest of the world we need access to the relations that have been defined in our model.

Within the template we have now access to the `relations` through the variable `relations`. As we have defined a relation to keep information about the types that are related as well as the cardinality we are now able to define references from one object to the other.
By means of the cardinality we are able to define the JPA annotations to make the java classes EJB3 conform.

```xml
DEFINE oneToOneFrom FOR Relation
  @javax.persistence.OneToMany(cascade = javax.persistence.CascadeType.ALL)
  @javax.persistence.JoinColumn(name="<to.name.toUpperCase()>_FK")
  private <to.name.toFirstUpper>() <to.name.toFirstLower>();
ENDDEFINE
```

### 3.1.2 AOP

The OAW generation facilities support aspect oriented programming. Aspects can be defined and used in the templates in order to keep them simple and comprehensive. The definition and registration of the advice generator is done within the workflow:

```xml
<component adviceTarget="generator" id="SerializableAdvises"
  class="oaw.xpand2.GeneratorAdvice">
  <advises value="org::rental::aspects::AspectSerializable"/>
</component>
```

The `adviceTarget` is used to refer to the `generator` that was defined in the workflow before. The advices that are weaved into the code are defined in the org.rental.aspect.AspectSerializable. The extensions in the template are used as pointcuts. As an entity bean needs to be serializable the class needs to implement this interface.

```xml
AROUND org::rental::dsl::Persistence::classDefinition FOR Entity
  <targetDef.proceed()>
  implements java.io.Serializable
ENDAROUND
```

When the generator executes the extension `classDefinition` in the file org.rental.dsl.Persistence the advice generator is called instead. The target definition is executed followed by the advice that is defined in the `AROUND`s body.

We have seen the most powerful features that come with OAW and XText. The generator produces the corresponding java files.
The following figure shows an excerpt of a generated class-

```java
package model;

import java.io.Serializable;

/**
 * DO NOT EDIT
 * This file was generated automatically
 **/
@Entity
@Table(name = "CAR")
public class Car implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "SEQ_CAR")
    @SequenceGenerator(name = "SEQ_CAR")
    private Long id;
    @Column(length = 20, name = "make")
    private String make;
    @Column(length = 20, name = "model")
    private String model;
    @Column(length = 0, name = "rentalPrice")
    private double rentalPrice;
}
```

### 3.1.3 The Workflow

The openArchitectureWare framework provides a workflow component that allows a sequential definition of executions around the defined model. It is described in an XML based file with the file extension .oaw.

We will have a look at the workflow that is used to generate the persistence layer of the application. Properties that are used throughout the workflow are defined at the beginning of our workflow.

```xml
<workflow>
    <property name="modelFile" value="model.dsl"/>
    <property name="targetPersistencePackage" value="src-gen/model"/>
</workflow>
```

There are two properties `modelFile` which is initialized with the model source file and `targetPersistencePackage` initialized with the target directory to store the generated sources.

The next component we will have a look at is one of the generated XText artefacts – the parser. The parser is a specialized component for our very own DSL and the model that is defined by its means.

```xml
<component file='org/rental/dsl/parser/Parser.oaw'>
    <modelFile value='${modelFile}'/>
    <outputSlot value='persistenceModel'/>
</component>
```

The parser will transform the input file into a concrete instance of the meta-model and store it in the outputslot ‘persistenceModel’. The outputslot is used to allow information interchange between each component. Therefore the following components have access to the model instance through the ‘persistenceModel’.
Based on the model it is now possible to transform the model into concrete text, i.e. the source code for your target platform.

```xml
<component class='oaw.xpand2.Generator' id="generator">
    <metaModel id='em' class='org.eclipse.mdt.type.emf.EmfRegistryMetaModel'/>  
    <expand value="org:rental:dsl:Persistance:main("model") FOR persistenceModel"/>
    <genPath value="{$targetPersistencePackage}"/>
    <beautifier class="org.hybridlabs.source.formatter.JavaImportBeautifier" organisImport="true" format="true"/>
</component>
```

The above declaration includes the XPand generator into the workflow. The template for the generator is defined in the expand section. It reads the value as command: "Call the template definition main in a file Persistence.xpt that is in a package org.rental.dsl, use the String model as parameter and the persistanceModel as input that is to be processed". We will have a close look at the template in the next chapter. The generator is expanded with a beautifier class. Beautifiers are used to make the source code readable. The beautifier does the indentation that may get lost within the source code generation. The hybridlabs beautifier used in this example is different from the OAW built-in beautifier. It will also extract package names of fully qualifies classes and transform them into import statements in the java file.

### 3.2 ACTIVITIES RELATED TO MDSD

Model Driven Software Development, even with a framework like openArchitectureWare and its components requires activities accompanying its processes. It also requires new skills, roles and responsibilities.

#### 3.2.1 Model Driven Design Activities

Requirements specification is based on more or less formalized models. The increase of formalization is related to the degree of detail discovered in the project. When moving from analysis to design a translation is performed from what needs to be done into how it will be implemented. During this activity different architectural views are designed.

Design establishes all elements the targeted working software will be composed of. They are usually distinguished in terms of components, tiers, layers, packages, frameworks and the like. At the end of the day working software will be composed of artefacts created during project development and others used by but not created in the project. All artefacts are then packaged and deployed. From the appearance of a deployed project it is not obvious whether it was the result of a model driven design or not. So what makes a design model driven?

Recall that the intent was to achieve automation via formalization. Therefore the base for a model driven design is a formalized domain-model. How much of this domain-model is used to actually generate artefacts can provide a measure for a project being "model driven". It represents the degree of automation reached in the development process.

Many projects today use UML as modelling language. Parts of a UML model can be used to generate source code (e.g., class diagram). In that respect UML can be considered a domain specific language in the general domain of writing software.

If you want to use a particular framework, address a certain target platform or cover certain domain specific features for your application but still want to generate code, additional information is needed. To this end from a design point of view you can...
either extend UML (make it domain specific) or invent your own DSL which can be perfectly tailored to your project needs.

In previous CSC projects that we know of following a model driven approach the latter possibility was always chosen as it seemed to provide an easier way to slowly build up an understanding about the DSL, with it the required model information and how generation works.

In order to plan for increasing the degree of automation during design the following major steps must be performed:

1. Find generation candidates,
   - Look for structural or behavioural patterns to find which artefacts can be generated,
   - Typical candidates for code generation are data model related artefacts — such as Java™ Beans or DAOs or Relations — further Business Objects, Validations, Navigations, UI-Handling, Facades, State-Transitions, Forms, Dialogs, Wizards, not to forget Reports and much more.

2. Find the information necessary for generation,
   - Look for attributes or structures which constitute this information,
   - Try to simplify or normalize where possible,
   - Avoid interdependencies,
   - Make every piece of information unique.

3. Find or create a place where to put it in your DSL,
   - Look for suitable location the information fits in,
   - Find comprehensible grammar extensions (wording, symbol) that make sense to the (grammar-)user,
   - Rather put information where it belongs from a business point of view than from the generator’s view,

4. Design generation workflows,
   - Create suitable templates,
   - Use at least one template per artefact type,
   - Create a clear template structure,
   - Define clear responsibilities for every template,
   - Change structure of DSL if templates can be simplified significantly.

These steps can be summarized as finding the right meta-model, DSL and generation mechanisms.

3.2.2 Model Driven Development Activities

After artefacts suitable to be generated have been discovered in a design step implementation of these artefacts is characterized by iteration steps of the following type:

1. Write code,
   - Implement a representative instance of the artefact (boilerplate for template),
   - Build, deploy, test until the desired functionality is achieved,

2. Find and create meta-info,
   - find meta information for that particular artefact instance, enhance the domain
model with this information, if necessary also enhance the meta-model in case it has not been specified to all needs during design,

3. Write template, write structured template(s) that will generate the same code written in step 1, use only information from the domain model,

4. Generate, generate using template written and domain model, repeat previous steps if necessary until the desired functionality is achieved just like for the code written in step 1,

5. Test, test generated code, repeat steps 2 and 4 to create other instances of the same type of artefact, build, deploy, and test until the desired functionality is achieved.

3.3 DEVELOPMENT PROCESSES CHANGE

Iterations during model driven software development are outlined in Figure 4. On the left hand side it roughly sketches the results of a design step.

Figure 4. MDSD Iterations

In this step artefact candidates are discovered and an appropriate DSL is created specifying syntax rules for domain modelling. For all artefact candidates templates and generation mechanisms should be implemented during development. To do this they are picked from this list piece by piece.

If no generation mechanism is available for the artefact picked - for instance from previous projects - it will have to be written from scratch. This is done in iteration step 1 through 5 (IS 1 to IS 5) as described above. Steps 1 and 3 are marked in bold letters since they are more elaborate and time consuming than the other steps. The dotted area marked MDSD denotes elements specific to model driven development that would not be found in a conventional approach.

One should notice the development cycles marked yellow in Figure 4. They are characterized by the type of activity performed, their frequency and effort.

1. Development cycle: The activity in this cycle is usually writing code. Writing code in this context is not restricted to Java™, but comprises all target languages reflected by the artefact candidates. It is important for this step to pick a
representative instance of the artefact concerned to cover the majority or even all aspects for later code generation. The code must be written as if it was for the working software as it is base for the template written in the next cycle. The template in return constitutes the quality of the generated code. This cycle should be performed only once per type of artefact candidate. It is performed until the picked representative instance works as desired.

2. Template Cycle: The activity in this cycle is writing templates. Code written and tested in the previous cycle is scanned for structural and repetitive patterns with respect to the available meta-information. Writing templates can be considered rewriting the artefact in terms of DSL. This cycle should be performed only once per type of artefact candidate. Template development is finished when the generation mechanism using the template produces the same code as would have been handwritten.

3. Completion Cycle: The activity in this cycle is characterized by filling the domain-model. When templates, generation mechanism and DSL are in synch and produce the desired artefact results for representative artefact candidates, all that remains is completing domain-model information. This cycle should be performed for each instance of the type of artefact candidate. It is finished when code generation works for all instances of the concerned artefact type.

In practice one will often discover the need to extend or modify the DSL developed in a design step. This is illustrated by the “revisit” arrow in Figure 4. The number of revisit cycles usually decreases during project development.

3.3.1 Responsibilities and Roles
In software development projects certain skills, roles and activities are required for a successful achievement of the project’s objectives. For example the architect creates the software architecture and model, the developer implements the solution according to the architecture, its model and the business requirements. MDSD focuses on the meta-model, the model and the model transformation; it requires additional responsibilities and skills. The project team has to define the responsibilities for the specification of the meta-model, the definition of the appropriate transformations or the checking of the consistency between the models. These responsibilities and skills can be divided in 2 groups: the “meta” and the “project” skills and responsibilities groups.

3.3.1.1 The “Meta” skills and responsibilities.

3.3.1.1.1 Domain Knowledge
The domain knowledge is a detailed understanding of the application domain. It includes ability to abstract and categorize concepts and relationships in the application domain. It provides the base knowledge needed for the creation of the meta-model.

3.3.1.1.2 Platform Know-how
Creation and transformation of a model is impossible without a platform specific model (PSM). The Platform Know-how skill consists of a detailed know-how about the platform’s components, configurations and modules. It provides the knowledge necessary for creating a platform specific model and contributes to the architecture and application model and its transformation.

3.3.1.1.3 Language Engineering
A meta-model has to be created using one or several domain specific languages (DSL). The team member responsible for the “language engineering” creates these
languages and the meta-model by means of a meta-modelling framework like EMF Ecore or openArchitectureWare XText.

### 3.3.1.4 Transformation Specification

The responsibility for the transformation specification is vital and required by MDSD. The Transformation specification defines the relationship between domain and platform model. It requires a detailed know-how about the transformation language (e.g., openArchitectureWare XTend and XPand2) and the technologies and frameworks used by the generated artefacts.

### 3.3.1.5 Method Engineering

The “method engineer” identifies and manages the activities needed by the project. He identifies the artefacts that have to be produced and defines the development process.

---

**Figure 5. “meta” Skills in the MDSD Development Process**

#### 3.3.1.2 “Project” Skills and Responsibilities

**3.3.1.2.1 Application Design**

The application design includes the architecture design, coding and testing. The architecture design is the creation of the application model using the meta-model language created. The biggest part of the coding is realised through coding of templates as input for the generators (transformation), while the manual coded part has to respect and follow the conventions and specifications defined for the model and the generated code. While some test units can be generated, a lot of them still have to be written manually.
3.3.1.2.2 Tester
The tests do not only matter the generated or manually written code, but also the model and meta-model. Model tests can be generated on the same way as the source code. Model testing can be supported by Model simulation, a feature provided by some MDSD Tools.

In the majority of cases it is not possible to map each responsibility to exactly one project team member. In most cases each member will have several MDSD related responsibilities according to the existing skills.

3.4 MDSD AND CATALYST
The precedent chapter handles the impact of MDSD on standard software development methods, cycle and requirements. But we have another interesting case that we should look at: MDSD and Catalyst.

Catalyst is CSC’s methodology that defines processes, techniques and roles to provide solutions to business problems and increase the satisfaction of the customer. It has defined processes, methods and roles for the most known architecture and development standards, but at this time there is no process defined for model driven software development. The matter of this chapter is not an introduction to Catalyst, but explains how MDSD could be integrated in it and why it
should be integrated. So the reader should have or acquire a basic knowledge of Catalyst.

Model driven software development meets most Catalyst requirements:

- Ability to adapt to new technologies: this is one of the basic principles of MDSD. The use of a platform independent model (PIM) keeps you free to choose any technology.

- Scalability: it is easier to reach higher scalability by using models and generators because the changes or extensions of an application are made at central places in the used MDSD framework.

- Maintenance: as a basic principle of MDSD the maintenance principally focuses on the meta-model, model, generators or interpreters.

- Reusability: nowadays most MDSD tools allow reusing the meta-model languages or grammar (DSL) and the transformation components (generators or interpreters). They can be used as cartridges that can be put together depending on the business needs to generate an application.

3.4.1 MDSD and Catalyst Guiding Principle

Model driven software development supports the Catalyst Guiding principles. It provides a simple language that can be used in the communication with the customer; it keeps all parts of the solution development consistent and is able to guarantee a high quality of the results.

3.4.1.1 Deliver Results That Satisfy Customer Priorities

Several aspects of MDSD, like the Domain Specific language, allow involving the customer in each phase of the project according to the Catalyst methods. Customer and projects can use a common language; working on meta-model and model level accelerates the realisation of requirement changes.

3.4.1.2 Reuse Knowledge to Solve a Specific Business Need

The meta-model in MDSD is based on the Enterprise, the Business and the System Architecture. It defines a language to model the solution for the enterprise and business requirements. This language or at least some of its elements and the transformation components used to generate the solution over the model created with the language can be abstracted and reused for similar business needs in following projects.

3.4.1.3 Guide Solutions with an Architecture

The principle of horizontal and vertical consistency of MDSD supports this Catalyst principle. Changes always have to be integrated at the architecture or rather at the
model level. Change requirements are met at all levels of the solution development, then the solution is in its biggest part generated and so automatically consistent to the model and the business needs.

3.4.1.4 Achieve Operational and Quality Results through Process Improvement and Innovation

With MDSD what is generated and runs successful once will always run without errors in its own context. Since the generator of each artefact will be written once, the quality of the generated code can be increased compared to code written manually within standard development methods where lack of time can affect the quality of the code. MDSD is not dependent of the technology it generates; it can integrate the newest frameworks and languages. MDSD is able to ensure high quality, process improvement and innovation.

3.4.2 MDSD and the Catalyst Framework Concept

MDSD can be fully integrated in the Catalyst Framework Concept. The Business processes, Data and technology parts of the Domain of Change, the Catalyst Model Views and Units of Scope represent the base used by the MDSD meta-model. The MDSD Generators generate the Application using the model and a certain technology.
The MDSD development processes can be used in the Catalyst Lifecycle Framework and provides Work, Roles and techniques to achieve the goal of the Catalyst Frameworks and methods.

### 3.4.3 MDSD in the Catalyst Lifecycle

#### 3.4.3.1 Vision and Strategy

<table>
<thead>
<tr>
<th>Vision and Strategy</th>
<th>Process Enactment</th>
<th>Organizational Change</th>
<th>Technical Infrastructure</th>
<th>IT Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise Transformation Planning</td>
<td>C-REF</td>
<td>XAD</td>
<td>PMD</td>
<td>ICP</td>
</tr>
<tr>
<td>Information Technology Strategy</td>
<td>GCC</td>
<td>OCM</td>
<td>SEC</td>
<td>C-REF</td>
</tr>
</tbody>
</table>

Depending on the business requirements and needs of the customer, MDSD provides the features for a flexible, scalable and innovative Information Technology Strategy, always considering the Enterprise Transformation Planning, through its iterative development of meta-model and model and the generating of source code.
3.4.3.2 Architecture

The MDSD meta-model or Domain Specific Language is based on the Enterprise and the Business Area Architecture and produces the Solution Architecture. The Enterprise Architecture provides all information necessary to create the meta-model and model (Data, Business processes, and Technology).
3.4.3.3 Development

The MDSD Development Phase essentially covers the development of generators and templates. Those components allow the transformation of model information into platform specific artefacts. MDSD is not dependent on the technology used and generated; it is able to produce the work products according to the existing standard development method (for example Package Based Development).

3.4.3.4 Integration and Deployment

Through the flexibility provided by the features of MDSD, components for system integration can be defined and built in the meta-model and model to be generated during the model transformation. Mentioned in a chapter above, the test phase of a MDSD project focuses on the model rather than on the source code.

Additionally MDSD tools and frameworks provide interfaces and components to make the integration in build, configuration and deployment management as easy as possible. For example the openArchitectureWare framework offers Ant Tasks and Maven plugins.

3.5 THE REMAINING 20%

MDSD is very often faced with allegations on its limited potential to deal with complex business requirements. Base for MDSD is its model and a model can only be as exact as its meta-model allows specifying it. Rules, transformations, validations and transactions can be modelled somehow for sure but one should wonder if this approach is helpful. MDSD shall not end in itself and does not claim to bring the solution for every problem. But it can separate technical details from the
domain of interest. It reduces the number of error sources. It reduces the code that has to be written manually and it increases the possibility the reuse your components.

The following listing shows a modelling approach for adding domain related functionality into the model.

```java
BusinessModel:
  "BusinessModel" name="ID" {
    (components+=Component)
  };

BusinessMethod:
  "BusinessMethod" name="NAME ";
  (parameters+=Entity)
  (returnType=Entity)
};

Component:
  "Component" name="NAME ";
  (businessMethods+=BusinessMethod)
};
```

The business model is composed of several components where each component can provide several domain methods.

The model can easily be enhanced by adding components that have a specialized behaviour. For instance, one can model an Availability Check that is able to check Availability for a given Car at a given Date and returns true or false as response. Components would then interact on the base of Interfaces and via configuration the runtime behaviour of the application is dedicated through manually written implementations.

Components of the application are defined as interfaces. In the interfaces solely the signature of a method is defined. The information contained by the signature is kept in the model. Therefore only the implementation of the concrete method has to be done to complete the business process. This concept separates the model specific information from the concrete implementation. It also allows the interaction of manually written code with other code that has been generated.
4 SUMMARY

Chapter 2 has given an overview of the general MDSD concepts. Chapter 3 describes a technical and theoretical view on the MDSD frameworks, activities, roles and required skills. It describes also the impact of MDSD on standard and CSC development processes (Catalyst). This chapter gives you an overview of our experience with MDSD, best practices and anticipates its probable future.

4.1 EXPERIENCES

This paper is based on project experience with MDSD over a couple of years. We have evaluated a lot of methods and frameworks. We have tried to get the most out of MDSD, this means to generate as much as possible code and artefacts for a system or an application. Some MDSD tools and frameworks did not meet our requirements. Other ones fulfilled nearly all requirements. At the end we have kept Eclipse, EMF and openArchitectureWare to operate MDSD. In our opinion is it the most flexible and powerful tool and framework combination.

Our evaluations confirm that using Model Driven Software Development lets us develop software faster and more efficiently. MDSD reduces the costs of development through its flexibility, scalability and the reusability of its features, components and products.

4.2 BEST PRACTICES

4.2.1 Separate Generated and Manually Written Code

Define a clear border between the generated and the manually written code. Generated code should not be manipulated by any developer (using of abstract classes). Use protected regions in the generated code if manual changes in it cannot be avoided.

4.2.2 Do Not Check in Generated Code

As long as there is no change in the meta-model, the model or the generators, the generated code will not change. These version control resources are useless. Check the meta-model, model and templates for the generators instead.

4.2.3 Integrate the Generator In the Build Process

The automation of the build process (Ant, Maven) increases the productivity of the development Team. Add a generation phase using the generator placed before the compile phase. Use tools and frameworks which provide interfaces or components for the integration of the build process.

4.2.4 Generate Clean Code

In the templates a generated class is written only once. So take the time to create high quality code. The code may be read at any time.

4.2.5 Talk “meta”

Use the terms and elements defined in the meta-model to communicate within the project team, in other words use the DSL as a common language not only by developing but also while discussing aspects, concepts of the application. This will help to keep the meta-model meeting the requirements.

4.2.6 Develop Iteratively

DSLs have to be developed iteratively. They grow with the understanding of the domain they describe. Elements can be added or removed. Always check if any change needed can be modelled in the meta-model.
4.2.7 Use a Reference Model
Use a reference model while developing the DSL. The generation of code from this reference model has to be a part of iteration in the development of the DSL. This way the vertical and horizontal consistency of the project is kept.

4.2.8 Use a Simple Syntax
One of the goals of MDSD is simplicity. Keep your DSL simple but comprehensive. Model only the necessary, do not model and define elements that will be anyway generated from a simple term. For example if you model a persistent object you do not have to model its DAO interface which can be generated from the persistent object model, if necessary only add the needed properties to the model element.

4.2.9 Use Model-to-Model Transformation
Use model-to-model transformation to reduce the complexity of the generators. Let the generator transform the “interim” models. This also increases the scalability and reusability of the MDSD components.

4.2.10 Do Not Reinvent, Use Existing Frameworks
Do not generate new frameworks, use existing and improved ones. For example do not create a new persistence framework while developing a meta-model for a persistence domain, generate JPA, Hibernate or Ibatis code. For almost each development domain one or more framework exists. Take the time to look for them before writing the DSL and the generators.

4.3 PRO & CONTRA
MDSD is not the magic solution for every problem. Let us have a look at the benefits:

- Formal way to specify domain specific know how. Domain experts always have a way to communicate their knowledge but in a technical not formal way. Domain models can be processed by machines.

- Separation of technical details from domain specific knowledge. There is always the danger to annoy domain experts with technical implementation details. MDSD provides means that let you concentrate on domain specific information.

- Reusability of DSLs, models and templates: While single components of an application can hardly be reused as they are usually part of a composition MDSD components are specialized for a model view. If the model is appropriate for reuse the components will be as well.

- Vertical and horizontal consistency: Templates produce vertical consistency as either all or no component will be correct. The usage of one model concentrating all domain specific knowledge allows the usage of this single information in all layers of the application. This is known as vertical consistency.

- Domain model will always be constant independently from platform: The domain model has no dependencies to the concrete implementation or platform. This fact allows an easy change of technologies.

- Common communication platform for technicians and domain experts: Speaking one language is not only in software development the essence of communication. Ambiguity is avoided to a high level.
• Efficiency by generation of repetitive code: The generation of repetitive code structures is only implemented once. Changes in the template have effect on all artefacts.

• Agile development process: All domain specific knowledge is kept in one place. Changes in the model directly lead to different behaviour throughout the application.

• Avoidance of accidental errors: Either all or no artefact is correct. Developer tests can concentrate on one representative region of the application.

On the other side some problems or challenges exist:

• Higher initial investment: DSL(s) have to be developed. Generator templates need to be implemented. Analysis for reusable artefacts and redesign of DSL may become necessary until the breakeven point is reached.

• Additional training necessary: It is impossible to live MDSD at a moment’s notice. It is more complex to generate a source file than write it in 3GL. Abstraction to a meta-model that allows the precise coverage of a domain is not easy.

4.4 THE FUTURE

MDSD always oscillated between temporal zeniths and low adherence in the developer community. Some companies focus on MDSD as their strategy as minted by individuals that are convinced by the strength of a model driven approach. People with an affinity to abstraction of model information are the precondition for a successful usage of MDSD. Imposing the paradigm of thinking “meta” will not be successful without convincing the team and demonstrating the possibilities. High fluctuations from one project to the next one impede the setup of a constructive MDSD structure.

It is also a crucial fact that the factual coding activities are mostly done by younger employees who need to get used to the companies processes and structures. The first years in their professional life is affected by learning tools, programming languages and communication skills. They take a leaf out of their more experienced colleague’s books.

The vast number of available frameworks for each technical platform and concern is also an important factor to let MDSD seem unnecessary. They suggest a feeling that knowing the framework will solve all problems but knowing the framework is just the precondition not to run into traps. MDSD shall not substitute existing frameworks but separate different concerns — technical from business.

Today MDSD is made rather easy as there are lots of open source tools, books and documentations on that concept. The developer community is consequently enhancing tools for creation, transformation and processing models in a straightforward way. We see a very big potential in the Eclipse Modelling Framework in combination with openArchitectureWare. Models can be transformed from several sources like Java™Beans or UML2 models into an instance of the Ecore meta-model. Transformation and processing of the model as well as source code generation can easily be done by means of OAW.

The preconditions are fulfilled. Let’s think Meta.
5 APPENDIX

5.1 ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>ASP</td>
<td>Active Server Pages</td>
</tr>
<tr>
<td>CWM</td>
<td>Common Warehouse Metamodel</td>
</tr>
<tr>
<td>DAO</td>
<td>Data Access Object</td>
</tr>
<tr>
<td>DDL</td>
<td>Data Definition Language</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain specific language</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Beans</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>I18N</td>
<td>Internationalization</td>
</tr>
<tr>
<td>JPA</td>
<td>Java Persistence API</td>
</tr>
<tr>
<td>JSF</td>
<td>Java Server Faces</td>
</tr>
<tr>
<td>JSP</td>
<td>Java Server Pages</td>
</tr>
<tr>
<td>M2M</td>
<td>Model to Model</td>
</tr>
<tr>
<td>M2T</td>
<td>Model to Text</td>
</tr>
<tr>
<td>MDSD</td>
<td>Model Driven Software Development</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>OAW</td>
<td>openArchitectureWare</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
</tbody>
</table>

5.2 LINKS

Homepage Eclipse Modeling Framework:
http://www.eclipse.org/modeling/emf/

XText reference:

Homepage openArchitectureWare:
http://www.openarchitectureware.org/

Homepage Eclipse:
http://www.eclipse.org/

Homepage GenFw:
http://genfw.berlios.de/

5.3 BOOKS


Eclipse Modeling Project, A Domain-Specific Language (DSL) Toolkit [Richard C. Gronback]
DISCLAIMER

The information, views and opinions expressed in this paper constitute solely the authors’ views and opinions and do not represent in any way CSC’s official corporate views and opinions. The authors have made every attempt to ensure that the information contained in this paper has been obtained from reliable sources. CSC is not responsible for any errors or omissions or for the results obtained from the use of this information. All information in this paper is provided “as is,” with no guarantee by CSC of completeness, accuracy, timeliness or the results obtained from the use of this information, and without warranty of any kind, express or implied, including but not limited to warranties of performance, merchantability and fitness for a particular purpose.

In no event will CSC, its related partnerships or corporations, or the partners, agents or employees thereof be liable to you or anyone else for any decision made or action taken in reliance on the information in this paper or for any consequential, special or similar damages, even if advised of the possibility of such damages.

Copyright © 2010 Computer Sciences Corporation. All rights reserved.