CHANGEABILITY IN MODEL DRIVEN WEB DEVELOPMENT

THESIS

SUBMITTED IN FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

SOFTWARE ENGINEERING

BY

DAVID VAN DIJK

BORN IN IJSSELSTEIN, THE NETHERLANDS

FINAL 06-08-2009

STATUS OF PUBLICATION: PUBLIC

Institution:
University of Amsterdam
Faculty of Science
1098 SM Amsterdam
The Netherlands
www.science.uva.nl

Company:
Capgemini BAS
Staalmeesterlaan 410
1057 PH Amsterdam
The Netherlands
www.capgemini.com

ABSTRACT

This research project is conducted to find out whether the changeability of a Model Driven Web Development approach can be competitive to a classical software development approach. In the project the changeability of both kinds of approaches is analyzed and compared.

The domain is scoped to small e-commerce web applications build on an ASP.NET architecture. For the development environment of the classical approach standard Microsoft tooling is applied. The model driven approach exploits the use of textual DSLs and uses the openArchitectureWare tooling for its environment.

The changeability of the approaches is analyzed as follows. In both approaches an application based on the Java Pet Store is constructed; this is used as a baseline. A number of changes that form a representative early evolution are applied to both environments. The Maintainability Index and the effort in time per change metrics are used to determine complexity growth and cost of changes. Based on the measurements a quantitative analysis is performed. The results of the previous are brought into perspective by a qualitative analysis in the form of a technical discussion on the maintainability of both approaches conducted by the developer.

The analyses show the model driven approach to be competitive to the classical software development approach in early evolution. Other outcomes are that complexity grows far steeper in the classical environment and that overall cost for changes lower in time in the model driven approach because the domain coverage becomes higher during development. During evolution the difference in changeability between the approaches will become larger, in favor of the model driven approach.

During the project modularity turned out to be of major influence on the changeability of the code. Lack of modularization will make files grow large leading to maintainability and scalability problems. The openArchitectureWare tooling lacks support for modularization of grammars and DSLs at the moment, which had a large effect on the experiment's results. If supported the Maintainability Index measurements would have been in favor for the model driven approach from the start and the approach would provide a better fit to domains dealing with larger applications.

ACKNOWLEDGMENTS

I would like to thank Jurgen Vinju for his guidance and support. Our conversations were motivating and they provided me with useful insights. Jurgen helped me in finding my own path through the project. I would also like to thank Hans Dekkers for the help in starting up this project, for reviewing my writings and his general support. And last but not least I would like to thank Pieter Faber and Jan Willem van Veen. They made it possible for me to work on the subject of Model Driven Software Development within Capgemini BAS. I thank them for their efforts in providing the necessary means for the project and for the interesting talks.

LIST OF ABBREVIATIONS

Abbreviation	Meaning
AOP	Aspect oriented programming
ASP	Active Server Pages
ATL	Atlas Transformation Language
CIM	Computational Independent Model
CMOF	Complete MOF
CSS	Cascading Style Sheets
DSL	Domain Specific Language
EBNF	Extended Backus Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
EMF	Eclipse modeling Framework
GMF	Graphical Modeling Framework
GP	Generative Programming
GPL	General Purpose Language
HTML	Hyper Text Markup Language
M2T	Model to text
M2M	Model to model
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MDWE	Model Driven Web Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OAW	openArchitectureWare
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
SQL	Structured Query Language
UML	Unified Modeling Language
UWE	Uml based web engineering
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

LIST OF FIGURES

LIST OF CODE FRAGMENTS

LIST OF TABLES

TABLE OF CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

Software needs to change in order to stay useful. This leads to a growth of complexity in its structure unless action is taken to prevent this [Lehman2001]. The complex nature of software makes it hard to manage change and keep of deterioration. The introduction of high level languages proved to be a major step in this area and Model Driven Software Development might well be the next.

Model driven software development drives on abstraction and automation. It makes it possible to simplify the software's complexity by separating its concerns into high level domain specific abstractions. The system is described by these abstractions, making it possible to infer the implementation from them or execute the abstractions directly. The approach delivers high changeability for all that can be described in its abstractions.

Model driven software development reduces complexity in the regular evolution dimension, but also adds complexity by introducing multiple dimensions of evolution [Deursen2007]. In the end, will the total complexity be reduced and will higher changeability be delivered? This project looks for an answer to this question in the domain of web engineering.

1.2 SCOPE

FOCUS AREA

At the Microsoft department of Capgemini BAS there is a need to raise productivity in the development of small web applications. Small web applications are project with less than 250 function points, based on an ASP.Net architecture. Most often the applications are transactional web applications, allowing a user to interact with and update data that resides in a database. These kinds of projects have a short time to market and a high rate of changes.

For large projects Model Driven Software Development (MDSD) is applied, delivering a substantial productivity gain. MDSD aims at improving productivity by raising the level of abstraction. Abstraction simplifies management of complexity and creates opportunities for automation. Based on the experience with applying MDSD to large projects the organization believes that applying MDSD to small web application projects will lead to the desired productivity gain, and will deliver desired side effects like improvement of software and process quality.

The MDSD approach used on the large projects doesn't fit well on the small web applications projects. The approach focuses on changeability on the determined functional domain. In the domain of small web applications the functional domain often is unfamiliar, small or unstable. Compared to large projects there is a higher percentage of technical changes. Therefore small web application projects require a more broad changeability in the MDSD approach to enable a substantial productivity gain.

Also, the current MDSD approach applies a waterfall oriented process which only supports evolution within the determined functional domain. A waterfall approach doesn't support the controlled evolution needed for the development of small web application projects and the co-evolution of their model driven development approach. The Rational Unified Process (RUP) is currently used as the develop process methodology for small web applications. The iterative and incremental development provides fast feedback on changes and reduces risks. To apply MDSD successfully on small web application projects it is must support iterative and incremental development.

OBJECTIVE

The MDSD approach for small web applications must provide high changeability in order to bring a substantial productivity gain. The project's main objective is to find out if this is feasible. Knowledge on changeability in model driven software development for small web applications will be retrieved. Knowledge will be delivered on the subject that will provide insight in:

- What changeability of MDSD in web development beholds
- How a MDSD environment for the domain can be created to provide this changeability
- How the changeability of a MDSD approach relates to a classical approach

1.3 RESEARCH QUESTIONS

As stated in the objective the MDSD approach must provide high changeability. To be feasible the approach must be at least competitive to the classical approach. Therefore the main question of this project is:

RQ1. Can MDSD supply the needed* changeability to be applied to the domain of small web applications?

* Needed means at least competitive to a classical approach.

The answer to the main question will be based on how the changeability of a MDSD approach compares to a classical approach. To come to this comparison the following sub questions are defined:

RQ1.1. How does the measurable changeability of the MDSD approach relate to that of the classical approach?

RQ1.2. How is the changeability of MDSD approach perceived by a developer in comparison to the classical approach?

The expected result of the project is:

An assessment of the changeability of a model driven web development approach.

1.4 CONTRIBUTIONS

The contributions of this thesis are as follows:

- A design and prototype implementation of a MDSD approach for web engineering applications
- An analysis of the changeability of MDSD software as compared to traditional programming language software in the web domain

1.5 RESEARCH APPROACH

The main question will be answered by comparing the changeability of a classical approach to that of a model driven approach.

The projects main component is an empirical case study. This is preceded by a literature survey, conducted to acquire information about how to setup the experiment and how to assess the research environments. Based on the main research question and the literature study hypotheses will be formed. The results of the measurements in the case study and developer's review will be used to test these hypotheses.

The experiment will contain two development environments, a model driven and a classical one. Both will be used to develop the same simple web application, based on the "Java Petstore" [Petstore]. To enlarge external validity the environments are build as representative as possible.

On both environments changes will be applied, showing an expected evolution. Metrics will be used to analyze the changeability of the environment, and form the base of the quantitative analysis.

The metrics are methods used without discussion and their validity is not part of the study; this is not a study about metrics. The maintainability index (MI) is used to determine complexity; time needed per change is used to measure effort.

A qualitative analysis will be performed by conducting a developer's review on the technical maintainability of both approached.

The results of the project will be analyzed, leading to the conclusion of the work.

1.6 THESIS OVERVIEW

The remainder of this thesis is as follows:

Chapter 2	provides background information on MDSD and Web Engineering.
Chapter 3	formulates the hypotheses to be tested.
Chapter 4	describes the design of the environments, the metrics used in the experiment and the changes that are applied.
Chapter 5	beholds the quantitative analysis. It shows the results and analysis of the measurements, and the tested hypotheses.
Chapter 6	beholds the qualitative analysis. It gives a validation of the results based on a developers review.
Chapter 7	shows what threats there are to the validity of the projects results.
Chapter 8	Described the conclusions of the project.

2 LITERATURE SURVEY

In section 2.1 a short introduction into the fields of Model Driven Software Development (MDSD) is presented. The general idea behind MDSD is explained as well as related terms that are of importance to this project. The related terms are models, DSLs and the evolution of a MDSD approach. The section is based on the text in appendix D, which contains a more elaborate introduction into the field of MDSD.

Section 2.2 defines the scope of the Small Web Application Domain and describes the concerns to be addressed.

Section 2.3 shows how the domain can be combined with a MDSD approach.

In section 2.4 described some on the relation between software engineering and managing complexity

Section 2.5 defines changeability, and provides insight on how to assess changeability in the research environments.

To be able to answer the main question changeability of a MDSD and a classical approach needs to be compared. To do so it must be known what changeability in the approaches means, how it can be measured and how a high changeability can be provided. The questions below were therefore defined to be answered by the literature study.

- **How can changeability of both approaches be defined?**
- **How can changeability of both approaches be assessed?**
- **How can changeability of both approaches be optimized?**

Section 2.6 summarizes the answers found.

2.1 MODEL DRIVEN SOFTWARE DEVELOPMENT

Model Driven Software Development (MDSD) is a software development approach in which abstract models are the primary artifacts, instead of the code in a classical approach. The approach makes it possible to focus on the essence of the system and minimize the accidental complexity [Brooks1987] talks about.

The main purpose of MDSD is to improve productivity by reducing development effort. MDSD offers simplification by abstraction and automation to realize this. Abstraction takes the form of models; automation takes the form of transformations.

Models define what is variable in a system, and code generators produce the functionality that is common in the application domain [Deursen2007]. Besides productivity gain, also development costs and time to market can be reduced, and quality can be improved. The approach basically tries to offer a better way to deal with complexity.

MODELS

A models representation can be graphical, textual or a combination of both. Source code can also be seen as a **model** [Mens2005][Kleppe2008] [Favre2004]. Kleppe introduces the term “Mogram” to describe a product written in a software language. As programming or modeling languages are described by their grammar/**metamodel**, these also can be seen as models [Favre2004]. [Kleppe2008] states that a metamodel specifies a modeling language, and is usually the model of an abstract syntax. The language used to describe the grammar/metamodel is called the **metametamodel**. This language can describe itself.

The layering of these models, based on the four-layered architecture of the MOF standard from the OMG, is depicted in figure 1. It shows a mega-model, a term [Bezivin2005] and [Favre2004] introduced to describe the model that uses concepts as model, metamodel and metametamodel.

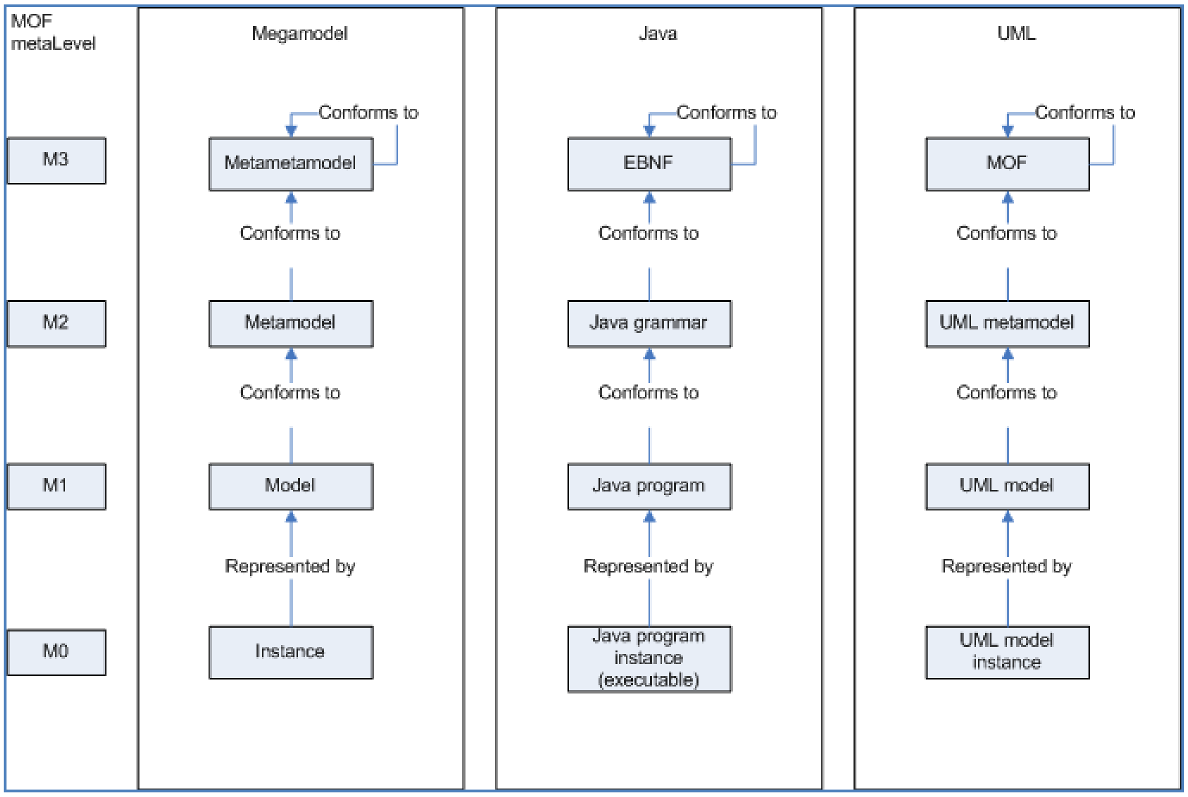


Figure 1 – Metamodeling architecture

DOMAIN SPECIFIC LANGUAGES

A modeling language can take the form of a General Purpose Modeling Language as the Unified Modeling Language (UML). But because models are used to describe separated and well scoped concerns Domain Specific Languages (DSLs) can exploited, offering more expressive power.

Domain specific language (DSL)

A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

[Deursen2000]

DSLs can be concise, enable reuse, can enhance productivity, reliability and maintenance, and improve communication with domain experts. On the other side the cost of designing, implementing and maintaining a DSL and educating the users can be high, especially because of the smaller user group compared to a GPL [Mernik2005] [Deursen2000][Deursen1997].

EVOLUTION

While MDSD simplifies the "coding" of the system, complexity is added through the introduction of a multi dimensional evolution. [Deursen2007] defines the dimensions as follows:

*In **regular evolution**, the modeling language is used to make the changes. In **metamodel evolution**, changes are required to the modeling notation. In **platform evolution**, the code generators and application framework change to reflect new requirements on the target platform. Finally, in **abstraction evolution**, new modeling languages are added to the set of (modeling) languages to reflect increased understanding of a technical or business domain.*

The "coding" of the system takes place in regular evolution, where changes are cheap because of the simplification. But the costs for this simplification are paid in the other evolution dimensions where changes are more expensive. If for example an unstable domain or platform causes a lot of changes in the metamodel or platform evolution dimension the costs of the approach are likely to rise above the costs of a classical approach. In order for the MDSD approach to be viable certain conditions need to be satisfied, e.g. the domain addressed needs to be well known and reasonably stable.

2.2 SMALL WEB APPLICATION DOMAIN

DOMAIN SCOPING

At the Microsoft department of Capgemini BAS there is a need to raise productivity in the development of small web applications. Small web applications are project with less than 250 function points, based on an ASP.Net architecture. Most often the applications are transactional web applications, allowing a user to interact with and update data that resides in a database. These kinds of projects have a short time to market and a high rate of changes. The project is set to address this domain of what will further be addressed as the Small Web Application Domain.

The domain is scoped by its focus on web applications. Applications need to conform to the definition of a web application. The domain is also scoped by the size of the application. The maximum size of an application is set to 250 function points. Furthermore the application is build on an ASP.Net architecture. This narrows the scope in terms of technologies and languages. For example languages as PHP and Java and related frameworks as Struts and Swing are out of scope.

Because the applications in the domain most often are transactional web applications, using the web application categorization of [Kappel2006], this is also used to scope the domain. Transactional web applications are web applications that not only provide read-only interacting to the user, but also allow the user to perform updates to the underlying content, which most often resides in a database. Online banking and shopping applications belong to this category.

WEB APPLICATION

> **Web Application**
>
> A web application as a software system based on technologies and standards of the World Wide Web Consortium [W3C] that provides web specific resources such as content and services through a user interface, the web browser.
>
> [Kappel2006]

Web applications are build on the infrastructure of the World Wide Web and therefore utilize web specific technologies, standards and languages. These include technologies and standards of the W3C as HTTP, HTML, CSS and XML. They also include web specific technologies and languages as Ajax and JavaScript. Because of the use of a web browser as user interface web applications are platform independent. The aforementioned specifics of web applications have a large impact on the design and architecture of web applications. For example the use of a browser implies a “thin client” architecture, and the use of HTML restricts the possible navigational structures.

CONCERNS

Concerns to be addressed in the domain are those common to web applications and those common to transactional applications.

Many web engineering methodologies exist, e.g. Hera, WebML or UWE [Wimmer2007][Kappel2006]. The methodologies all have a different domain focus and address concerns of the domain in their own way. Looking at the concerns addressed by these methodologies the fundamental concerns of web applications can be distilled.

These fundamental concerns are [Fraternali1999][Kappel2006]:

•	**Content**	the information and application logics underneath the Web application, e.g. entities and relationships
•	**Navigation**	the structuring of the content into nodes and links between these nodes, e.g. links between pages
•	**Presentation**	the user interface or page layout, e.g. objects in the user interface

[Kappel2006] characterizes web application evolution as driven by continuous change, competitive pressure and fast pace. According to this statement the need for high changeability is a common concern. Other common concerns are cross-cutting concerns as security, logging, and exception management.

The field of transactional web applications adds the need for a way to persist data. Also performance is a concern. But most important is the business logic that needs to be addressed.

In table 1 the concerns are mapped to the languages that address them in a classical ASP.Net architecture.

Table 1 - Domain concerns and classical languages

Concern	Language
Content (Information)	SQL
Content (application logic)	C#
Navigation	HTML/C#
Presentation (layout)	HTML
Presentation (style)	CSS
Business logic	C#/SQL
Security, logging, and exception management	C#

Concerns in the form of system quality characteristics as changeability and performance are addressed by all of the languages.

Beside C# all the languages mentioned above are DSLs.

2.3 MODEL DRIVEN WEB DEVELOPMENT

The technical domain of web engineering has been recognized for some time now as suitable for a MDSD approach. The combination of the two has been named Model Driven Web Development (MDWD). [Moreno2008] states that MDSD can be successfully applied to this domain due to the fact that there is a precise set of concerns that need to be addressed, that the basic kinds of applications is well known and the architectural patterns and structural features used in web systems is reduced and precisely defined. The previous section describes this precise set of concerns that need to be addressed, underlining this statement.

Current MDWD methodologies are tied to particular architectural styles and technologies, deal with a fixed set of concerns, and often use proprietary notations and tools. Most of these MDWD methodologies use graphical modeling. An example of a graphical approach can be found in [Kraus2007]. He describes a MDA oriented way of model driven web engineering using [UWE]. A textual approach that could be placed into the MDWE category is WebDSL, a domain specific language scoped on interactive dynamic web applications with a rich application-specific data model [Visser2008].

Both approaches are scoped within the web application domain using abstraction and automation, but have little in common in their implementation. Therefore the evolution of the two also differs a lot. In comparing the changeability of the two aforementioned approaches the approach of [Kraus2007] is likely to lose. The approach addresses more concerns, accommodates manual refinements on multiple abstraction levels and uses standard which introduce overhead and reduce flexibility.

The approach that will be taken in this project will lean more toward the approach of [Visser2008]. Essential differences still remain between the approaches, e.g. in the technologies and tooling used.

2.4 SOFTWARE ENGINEERING

At the first NATO Software Engineering conference in 1968 Fritz Bauer coined the terms “The Software Crisis” and “Software Engineering” [Wirth2008]. The first term, “The Software Crisis”, was used to describe the inability of tackling the problems to be addressed in creating complex software systems. This results in software that comes too late, at higher costs than expected, and does not fulfill the promises made for it [Bauer1971]. The latter term, “Software Engineering”, was deliberately used provocatively, in implying the need for theoretical foundations and practical disciplines in the field of software manufacture [Bauer1971][Mahoney1990]. The field of software engineering basically studies and provides approaches to master complexity in the field of software manufacture. According to Dijkstra (1969) computer science is the study and management of complexity [Wegner1976].

After the software engineering field tried to manage this complexity from a data centric viewpoint, the object oriented paradigm arose. Object orientation is strongly based upon the classical view on concerns [Taivalsaari1997]. [Czarnecki1998] states that objects orientation has a lot of advantages, but failed in the management of complexity. He presents his solution in the Generative Programming approach, which combines Generic Programming, DSLs and Aspect Oriented Programming (AOP) with some additional techniques. [Bezivin2005] also points out some "incomplete achievement" of object technology and mentions MDSD and AOP as parts of a better way to deal with the complexity. In the [AMPLE] project research is being done on Aspect-Oriented, Model-Driven Product Line Engineering, aiming at the same.

In this work the focus has been set to the structural complexity of the product, the software. Software is often seen as the programs code, compiled or not, and sometimes documentation is included in the definition. [Vanderose2008] suggests defining software as a composite artifact of all artifacts within the development process, a view followed in this thesis.

2.5 CHANGEABILITY

Software complexity needs to be managed when change is to be applied. Changing requirements can have various causes, e.g. changes in stakeholder objectives or in the environment. In [Lientz1978] Lientz & Swanson defined corrective, adaptive, and perfective changes. This formed the foundations for ISO/IEC 14764. [Lientz1978] showed that over 60% of maintenance is perfective and [Vliet2000] showed that the most costs are being spent on perfective and adaptive changes.

According to philosopher Heraclitus change is constant. In [Brooks1987] this statement is applied to software, stating software is constantly subject to change. The software's structure determines largely the capacity to deal with these changes. How the software deals witch these changes determines its evolution. Within this thesis there is no distinction between development and maintenance phases, following the agile thought on change. Every structural change of an artifact in the development process is considered a change. All artifacts together are considered the system. Every change influences the evolution of the system.

As indicated by the first law of software evolution of [Lehman2001], software changes continuously or becomes less useful. As pointed out by the second law of [Lehman2001] this dynamic nature leads to growth of complexity of the software's structure, unless actions to prevent this are undertaken.

[Parnas1994] also acknowledges the fact of continuous need for change, and describes it as the driver of the inevitable aging of software. Software evolution and software aging are interchangeable concepts, describing the gradual development or growth of the complexity of software structure(s) and its associated decay. Parnas claims that there are two types of software aging. *The first is caused by the failure of the product's owners to modify it to meet changing needs, and the second is the result of the changes that are made.* The latter refers to the deterioration of structure caused by changes made. This deterioration increases complexity of the structure and feeds the inability to keep up with changing needs. [Parnas1994] emphasizes the importance of a focus on a long term health of products, because the process of software aging can be slowed down.

DEFINING CHANGEABILITY

Terms like changeability, flexibility, maintainability, modifiability, extensibility and adaptability all deal with change and are used interchangeably with different definitions, as can be seen in quality models from McCall, Boehm and ISO 9126 [Ber2005]. Often the terms differ only in the kind of changes they address.

In [Northrop2004] modifiability is about the cost of change and refers to the ease with which a software system can accommodate changes. [Bass2003] describes modifiability as the costs of change. In ISO 9126 changeability is a sub characteristic of maintainability and characterizes the amount of effort to change a system.

In [Eden2006] Eden and Mens use the term software flexibility interchangeable with software aging. The term is not used here to describe software evolution, but the way in which the software system is able to deal with change; the term flexibility is used as a property of the system to describe its ease of change. It describes how capable a system is to keep of deterioration.

MDSD aims at lowering complexity in order to minimize the costs of change. In this thesis therefore the term changeability refers to the ease of change, and is focused on the relation between the complexity of the structure and the amount of effort needed to apply changes. Changeability characterizes how well the structure accommodates change and minimizes the amount of effort needed. This definition conforms to ISO 9126.

OPTIMIZING CHANGEABILITY

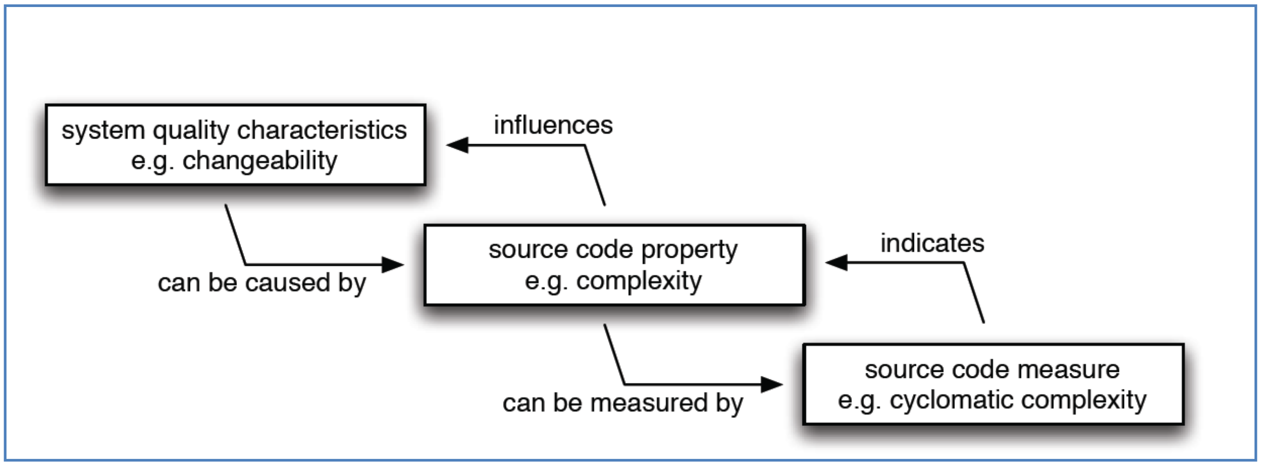


Figure 2 – Mapping system quality, source code properties and measures [Heitlager2007]

Figure 7 shows the mapping between system quality characteristics, source code properties and source code measures. The source code property related to changeability is complexity. In order to maximize changeability, complexity needs to be minimized.

Most ways to attack complexity can be brought back to a few techniques, being conceptualization, generalization, abstraction, separating of concerns [Dijkstra1974] and adding structure like hierarchies [Booch1993]. Conceptualizations form the base; they define the items to work with [Czarnecki1998].

Methods to attack complexity can take many forms, e.g. applying design principles like the Liskov Substitution Principle (LSP), tactics/patterns as described in [Bass2003][Buschmann1996][Gamma1995], or using MDSD. The methods focus on minimizing complexity, making the structure highly changeable.

To create a changeable structure, [Parnas1984] applies these techniques to create a hierarchical modular system based on information hiding. To build such a structure abstraction layers are used to reduce the amount of concept to deal with in each layer. This is a vertical separation of concerns (SOC). Per layer a horizontal SOC is applied to narrow the focus areas, creating loosely coupled high cohesive modules. These separations form a hierarchical structure, which brings cohesion to the global system.

In the approach taken in this project conceptualization can be found in the concepts used in the DSLs. Generalization can be seen in the components of the underlying framework. Abstraction is found in the form of DSLs, which also are used to separate concerns. An example of a hierarchical structure can be found in the code generation templates. In chapter 4 more information is given on the framework and the DSLs. Code generation templates can be found in the code fragments shown in appendix C.

ASSESSING CHANGEABILITY

In order to assess changeability qualitative or quantitative analysis can be used. The first often takes the form of a review, reflecting the user experience. The latter uses metrics to measure aspects of a source code property, see figure 2. Within this thesis both quantitative and qualitative analysis are used in order to make the assessment as good as possible.

Changeability refers to the ease of change and is reflected in the amount of effort needed for changes. This amount of effort is an external aspect of the complexity of the systems structure. Internal aspects of this complexity are e.g. size, coupling and cohesion. The internal aspects influence the external aspects; if the internal structure becomes more complex, more effort is needed to apply changes.

In the qualitative analysis these aspects can be assessed by the reviewer by reflecting his perception of them in an in-depth discussion on the technical maintainability of both the MDSD and the classical approach. This discussion provides a sense of the changeability of the approaches. But the perception can be influenced by many factors, such as the experience of the assessor or how he defines effort.

In the quantitative analysis metrics are used. Effort relates directly to time spend and is therefore straightforward to measure. Source code complexity cannot as directly be related to an aspect as effort. As shown in figure 2 the complexity can be measured by using the Cyclomatic Complexity metric. However the metric only looks at a certain aspect of complexity. To enhance the measurements reflection of the complexity other metrics can be applied, e.g. Lines of Code or Halstead Volume. The

MDSD environment is a limiting factor in the usable metrics, which is further discussed in section 4.2. Also the metrics interpretations are discussed in that section.

2.6 QUESTIONS ANSWERED BY LITERATURE

How can changeability of both approaches be defined? Changeability refers to the ease of change, and is focused on the relation between structure and effort. Aspects like understandability or testability are not part of the definition. Changeability characterizes how well the structure accommodates change. The source code property related to changeability is complexity. In order to maximize changeability, complexity needs to be minimized.

How can changeability of both approaches be assessed? Changeability can be assessed by qualitative or quantitative analysis. The first is often in the form of a review, reflecting the user experience. The latter uses metrics to measure properties of an aspect. For changeability the metrics can e.g. measure source code complexity and effort.

How can changeability of both approaches be optimized? Changeability can be optimized by applying techniques as conceptualization, generalization, abstraction, separating of concerns [Dijkstra1974] and adding structure like hierarchies [Booch1993].

3 HYPOTHESES FORMULATION

In this chapter a general expectation and related hypotheses are formulated based on the literature study and the experiment approach. The experiment is discussed in chapter 4. Below a high overview of the approach is given, followed by a description of the applied changes and measurements.

The experiment will contain two development environments, a model driven and a classical one. Both will be used to develop the same simple web application, based on the “Java Petstore” [Petstore]. On both environments changes will be applied, showing an expected evolution. Metrics will be used to analyze the changeability of the environment, and form the base of the quantitative analysis described in chapter 5. The maintainability index (MI) is used to determine complexity; time needed per change is used to measure effort. A qualitative analysis will be performed by conducting a developer’s review on the technical maintainability of both approached, as described in chapter 6. The goal of both analyses is to test the hypotheses. Based on the analyses results conclusions will be stated.

3.1 APPLIED CHANGES

[Deursen2007] defines four dimensions in the multi dimensional evolution of MDSD. In this research these dimensions can be mapped to languages in the MDSD approach, as shown in table 2.

Table 2 –Multi dimensional evolution layers and languages

Evolution layer	Languages
Abstraction	Set of XText grammars
Metamodel	XText grammar
Regular	DSL
Platform	XPand, XTend

The minimal change at the abstraction layer would be to add a grammar for a new DSL, which isn’t feasible within the project’s means. Changes at this level won’t be frequent. Changes that only affect the platform layer are also expected to be rare because the platform is fixed. The most changes are likely to occur on the metamodel and the regular layer for here the domain concepts are defined and used. The changes on the metamodel layer will ripple through the layers below. To reflect a normal evolution as possible within the means of the project the focus is set to the changes on the metamodel and the regular layer, leaving changes on the abstraction and platform layer out of scope.

The same six changes are applied to both environments. The changes were chosen to reflect a normal evolution, in which accounted and unaccounted changes take place. Unaccounted changes occur in the metamodel layer of the multi dimensional evolution of MDSD. The accounted changes occur in the regular layer. The expectance is that more changes will fall into the accounted category, as a result of a reasonable domain analysis. The applied changes and effected code parts are show below in table 3. To be on the save side a third of the applied changes was decided to fall into the unaccounted category.

The changes are all perfective, as most of the changes in software evolution are, see section 2.5. The changes are representative for the domain as they address its major concerns: content, presentation and navigation.

Table 3 – Applied changes

Changes	
Nr	**Description**
1	Add table Pet and relate it to Person
2	Add a field to a page and change the order of the fields
3	Add a page with personal information of the person logged in, including their pets
4	Add a link to the account page that points to the page added in change 3
5	Change the order of the menu items
6	Add pictures to products

The effort and maintainability metrics are used for measurements. Per approach per change the time needed to apply it is measured as well as per language the LOC and the Halstead vocabulary and length. Based on the measurements the Halstead volume and the MI are calculated.

Languages measured are:

- Classical approach
    - C#
    - HTML
    - SQL

- MDSD approach
    - XText
    - XPand
    - Xtend
    - Presentation DSL
    - Content DSL

The workflow language was left out for it interpreted as configuration. It is in line with project files of ASP.Net projects, which are also out of scope. Also CSS was used and categorized like the previous.

Code examples can be found in appendix C.

3.2 EXPECTATION

MDSD holds the promise of raising productivity by applying abstractions and automation. The abstractions reduce complexity, but complexity is also added e.g. by the introduction of co-evolution.

Changes on higher evolution layers, as metamodels, have a higher cost because changes ripple through lower layers. The impact of co-evolution can partly be limited by good domain analysis and which will lead to stable metamodels. Also the domain itself needs to be stable enough. Theoretically MDSD can deliver higher changeability than the classical approach by the reduction of complexity.

In the MDSD approach taken textual DSLs are used to model concerns of applications that form a software system family for transactional web applications. The tooling seems mature enough to make this kind of MDSD approach feasible. Because of tool improvement higher level changes are also lowered in costs. The concerns, the use of textual DSL's and the chosen tools have proven themselves before in many cases according to literature. The concerns prove to be stable and therefore the following result is expected:

The MDSD approach can provide equal or higher changeability than a classical development approach.

The main research question is thereby expected to be answered positively by the results of the experiment. The expectation is further refined to the hypotheses stated below.

3.3 HYPOTHESES

HYPOTHESIS 1

In the MDSD approach changes that are accounted for will be cheaper and unaccounted changes will be more expensive than in the classical approach.

The DSLs form a simplification of the classical approach in order to make changes cheaper. Both DSLs and the classical approach fall into the regular evolution dimension. The other evolution dimensions are needed to support the simplification.

Changes that are accounted for in the MDSD approach will only affect the DSLs. Caused by the simplification changes in DSLs will be smaller and often more local. Therefore they will take lower effort and have less impact on complexity growth than in the classical approach.

Unaccounted changes will be more expensive though. They can affect multiple evolution dimensions, e.g. a grammar change that will ripple through a DSL and a generator. Compared to the classical approach their implementations are larger and less local. Therefore they will take a larger effort and have higher impact on complexity growth than in the classical approach.

HYPOTHESIS 2

In early evolution in the MDSD approach the extra effort for the unaccounted changes will middle out against the lower effort for accounted changes, making the total effort competitive to that of the classical approach.

The effort needed to support an unaccounted change in the MDSD approach will take more effort than in the classical approach, accounted changes will take less. In early evolution the fundamentals of the domain are expected to be accounted for by the MDSD approach, making the larger part of changes lower on effort costs than in the classical approach. This will make up for the extra effort needed to implement unaccounted changes, leading to a competitive effort between the approaches in early evolution.

HYPOTHESIS 3

The MDSD approach will show a lower increase of complexity in its evolution than the classical approach.

In its evolution the domain coverage in the MDSD approach will grow after each implementation of an unaccounted change, making them accounted for. As the domain coverage grows the unaccounted changes will become less frequent. Most changes will be covered by the DSLs with a low impact on complexity. As the DSLs abstract from the classical approach the impact of changes is higher in the latter. The increase of complexity will therefore be lower in the MDSD approach than in the classical approach and the difference will grow larger during evolution.

HYPOTHESIS 4

Further into the evolution process effort will become lower in the model driven approach compared to the classical approach.

As stated in the latter paragraph the domain coverage grows in the MDSD approach, the unaccounted changes will become accounted for. The larger part of changes will be implemented in the DSLs. The changes in the DSLs are smaller and more localized than in the classical approach, making them less complex. This will translate itself to a lower effort needed to implement a change in the MDSD approach compared to the classical approach.

4 EXPERIMENT

In section 4.1 the experiments setup is presented. The base application is described as well as the development environments of the classical and the MDSD approach.

Section 4.2 discusses the metrics and tooling used for the measurements.

Section 4.3 describes the changes to be applied and why they were chosen.

4.1 EXPERIMENT SETUP

BASE APPLICATION

The base application is the application used in the project as a starting point and on which the changes are applied. Part of the “Java Petstore” functionality forms the base application. This is an e-commerce application which supports functionalities like browsing of a catalog and the placing of orders. The application falls into the category of transactional web applications. The application is used by Sun as a sample application to demonstrate a web application based their java platform. Microsoft uses a similar sample application called “Petshop” to show their best practices in building a .net web application. Both companies use the application as representative of a standard web application, and the Petstore is referenced in many books and papers, e.g. [Conallen2000].

The application is build on a model-view-controller architecture. *The MVC pattern divides an interactive application into three components. The model contains the core functionality and the data, Views display information to the user. Controllers handle user input* [Buschmann1996]. The separated components are loosely coupled which enhances maintainability. The pattern is often used in web applications and many web frameworks that implement it exist, e.g. ASP.NET MVC.

The base application is build with ASP.NET, uses a SQL Server database and runs on a windows platform, which is a standard setup for a Microsoft web application and a requirement of the domain.

To represent the complexity of small transactional web applications the essential functionalities form the base of the application. These essential functionalities are:

- Log in
- Create/update account
- Browse catalog
- Update shopping card
- Place order

The applications user interface is shown in appendix E.

An indicative functionpoint analysis [Nesma] was carried out by a professional function point counter of Capgemini BAS. In this analysis this application's size is counted to be 210 function points. This leaves enough room for the changes to be applied and still be large enough to be representative to the domain.

CLASSICAL DEVELOPMENT ENVIRONMENT

The classical approach uses the software shown below.

- MS Visual Studio 2008
- ASP.NET MVC Preview 3
- SQL SERVER 2005
- Windows 2003

MDSD ENVIRONMENT

ARCHITECTURAL DECISIONS

This section describes the architectural decisions that were made.

THE USE OF TEXTUAL DSLS

[Visser2008] states that most (successful) DSLs created to date are textual. There is more tooling available for textual languages and it is more mature. Adding graphical editors adds complexity to the approach. The MDSD approach needs to be as simple as possible, and will therefore use textual DSLs.

NO AIM FOR MDA COMPLIANCE

The MDA is interesting because it is a major influence in the field and it provides some useful thoughts and terminology. Though some of the aspects of the MDA might be useful and applied, the projects MDSD approach does not aim for MDA compliance. This is because the MDA's standards and vision form no perfect match to the goals of this project. The MDA focuses on portability, interoperability and reusability whereas the MDSD approach in this project focuses on changeability. Because of this difference in focus, striving for MDA compliance makes no sense. Besides that it would lead to overhead in the MDSD approach. This overhead would for example be introduced by unnecessary realization of platform independence or the use of the MOF as metametamodel for textual languages. The purpose of the MOF is to provide a means for describing graphical languages e.g. the UML and it is not designed to describe textual languages. Also no MDA compliant tooling supporting textual languages is known by the author, making compliance unfeasible.

THE METAMODELS/GRAMMARS WILL BE DESIGNED FROM SCRATCH

In order to research the changeability of a MDSD approach the whole process from designing the DSLs to modeling the application and to generating the application needs to be addressed in the experiment. Therefore the decision is made to design the DSLs from scratch.

NO USE OF CONSTRAINTS ON MODELS

OpenArchitectureWare provides a constraint language called Check. To keep the project feasible no constraints were used on the DSLs.

NO SPECIFIC MECHANISM WILL BE USED FOR HANDLING GENERATED AND MANUAL CODE

No specific mechanism will be used for extending generated with manual code. This kind of mechanism is too complex to realize within the means of the project and the MDSD approach can be used without. Therefore it is considered out of scope. For industrial use this issue will need to be addressed, e.g. by using partial classes or weaving manual code into the generated code.

TOOLING

The MDSD environment will generate to the classical environment. Both approaches will deliver the same kind of application implementation.

The following software is used:

- openArchitectureWare 4.3
- Eclipse 3.4.0 (eclipse-modeling-ganymede-incubation-win32 package, June 2008)

The choice is made to use oAW for this project because of the following. According to [Gharavi2008] oAW is currently one of the leading open-sourceMDA generator frameworks. It is very extensible and supports model-to-model and model-to-text transformations. Also it provides all that is needed for a textual MDSD approach in one package.

For a textual MDSD approach oAW offers the following languages:

- XText (T2M)
- XTend (M2M)
- XPand (M2T)
- Check (constraints)
- Workflow (sequencing)

With XText an EBNF grammar can be written for a DSL from which an editor can be generated. This editor provides default features as:

- syntax highlighting
- code completion
- code folding
- error decoration

In this editor a model can be created. Check can be used for model validation. XPand is a template language that can be used to transform the model to code. XTend can be used for model to model transformations. The Workflow language coordinates the total transformation.

OAW also provides tooling for product line engineering and a recipe framework for programmer guidance. Because OAW is very extendible, it can use any concrete syntax, whether graphical or textual.

DSLS

The classical approach addresses the concerns of the domain by languages as mapped in section 2.2. The MDSD needs to address the same concerns.

To keep the project feasible the baseline of the MDSD environment will be kept at the minimum needed, which means that the approach will start with two DSLs. Based on these domain concerns the decision was made to build a “Content” DSL and a “Presentation” DSL. The latter also addresses the navigation aspects. The metamodels of the DSL can be found in appendices A and B.

Business logic will be added manually when needed. Security, logging, and exception management will not be handled explicitly, for they are optional. Basic exception handling is added within the generator. In future work these cross-cutting concerns might be added in an Aspect Oriented manner. This leads to the following mapping in table 4.

Table 4 - Domain concerns and MDSD languages

Concern	Language
Content (Information)	Content DSL
Content (application logic)	Content DSL
Navigation	Presentation DSL
Presentation (layout)	Presentation DSL
Presentation (style)	CSS
Business logic	XPand
Security, logging, and exception management	XPand

MDSD APPROACH ARCHITECTURE

XText grammar based DSLs are build on top of the MVC.Net framework and used to model the application. XPand templates are applied, and extended with XTend functions, to generate the application from the models and to export data from one DSL to another. The approach is represented below in figure 4.

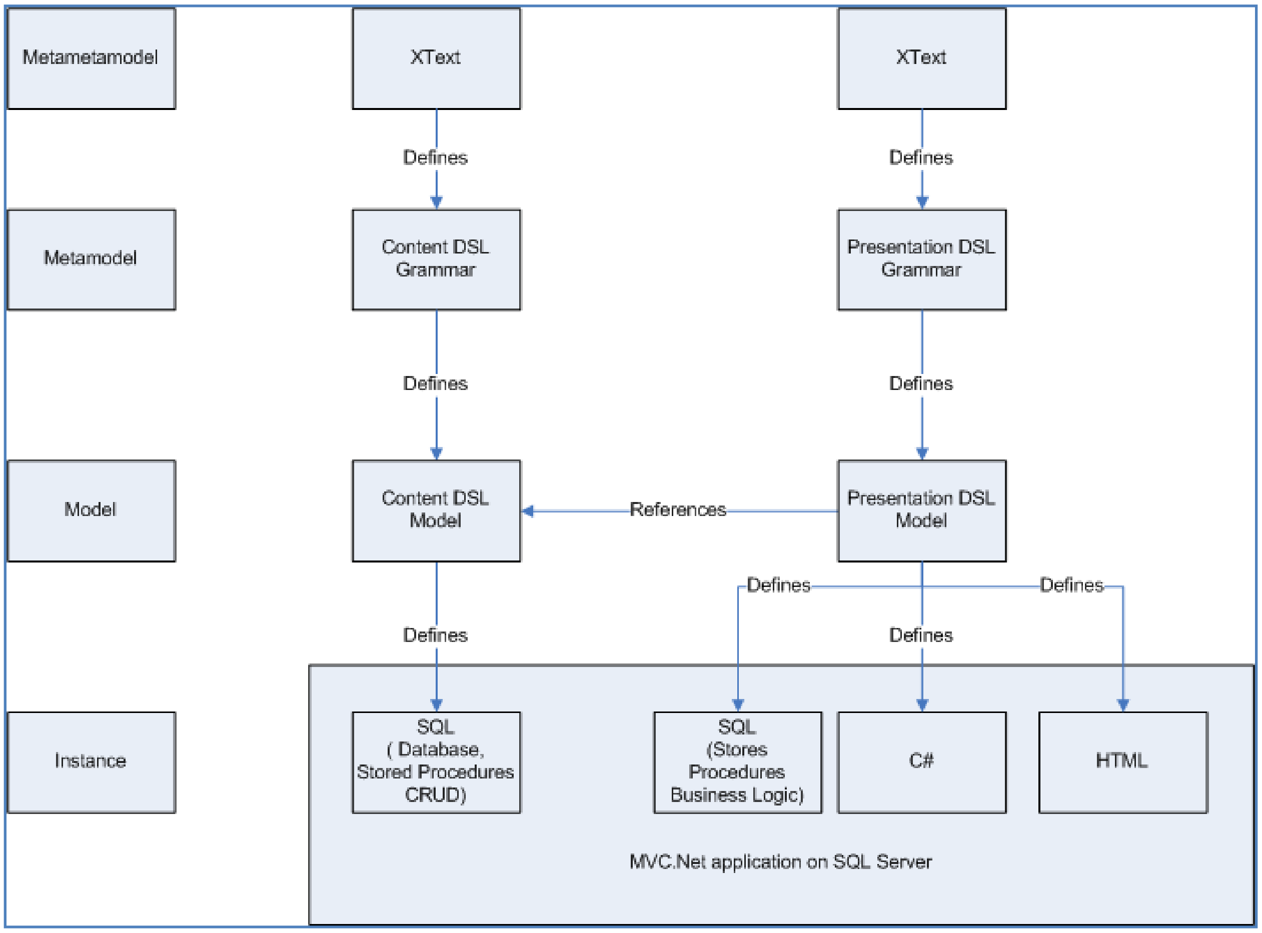


Figure 3 - MDSD Approach Architecture

4.2 METRICS

In the quantitative analysis the hypotheses described in chapter 3 are tested by using source code complexity and effort metrics. For the measurement of source code complexity several propertied can be looked at, e.g. cohesion, coupling and size. There are many metrics available to measure these kinds of properties, e.g. the Chidamber and Kemerer's Metrics Suite [Chidamber1994]. But to be of value in this research the metrics need to be generic, and the metrics mentioned before are object oriented. Grammar based metrics were also considered, but the grammars of the oAW languages were not available.

Generic complexity metrics found in literature are:

- Lines of Code
- Halstead Metrics
- McCabe Complexity

The Maintainability Index [Oman1994] combines the three. The choice was made to use the metric for this project. The metrics used in the MI are all complexity metrics and the MI basically is an indicator of the complexity of a program’s structure. As stated in section 2.5 this complexity influences the changeability, which is a sub characteristic of maintainability. The metric will be used as an indicator of the complexity of an approaches environment.

For effort the needed time per change is measured. This metric will be used to as an indicator of the needed effort of an approaches environment.

MAINTAINABILITY INDEX

The maintainability index (MI) was originally composed by Oman and Hagemeister. The metric is measured by applying a polynomial that combines other metrics and returns a single value. This polynomial is shown below. Oman and Hagemeister strived to a use a minimal number of metrics useful in predicting maintainability that are simple to measure [Oman1994]. The usefulness of the metric is lies in its aggregate strength and the simplicity of the concept [SEI]. The higher the value of the MI, the better the maintainability of the system is.

Maintainability Index =
Variant one: 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln (aveLOC)
Variant two: 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln (aveLOC) + 50 * sin (sqrt(2.4 *perCM))

aveV = average Halstead Volume V per module
aveV(g') = average cyclomatic complexity per module
aveLOC = the average count of lines of code (LOC) per module
perCM = average percent of lines of comments per module

[SEI]

In this research the choice was made to use the first variant, leaving comments out of scope. The reason is that the author agrees with [Heitlager2007] that counting the number of lines of comment in general has no relation with maintainability. As stated by [Heitlager2007] the use of the number of lines of comment as a metric implies code is better maintainable if commented. The metric does not evaluate the content of the comments which can be e.g. text that refers to the code, text that refers to previous versions of the code or commented out code. The stated implication is at least doubtful and therefore the comments are left out of scope.

Halstead Volume

Maurice Halstead introduces a suite of metrics in 1977 [Halstead1977] of which Halstead Volume is one. It represents the size of algorithms. The metric uses four primitive metrics:

n1 = number of distinct operators
n2 = number of distinct operands

N1 = total number of operators
N2 = total number of operands

From these the Vocabulary (n) and Length (N) are calculated:

n = n1 + n2
N = N1 + N2

The values of the Vocabulary and Length are used to measure the Halstead Volume (V):

V = N * log2(n)

Cyclomatic Complexity

The Cyclomatic Complexity metric (CC) was designed by Thomas McCabe in 1976 [Mccabe1976]. It measures program complexity by counting the number of independent paths in a program (P) and can be computed by the following formula in which the program is seen as a graph:

CC(Graph) = Number(edges) - Number(nodes) + Number(connected components)

Another way to determine a programs CC is to count the number of decision points in a routine and then add 1 and take the sum for the whole program.

Lines of Code

Lines of Code (LOC) is a metric that reflects the size of a program. The metric is used since the 1960's [Fenton2000] and is still very popular, probably because of its simplicity.

DISCUSSION: METRIC INTERPRETATION

Although this is not a study about metrics, the interpretation of the metrics influences the results. And the interpretation of the metrics used by the MI can be problematic and need to be done with care. For example Halstead does not present an unambiguous definition of the terms operator and operand, which can lead to different interpretations of the Halstead Volume metric. Another example can be supplied for LOC. various definitions of a line can be made up, leading to large differences in measurements. Jones reports differences as large as 500% [Jones1992].

By definition the metrics used in the MI are averaged per module. Some of the languages used (XText and the DSL programs) don't support modularization (yet) so this will influence the values. The definition of a module is unclear. According to [Kuipers2007] the natural languages concept for a module in C# can be a class or compilation unit, in COBOL it is a program and for C it is a file. For the same functionality a COBOL program would have 1 module and a C# program would have many. The choice is made to define a module as a physical file because this can be determined in all languages used and provides more similarity in what is compared then when the natural languages concept for a module is used. This choice means that if more files are used the MI becomes higher, representing better maintainability. This assumption is certainly arguable, because the content of the module might be useless or scattered leading to low cohesion.

The interpretation of the effort metric can also be problematic. For example the measurements are influenced by the person's knowledge and skills, which must be taken into account.

To analyze changeability of the approaches in a solid manner the decision is made to first perform a quantitative analysis in which the best suited metrics to measure internal and external properties will be used. And though the chosen metrics interpretations can be arguable, according to literature these metrics often have been successfully used for their purpose. Second the results of the previous will be brought into perspective by a qualitative analysis in the form of the developer's impression of the approaches. The strategy chosen in finding the best possible answer to the research question is to compare the changeability of the approaches in a quantitative-internal, a quantitative-external and a qualitative view and to analyze the results.

TOOLING

No tool was available that could measure the MI of the code for all other languages; therefore a tool was developed by the author.

In developing the tool some decisions needed to be made. To measure LOC for all the languages it was needed to define what a line was. The choice was made to use logical lines over physical lines. Physical lines are what humans perceive as lines, logical lines are statements. LOC is measured by counting statements defined per language. The tool uses simple filters to detect these statements. For example C# statements are detected by counting semicolons outside comments, for SQL the same filter is applied and for XPand the code between the characters "«" and "»" is counted. A grammar description in XText does not have statements, neither does an XTend library. Instead the unit measured in XText is a grammar rule and in XTend it is a function definition.

The use of the Cyclomatic Complexity (CC) cannot be motivated. This is caused by the missing concept of a routine in most of the languages, and the little use of decision points in some of them. The CC values needed for the calculation of the MI are set to 1 for all languages.

5 METRICS DATA

For the qualitative analysis internal complexity and effort metrics are measured per change and values are compared. Comparisons are made between:

- the complexity of the base application in both approaches
- the effort of the same versions of both approaches
- the complexity of the same versions of both approaches
- the complexity growth in both approaches

The versions, applied changes and affected code parts are shown in table 5 below.

Table 5 – Versions and affected code parts

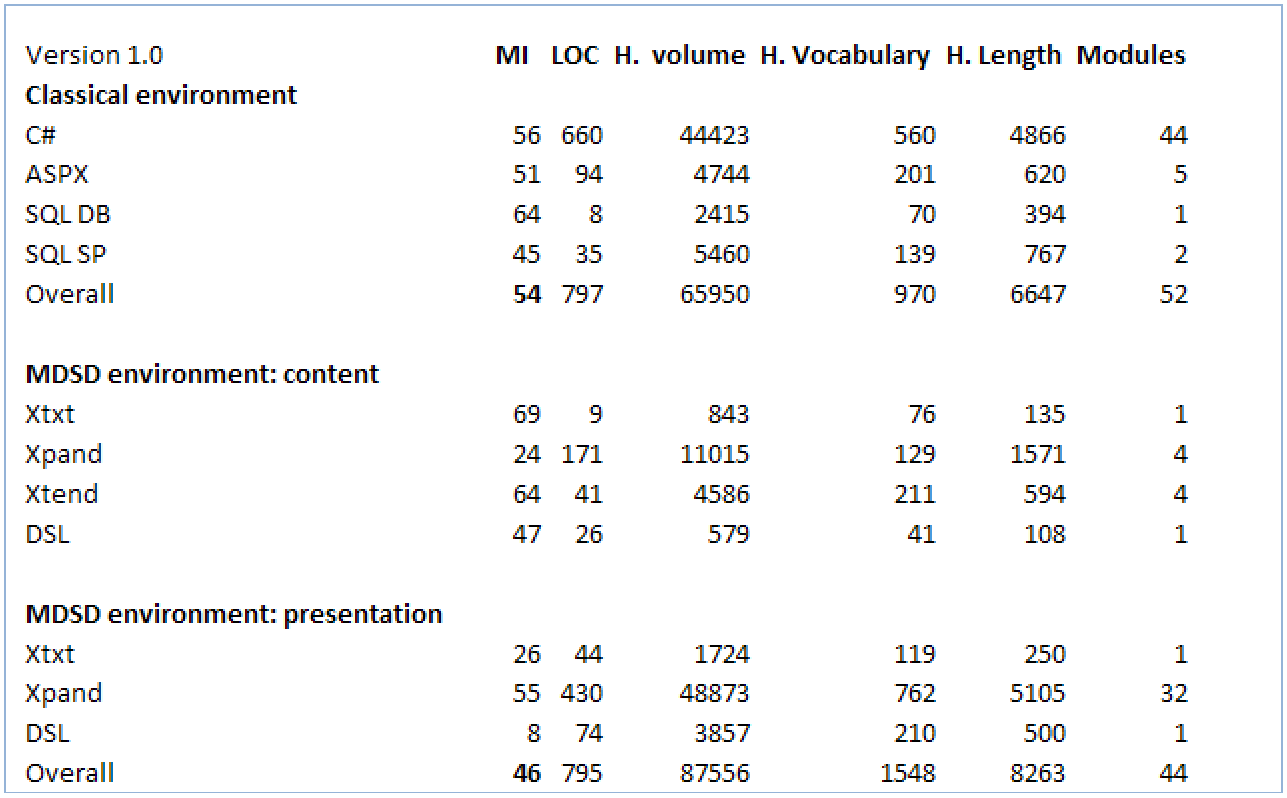
Version	Change
1.0	Base application
1.1	Add table Pet and relate it to Person
1.2	Add a field to a page and change the order of the fields
1.3	Add a page with personal information of the person logged in, including their pets
1.4	Add a link to the account page that points to the page added in change 3
1.5	Change the order of the menu items
1.6	Add pictures to products

Changes	Languages effected in the classical approach			
Version	C#	ASPX	DB	SP
1.1	x		x	x
1.2	x	x		x
1.3	x	x		x
1.4		x		
1.5		x		
1.6		x		

Changes	Languages effected in the model driven approach							
	Content				Presentation			
Version	DSL	Xtext	Xpand	Xtend	DSL	Xtext	Xpand	Xtend
1.1	x				x			
1.2					x			
1.3					x			
1.4					x	x	x	
1.5					x			
1.6					x	x	x	

5.1 MEASUREMENTS

Below the measurements of internal properties per version are shown. The versions relate to the changes, e.g. version 1.1 reflects change 1. Effected languages are marked with a blue background if the results are unchanged, and with grey when the numbers differ from the previous version. The changed values are given a red font color. Unchanged values have a black font.



Version 1.0	**MI**	**LOC**	**H. volume**	**H. Vocabulary**	**H. Length**	**Modules**
Classical environment						
C#	56	660	44423	560	4866	44
ASPX	51	94	4744	201	620	5
SQL DB	64	8	2415	70	394	1
SQL SP	45	35	5460	139	767	2
Overall	**54**	797	65950	970	6647	52
MDSD environment: content						
Xtxt	69	9	843	76	135	1
Xpand	24	171	11015	129	1571	4
Xtend	64	41	4586	211	594	4
DSL	47	26	579	41	108	1
MDSD environment: presentation						
Xtxt	26	44	1724	119	250	1
Xpand	55	430	48873	762	5105	32
DSL	8	74	3857	210	500	1
Overall	**46**	795	87556	1548	8263	44

Version 1.1	**MI**	**LOC**	**H. volume**	**H. Vocabulary**	**H. Length**	**Modules**
Manual code						
C#	51	796	54674	645	5858	46
ASPX	51	94	4744	201	620	5
SQL DB	60	9	2767	73	447	1
SQL SP	44	35	6160	145	858	2
Classical approach	**50**	934	78260	1064	7783	54
MDSD environment: content						
Xtxt	69	9	843	76	135	1
Xpand	24	171	11015	129	1571	4
Xtend	64	41	4586	211	594	4
DSL	43	29	655	44	120	1
MDSD environment: presentation						
Xtxt	26	44	1724	119	250	1
Xpand	55	430	48873	762	5105	32
DSL	6	78	4241	229	541	1
MDSD approach	**46**	802	88287	1570	8316	44

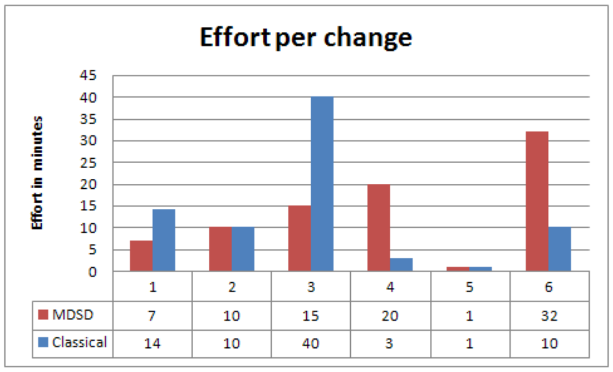
Version 1.2	**MI**	**LOC**	**H. volume**	**H. Vocabulary**	**H. Length**	**Modules**
Manual code						
C#	51	799	54959	649	5883	46
ASPX	50	96	4814	203	628	5
SQL DB	60	9	2767	73	447	1
SQL SP	41	39	6160	145	858	2
Classical approach	**50**	943	78655	1070	7816	54
MDSD environment: content						
Xtxt	69	9	843	76	135	1
Xpand	24	171	11015	129	1571	4
Xtend	64	41	4586	211	594	4
DSL	43	29	655	44	120	1
MDSD environment: presentation						
Xtxt	26	44	1724	119	250	1
Xpand	55	430	48873	762	5105	32
DSL	6	79	4257	229	543	1
MDSD approach	**46**	803	88308	1570	8318	44

Version 1.3	MI	LOC	H. volume	H. Vocabulary	H. Length	Modules
Manual code						
C#	53	833	57394	674	6108	51
ASPX	51	114	5788	223	742	6
SQL DB	60	9	2767	73	447	1
SQL SP	41	39	6457	158	884	2
Classical approach	**51**	995	82952	1128	8181	60
MDSD environment: content						
Xtxt	69	9	843	76	135	1
Xpand	24	171	11015	129	1571	4
Xtend	64	41	4586	211	594	4
DSL	43	29	655	44	120	1
MDSD environment: presentation						
Xtxt	26	44	1724	119	250	1
Xpand	55	430	48873	762	5105	32
DSL	3	87	4487	232	571	1
MDSD approach	**46**	811	88629	1573	8346	44

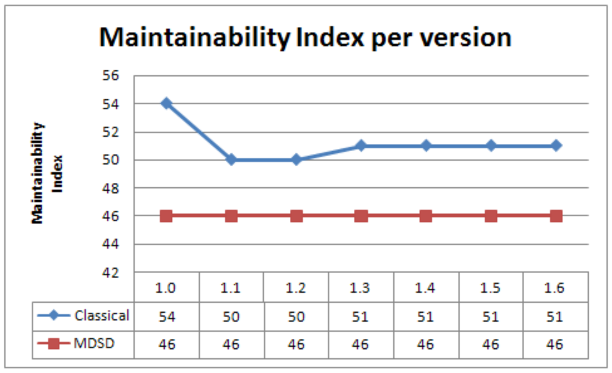
Version 1.4	MI	LOC	H. volume	H. Vocabulary	H. Length	Modules
Manual code						
C#	53	833	57394	674	6108	51
ASPX	50	118	5975	226	764	6
SQL DB	60	9	2767	73	447	1
SQL SP	41	39	6457	158	884	2
Classical approach	**51**	999	83206	1131	8203	60
MDSD environment: content						
Xtxt	69	9	843	76	135	1
Xpand	24	171	11015	129	1571	4
Xtend	64	41	4586	211	594	4
DSL	43	29	655	44	120	1
MDSD environment: presentation						
Xtxt	26	44	1724	119	250	1
Xpand	55	430	48873	762	5105	32
DSL	3	88	4525	234	575	1
MDSD approach	**46**	812	88686	1575	8350	44

Version 1.5	**MI**	**LOC**	**H. volume**	**H. Vocabulary**	**H. Length**	**Modules**
Manual code						
C#	53	833	57394	674	6108	51
ASPX	50	118	5975	226	764	6
SQL DB	60	9	2767	73	447	1
SQL SP	41	39	6457	158	884	2
Classical approach	**51**	999	83206	1131	8203	60
MDSD environment: content						
Xtxt	69	9	843	76	135	1
Xpand	24	171	11015	129	1571	4
Xtend	64	41	4586	211	594	4
DSL	43	29	655	44	120	1
MDSD environment: presentation						
Xtxt	26	44	1724	119	250	1
Xpand	55	430	48873	762	5105	32
DSL	3	88	4525	234	575	1
MDSD approach	**46**	812	88686	1575	8350	44

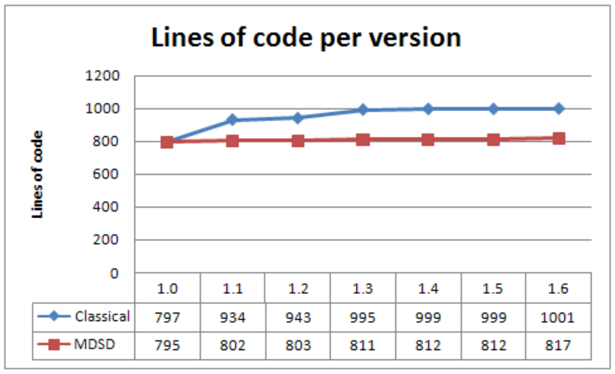
Version 1.6	**MI**	**LOC**	**H. volume**	**H. Vocabulary**	**H. Length**	**Modules**
Manual code						
C#	53	833	57394	674	6108	51
ASPX	49	120	5989	234	761	6
SQL DB	60	9	2767	73	447	1
SQL SP	41	39	6457	158	884	2
Classical approach	**51**	1001	83259	1139	8200	60
MDSD environment: content						
Xtxt	69	9	843	76	135	1
Xpand	24	171	11015	129	1571	4
Xtend	64	41	4586	211	594	4
DSL	43	29	655	44	120	1
MDSD environment: presentation						
Xtxt	24	48	1825	125	262	1
Xpand	55	430	48873	762	5105	32
DSL	3	89	4553	235	578	1
MDSD approach	**46**	817	88899	1582	8365	44



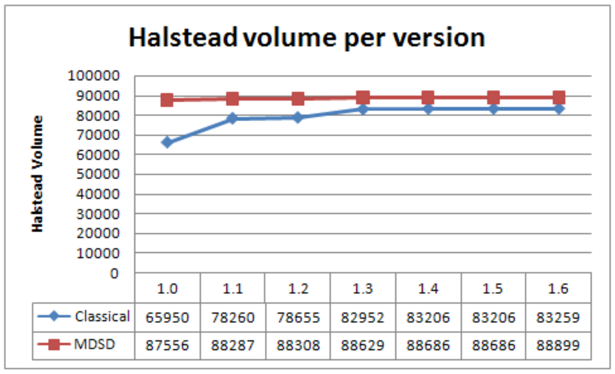
	1	2	3	4	5	6
🟥 MDSD	7	10	15	20	1	32
🟦 Classical	14	10	40	3	1	10



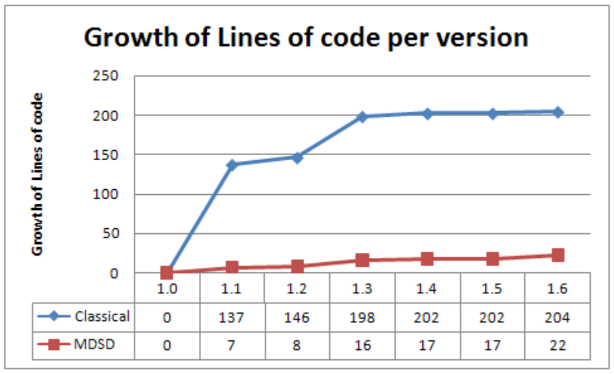
	1.0	1.1	1.2	1.3	1.4	1.5	1.6
Classical	54	50	50	51	51	51	51
MDSD	46	46	46	46	46	46	46



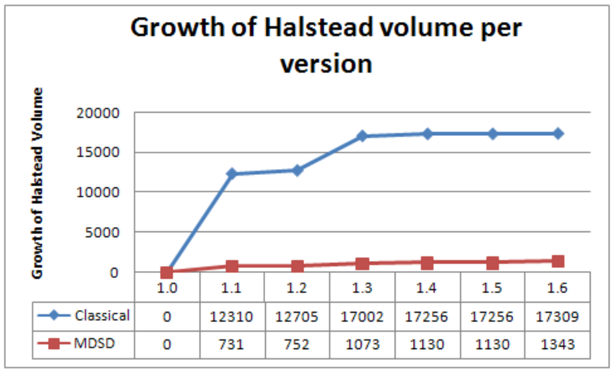
	1.0	1.1	1.2	1.3	1.4	1.5	1.6
Classical	797	934	943	995	999	999	1001
MDSD	795	802	803	811	812	812	817



	1.0	1.1	1.2	1.3	1.4	1.5	1.6
Classical	65950	78260	78655	82952	83206	83206	83259
MDSD	87556	88287	88308	88629	88686	88686	88899



	1.0	1.1	1.2	1.3	1.4	1.5	1.6
Classical	0	137	146	198	202	202	204
MDSD	0	7	8	16	17	17	22



	1.0	1.1	1.2	1.3	1.4	1.5	1.6
Classical	0	12310	12705	17002	17256	17256	17309
MDSD	0	731	752	1073	1130	1130	1343

Figure 4 – Graphs of measured values

5.2 TESTING OF HYPOTHESES

HYPOTHESIS 1

In the MDSD approach changes that are accounted for will be cheaper and unaccounted changes will be more expensive than in the classical approach.

Changes 1, 2, 3 and 5 are accounted for in the MDSD approach, 4 and 6 are unaccounted for.

Change 5 affected only one language in each approach. It altered the structure of the code in both approaches by reordering it, but it did not reflect at all in the complexity measurements for the reordering did not affect LOC or Halstead volume. In the classical approach three languages are affected by changes 1, 2 and 3, in the MDSD approach only one DSL is affected. Changes 4 and 6 are of the metamodel evolution kind and ripple through the regular and platform evolution. These changes affect three languages in the MDSD approach, in contrast to affecting only on language in the classical approach. Overall the accounted changes are more local than in the classical approach and unaccounted changes less.

Changes 2 and 5 show an equal effort cost in both approaches. Changes 1 and 3 show a higher effort cost in the classical approach. Changes 4 and 6 are more expensive in effort in the model driven approach. In the MDSD approach the total effort for accounted changes is 33 minutes; this is 65 minutes in the classical approach. The unaccounted changes cost 52 minutes in the MDSD approach, which took 13 minutes in the classical approach. The effort measurements confirm the hypothesis.

The added lines of code for a change are higher for almost all changes in the classical approach. Change 5 is equal as no lines are added in both approaches. Change 6 has a slightly lower amount of added lines in the classical approach and the amount of added lines for change 4 differs only a little between both approaches. The unaccounted changes show a competitive growth between the approaches. The amount of added lines in change 1, 2 and 3 showed to be larger in the classical approach. The accounted changes present a lower code growth in the MDSD approach than in the classical approach.

The graph of growth of Halstead Volume per version shows a similar shape as the growth of Lines of code per version. But the Halstead Volume growth is higher for all changes in the classical version; the difference is only less for change 4 and 6.

In the MDSD approach the MI values do not vary, the changes have a too low impact on the complexity. In the classical approach the MI values do vary, though only little. First the values decrease, later they increase because of added modules. The MI values show no relation to the other measurements as they are barely affected by the changes.

The effort measurements confirm the hypothesis. The LOC and Halstead Volume measurements show the accounted changes to have lower impact on complexity growth in the MDSD approach. The unaccounted changes show a competitive growth of lines of code between the approaches and less Halstead Volume growth in the MDSD approach. Still change 4 and 6 costs a lot more effort in the MDSD approach. One cause of this is the fact that the unaccounted changes in the MDSD approach are of the metamodel evolution kind and ripple through the regular and platform evolution. Another cause is the fact that both XText and the DSLs do not support modularization, which makes it harder to find the place of change and which also influences the MI values negatively.

HYPOTHESIS 2

In early evolution in the MDSD approach the extra effort for the unaccounted changes will middle out against the lower effort for accounted changes, making the total effort competitive to that of the classical approach.

As stated in the previous section, in the MDSD approach the total effort for accounted changes is 33 minutes; this is 65 minutes in the classical approach. The unaccounted changes cost 52 minutes in the MDSD approach, which took 13 minutes in the classical approach.

The total effort of the MDSD approach is 84 minutes and for the classical approach this is 78. The difference is relatively small. The measurements confirm the hypothesis.

HYPOTHESIS 3

The MDSD approach will show a lower increase of complexity in its evolution than the classical approach.

The MDSD approach starts off with a slightly lower MI value than the classical approach, caused by the following. In contrast with MI values of other languages the MI values of the Presentation DSL lie extremely low, with a lowest measurement of 3. The cause lies in the combination of a small amount of code with a high Halstead Volume, and a low modularization. The high Halstead Volume is caused by the imported elements from the Content DSL, which was needed because it was not possible to reference these elements. The lack of modularization is caused by the fact that XText doesn't support modularization (yet). All grammars and DSL programs reside in one file each. This also explains the low MI value for the XText code in the presentation part. When modularization can be used the MDSD approach will start with a competitive or even better MI value than the classical approach. Modularization will be supported by XText in the future.

The MI values of the MDSD approach do not vary, whereas the MI values of the classical approach decrease over time, showing a faster growth of complexity. This is despite of the lack of modularization for XText and the DSL programs.

The values of LOC and Halstead volume underline this picture. The total growth of LOC differs with more than a factor 9. The total growth of Halstead volume differs close to a factor 13.

The measurements confirm the hypothesis.

HYPOTHESIS 4

Further into the evolution process effort will become lower in the model driven approach compared to the classical approach.

The total measured effort per approach differs only slightly. When relating the effort measurement per change to the parts effected it shows what was expected. Changes accounted for in the model driven approach show an equal or lower effort. Unaccounted changes show a higher effort in the approach.

In the MDSD approach the changes on the metalevel are applied to make a kind of change be supported by the DSL. Therefore changes unaccounted changes will first be on the metalevel and ripple through the levels below. This is expensive at first, but the second time this type of change needs to be applied it will be cheap for it is now supported by the DSL.

The effort for applying changes that occurred on the metalevel for the second time is measured. The time needed to apply change 4 or 6 a second time was 1 minute for each. Taking these two changes into the equation would lead to 87 minutes for the MDSD approach against 91 minutes for the classical approach, which confirms the hypothesis. As domain coverage will continue to grow the percentage of unaccounted changes will become lower, making the average effort per change in the MDSD approach decrease even more in comparison to the classical approach.

EXPECTATION

The MDSD approach can provide equal or better changeability than a classical development approach.

The MDSD approach starts off with a slightly lower MI value, caused by low modularization. The measurements indicate that complexity increases faster in the classical approach. The overall effort is competitive between both approaches. The effort differs on the kind of changes. In the MDSD approach unaccounted changes proved to be most expensive. Because of the grammar’s growing domain coverage the changeability of the MDSD approach will increase in time. Based on the measurements the conclusion is that the MDSD approach starts on a competitive level of changeability compared to the classical approach, and will evolve to a better changeability.

6 DISCUSSION ON TECHNICAL MAINTAINABILITY

This chapter gives an "in-depth" discussion on the technical maintainability of both approaches. The developer's impression of the changeability of the approaches gets reflected. The developer's is the person who applied the changes in the experiment. This role was conducted by the author. The stated hypotheses are tested in this discussion.

HYPOTHESIS 1

In the MDSD approach changes that are accounted for will be cheaper and unaccounted changes will be more expensive than in the classical approach.

In the classical approach the small size of the project, the clear architecture and modular structure made it easy to locate the location of change. In the MDSD approach it was not hard to oversee the structure and pinpoint the location of change either, mainly because of the limited size of the project. This location was harder to find in the larger files, which often were XPand files. But the large size of certain files was not caused by changes; they were large from the start.

ACCOUNTED CHANGES

Overall the accounted changes were easier to change in the MDSD approach than in the classical approach. Changes were most often smaller in size and more localized. For the first change code was added in two DSLs, the other accounted changes only affected one DSL. In the classical approach all these changes affected three languages, except for change number 5 where only one language was affected. In the first change the objects were exported from one DSL to another in order to use them. In the developers opinion this polluted the code. The suggestion was made by the developer to implement a referencing mechanism to prevent this.

The DSL editors default provided features turned out useful in saving time, e.g. code completion made it easy to loop up correct commands and error decoration prevented problems further into the development process. The lack of support for modularity for XText grammars and DSLs made the files larger than the developer wanted, but because of the small size of the project it did not influence the needed effort. The developer indicated that he expects in further evolution that the files will grow large leading to maintainability and scalability problems.

The step of generating the code formed overhead compared to the classical approach. No problems were encountered in the generation step and it took little time. Still the developer commented that the need for total regeneration of the application to propagate a change from model to application is very inefficient and could lead to issues in further evolution.

As the MDSD approach did not have 100% code generation for some changes a small amount of manual code needed to be added to the generated code. A mechanism to deal with this was stated as out of scope for the project. But, as also stated by the project, this will need to be addresses in future use, because the manual code is overwritten in regeneration. Because of the lack of such a mechanism the developer had a hard time finding the place for adding the manual code.

As the MDSD approach generates to the classical environment building and testing is similar in both the MDSD and the classical approach. Debugging in the MDSD approach would lead to unaccounted changes in the XPand templates, but no bugs have been found in the project.

UNACCOUNTED CHANGES

The changes 4 and 6 were unaccounted changes. In both changes the grammar was adapted, which rippled through the DSL code and the generation templates. These changes took a lot more effort in the MDSD approach than in the classical approach.

Finding the place of change in the grammar was easy, because the grammar was small. Adapting it was relatively simple too. The developer stated that the lack of experience with the language could have influenced the needed effort. Adapting the DSL was similar as for the accounted changes as described in the previous section.

Adapting the XPand templates turned out to take some more effort. The templates can become quite large, and because the formatting of the template's code influences the generated code the templates are not very human readable. The templates link the grammar's concepts to implementation code. The implementation was first build, tested and debugged in the classical approach, and then the code was copied to a template that walks through the related grammar concept.

The effort for the unaccounted changes beholds the complete effort needed in the classical approach, the code to make it an accounted change meaning the grammar and the templates as well as the code to implement it in the DSL so the implementation code can be generated from it. Therefore these kinds of changes are more expensive in the MDSD approach than in the classical approach.

HYPOTHESIS 2

In early evolution in the MDSD approach the extra effort for the unaccounted changes will middle out against the lower effort for accounted changes, making the total effort competitive to that of the classical approach.

The developer stated the implementation of accounted changes in the MDSD approach to be simpler in terms of notation and smaller in size than in the classical approach, and related it to the effort needed which was clearly less in the MDSD approach. The unaccounted changes showed the opposite. The total effort in both approaches was quite competitive, therefore the hypothesis is perceived as true by the developer.

HYPOTHESIS 3

The MDSD approach will show a lower increase of complexity in its evolution than the classical approach.

Most of the changes fall into the accounted category. For the MDSD approach this translates to changes in the DSLs programs. These did grow only little in comparison to the code in the classical approach. The unaccounted changes were on a more competitive level in growth of size. Overall the MDSD approach showed a smaller increase in size as the classical approach.

The lack of modularization for grammars and DSL programs made the growth of size caused by a change of influence to other changes. The point of change must be found in a larger grammar or DSL program in a single file than before. In the classical approach the changes did not seem to interfere in this way. Though in further evolution this is certain to happen, the better modularization prevented this for these changes. The size of the grammars and DSL programs did not affect the needed effort however as the code was still small enough to be easily comprehended.

The developer valued the smaller growth of size above the lack of modularization for grammars and DSL programs. The hypothesis was perceived as true.

HYPOTHESIS 4

Further into the evolution process effort will become lower in the model driven approach compared to the classical approach.

Implementing accounted changes in the MDSD approach turned out to take a lot less effort than in the classical approach. When the unaccounted changes 4 and 6 were applied a second time it took almost no effort at all as they were now accounted for. The domain coverage grew because of the added concepts to the grammar which will lead to a smaller percentage of unaccounted changes. The needed effort in the MDSD approach will become smaller because of this.

But the code base is likely to grow in the evolution and with it the overall complexity. This is expected to influence the needed effort in both approaches. As the classical approach shows a faster growth this still underlines the hypothesis. The lacking modularization for XText and DSL programs could become an issue though.

The hypothesis is confirmed.

EXPECTATION

The MDSD approach can provide equal or better changeability than a classical development approach.

In the experiment the effort and complexity of the approaches felt competitive. The percentage of unaccounted changes was perceived as high in the project. During further evolution this percentage is expected to lower even more. When the issues mentioned in the review will be addressed, the MDSD approach is expected to beat the classical approach in terms of changeability.

In evaluation of the experiment the expectation is perceived as valid by the developer.

7 THREATS TO VALIDITY

7.1 INTERNAL VALIDITY

The metrics used were fairly simple as was the implementation of the measurement tool; both form no threat to validity of the results.

Though the changes do reflect an expected evolution, the amount of changes applied is quite small. This limits the value of the projects results, e.g. the impact on the results of possible noise in the measurements is higher than in with many applied changes.

The validity of the qualitative analysis is threatened by the fact that it is conducted by one developer only and the fact that this developer is the author, who could be biased. The author conducted the review as objective as possible.

It is possible for the results of the qualitative and the quantitative analysis to contradict one another; this could happen if unforeseen factors would influence the developer's experience. As the developer and the author are the same person this is unlikely to happen.

7.2 EXTERNAL VALIDITY

The results of the project are specific to the domain of small e-commerce web applications that are build on an ASP.Net architecture. However the author sees no reason why results would differ for other application types or other platforms.

The expected evolution of the MDSD approach assumes a fixed platform. The use of multiple platforms would lead to a higher level of changes on the platform evolution layer, which needs to be taken in to account. These kinds of changes are likely to be built in a classical environment after which they are copied to the generators templates. The latter step would make the MDSD implementation more expensive than the classical one, but the MDSD implementation can be reused. The project does not cover how the use of multiple platforms would affect changeability.

The used tooling and the architectural decisions are fundamental to the MDSD approach. Therefore they have a strong influence on the results. The results of the project do not extend to other kinds of MDSD approaches.

8 CONCLUSIONS

In the project the changeability of a model driven web development approach was assessed. Hypotheses were formulated and tested by qualitative and quantitative analysis to come to an answer. Did the MDSD approach supply the needed changeability to be applied to the domain of small web applications?

8.1 QUANTITATIVE ANALYSIS

The experiment confirmed the stated expectation and hypotheses. The measurements showed the implementation of accounted changes to take less effort in the MDSD than in the classical approach, and vice versa for the unaccounted changes. The extra effort needed to implement unaccounted changes in the MDSD approach in the early evolution did middle out against the lower effort needed for the implementation of accounted changes. The total effort needed for the implemented changes was competitive between the MDSD and the classical approach. The experiment showed the changeability of the MDSD approach to be equal or better in comparison to the classical approach. It supplied the needed changeability to be applied to the chosen domain.

Complexity increased steeper in the classical than in the MDSD approach, which is expected to negatively influence the future needed effort to implement changes. Also during evolution the needed effort in the MDSD approach was influenced positively by the growing domain coverage, which will lower the percentage of future unaccounted changes. Therefore the conclusion is drawn that not only the changeability between the approaches turned out to be competitive, also the difference in changeability between the approaches grows larger, in favor of the model driven approach.

8.2 QUALITATIVE ANALYSIS

In the discussion on technical maintainability needed effort for changes showed to be influenced by structural complexity, e.g. this complexity affected the ease of finding the point of change. The tooling also turned out to influence the effort need. For example the default features of the DSL editors as code completion did saved time. The notation of the DSLs was conceived as a positive influence, being concise and simple. This also positively influenced the growth of code. In implementing unaccounted changes the XPand templates turned out to take the most effort as they are hard to read and did grow large. The overhead encountered in implementing unaccounted changes in the MDSD approach did not outweigh the efficiency gained in implementing accounted changes in the approach in comparison to the effort needed in the classical approach.

Some issues of the MDSD approach were stated as concerns to be addressed as they were perceived as negative influences on the ease of change. One was the lack of support for modularity for XText and DSL programs, which made certain files grow larger than wanted. This is likely to lead to maintainability and scalability problems in the future. Another concern was the export of objects between DSL programs instead of a using a referencing mechanism, which unnecessary increased the program size. A third concerns stated was the need for total regeneration to implement a change which and a fourth was a missing mechanism to add manual code to the generated code while the approach did not support 100% code generation.

Some critical notes on the MDSD approach were stated. Still the results of the experiment were underlined, confirming the stated expectation and hypotheses. In providing higher changeability the review favored the MDSD above the classical approach. Addressing stated issues could better the changeability of the MDSD approach even more.

8.3 MODULARITY

Modularity is an important technique in managing complexity, and makes it possible to separate concerns and apply structure. It is a large factor in the Maintainability Index metric. Lack of modularization will make files grow large leading to maintainability and scalability problems.

The openArchitectureWare tooling lacks support for modularization of grammars and DSL programs at the moment, which had a large effect on the experiment. The lack of support for modularization of grammars and DSL programs was also mentioned in the discussion on technical maintainability as a large negative influence on the ease of change, as it makes it harder to overlook the code and find the point of change.

If the modularization was supported the Maintainability Index measurements would have been in favor for the model driven approach from the start. Possibly effort measurements would be influenced too, though the small size of the project made most code still easily comprehensible. But as size of code is likely to grow in evolution the lack of modularization most probably will influence the ease of change and become a concern to be addressed. And if this kind of MDSD approach will be applied to a domain of larger applications the issue will need to be addressed even in early evolution.

Though the main research question is answered positively, the author believes the most valuable lesson learned in the project was that of the impact of support for modularity in a MDSD approach on changeability.

BIBLIOGRAPHY

[Bass2003]	Bass, L., Clements, P. & Kazman, R., *Software Architecture in Practice, Second Edition* Addison-Wesley Professional, Hardcover, **2003**
[Bauer1971]	Bauer, F.L., *Software Engineering* IFIP Congress (1), **1971** , pp. 530-538
[Ber2005]	Ber, P., Damm, L., Eriksson, J., Gorschek, T., Henningsson, K., Jönsson, P., Kågström, S., Milicic, D., Mårtensson, F., Rönkkö, K., Tomaszewski, P., Lundberg, L., Mattsson, M. & Wohlin, C., *Software quality attributes and trade-offs* http://www.bth.se/tek/besq.nsf/(WebFiles)/5A52350A52726F51C12570A8004CB613/$FILE/Software_quality_attributes.pdf, **2005**
[Bezivin2005]	Bezivin, J., *On the unification power of models* Software and System Modeling, **2005** , Vol. 4 (2) , pp. 171-188
[Bezivin2001]	Bezivin, J. & Gerbe, O., *Towards a Precise Definition of the OMG/MDA Framework* ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering, IEEE Computer Society, **2001** , pp. 273
[Booch1993]	Booch, G., *Object-Oriented Analysis and Design with Applications (2nd Edition)* Addison-Wesley Professional, Hardcover, **1993**
[Brooks1987]	Brooks, F.P., *No Silver Bullet Essence and Accidents of Software Engineering* Computer, IEEE Computer Society Press, **1987** , Vol. 20 (4) , pp. 10-19
[Buschmann1996]	Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M., *Pattern-oriented software architecture: a system of patterns* John Wiley & Sons, Inc., **1996**
[Chidamber1994]	Chidamber, S.R. & Kemerer, C.F., *A Metrics Suite for Object Oriented Design* IEEE Trans. Softw. Eng., IEEE Press, **1994** , Vol. 20 (6) , pp. 476-493
[Conallen2000]	Conallen, J., *Building Web applications with UML* Addison-Wesley Longman Publishing Co., Inc., **2000**
[Czarnecki1998]	Czarnecki, K., *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models* Technische Universitlt Ilmenau, Germany, **1998**

[Deursen2000] van Deursen, A., Klint, P. & Visser, J., *Domain-specific languages: an annotated bibliography* SIGPLAN Not., ACM, **2000** , Vol. 35 (6) , pp. 26-36

[Deursen2007] van Deursen, A., Visser, E. & Warmer, J., *Model-Driven Software Evolution: A Research Agenda* Tamzalit, D. *(ed.)*, CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007), **2007** , pp. 41-49

[Deursen1997] Deursen, A. v. & Klint, P., *Little Languages: Little Maintenance?* Kamin, S. *(ed.)*, First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL'97, **1997** , pp. 109-127

[Dijkstra1974] Dijkstra, E.W., *On the role of scientific thought*, **1974**

[Eden2006] Eden, A.H. & Mens, T., *Measuring software flexibility* IEE Software, IEE Proceedings, **2006**

[Erlikh2000] Erlikh, L., *Leveraging Legacy System Dollars for E-Business* IT Professional, IEEE Computer Society, **2000** , Vol. 2 (3) , pp. 17-23

[Favre2004] Favre, J. & NGuyen, T., *Towards a Megamodel to Model Software Evolution through Transformations* SETRA Workshop, Elsevier ENCTS, **2004** , pp. 59-74

[Fenton2000] Fenton, N.E. & Neil, M., *Software metrics: roadmap* ICSE '00: Proceedings of the Conference on The Future of Software Engineering, ACM Press, **2000** , pp. 357-370

[Fowler2003] Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language* Addison-Wesley Longman Publishing Co., Inc., **2003**

[France2007] France, R. & Rumpe, B., *Model-driven Development of Complex Software: A Research Roadmap* FOSE '07: 2007 Future of Software Engineering, IEEE Computer Society, **2007** , pp. 37-54

[France2006] France, R.B., Ghosh, S., Dinh-Trong, T. & Solberg, A., *Model-Driven Development Using UML 2.0: Promises and Pitfalls* Computer, IEEE Computer Society, **2006** , Vol. 39 (2) , pp. 59-66

[Fraternali1999] Fraternali, P., *Tools and Approaches for Developing Data-Intensive Web Applications: A Survey* ACM Comput. Surv., **1999** , Vol. 31 (3) , pp. 227-263

[Gamma1995] Gamma, E., Helm, R., Johnson, R. & Vlissides, J., *Design patterns: elements of reusable object-oriented software* Addison-Wesley Longman Publishing

Co., Inc., **1995**

[Gharavi2008] Gharavi, Mesbah & van Deursen, *Modelling and Generating Ajax Applications: A Model-Driven Approach* 7th International Workshop on Web-Oriented Software Technologies (IWWOST'08), **2008** , pp. 38-43

[Greenfield2004] Greenfield, J., Short, K., Cook, S. & Kent, S., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* John Wiley & Sons, **2004**

[Hailpern2006] Hailpern, B. & Tarr, P., *Model-driven development: the good, the bad, and the ugly* IBM Syst. J., IBM Corp., **2006** , Vol. 45 (3) , pp. 451-461

[Halstead1977] Halstead, M.H., *Elements of Software Science (Operating and programming systems series)* Elsevier Science Inc., **1977**

[Heitlager2007] Heitlager, I., Kuipers, T. & Visser, J., *A Practical Model for Measuring Maintainability* QUATIC '07: Proceedings of the 6th International Conference on Quality of Information and Communications Technology, IEEE Computer Society, **2007** , pp. 30-39

[Jones1992] Jones, C. Research Report, S.I.N. 1992. (64. p., *Critical Problems in Software Measurement* Software Productivity Research, **1992** , Vol. November

[Kappel2006] Kappel, G., *Web Engineering* John Wiley & Sons, **2006**

[Kleppe2008] Kleppe, A., *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels* Addison-Wesley Professional, **2008**

[Kleppe2003] Kleppe, A.G., Warmer, J. & Bast, W., *MDA Explained: The Model Driven Architecture: Practice and Promise* Addison-Wesley Longman Publishing Co., Inc., **2003**

[Klint2005] Klint, P., Lämmel, R. & Verhoef, C., *Toward an engineering discipline for grammarware* ACM Trans. Softw. Eng. Methodol., ACM, **2005** , Vol. 14 (3) , pp. 331-380

[Kraus2007] Kraus, A., *Model Driven Software Engineering for Web Applications* Ludwig-Maximilians-Universität München, **2007**

[Kuipers2007] Kuipers & Visser, *Maintainability Index revisited – position paper* 11th European Conference on Software Maintenance and Reengineering, **2007**

[Kurtev2005] Kurtev, I. & van den Berg, K., *Building adaptable and reusable XML applications with model transformations* WWW, **2005** , pp. 160-169

[Kurtev2002] Kurtev, I., Bézivin, J. & Aksit, M., *Technological Spaces: An Initial Appraisal* CoopIS, DOA'2002 Federated Conferences, Industrial track, , **2002**

[Lehman2001] Lehman, M.M. & Ramil, J.F., *Rules and Tools for Software Evolution Planning and Management* Ann. Softw. Eng., J. C. Baltzer AG, Science Publishers, **2001** , Vol. 11 (1) , pp. 15-44

[Lientz1981] Lientz, B.P. & Swanson, E.B., *Problems in application software maintenance* Commun. ACM, ACM, **1981** , Vol. 24 (11) , pp. 763-769

[Lientz1978] Lientz, B.P., Swanson, E.B. & Tompkins, G.E., *Characteristics of application software maintenance* Commun. ACM, ACM, **1978** , Vol. 21 (6) , pp. 466-471

[Mahoney1990] Mahoney, M.S., *The Roots of Software Engineering* CM Quarterly, **1990** , Vol. 3 , pp. 325 - 334

[Mccabe1976] Mccabe, T.J., *A Complexity Measure* Software Engineering, IEEE Transactions on, Software Engineering, IEEE Transactions on, **1976** , Vol. SE-2 (4) , pp. 308-320

[Mellor2002] Mellor, S.J. & Balcer, M., *Executable UML: A Foundation for Model-Driven Architectures* Addison-Wesley Longman Publishing Co., Inc., **2002**

[Mens2005] Mens, T. & Van Gorp, P., *A taxonomy of model transformation* Proc. Int'l Workshop on Graph and Model Transformation, **2005**

[Mernik2005] Mernik, M., Heering, J. & Sloane, A.M., *When and how to develop domain-specific languages* ACM Comput. Surv., ACM, **2005** , Vol. 37 (4) , pp. 316-344

[Miller1956] Miller, G.A., *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information* The Psychological Review, **1956** , Vol. 63 , pp. 81-97

[Miller2003] Miller, J. & Mukerji, J., *MDA Guide Version 1.0.1* Object Management Group (OMG), **2003**

[Moreno2008] Moreno, N., Romero, J.R. & Vallecillo, A., *An Overview Of Model-Driven Web Engineering and the Mda* Web Engineering: Modelling and

Implementing Web Applications, **2008** , pp. 353-382

[Oman1994] Oman, P. & Hagemeister, J., *Construction and testing of polynomials predicting software maintainability* J. Syst. Softw., Elsevier Science Inc., **1994** , Vol. 24 (3) , pp. 251-266

[Parnas1994] Parnas, D.L., *Software aging* Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on, Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on, **1994** , pp. 279-287

[Parnas1984] Parnas, D.L., Clements, P.C. & Weiss, D.M., *The modular structure of complex systems* ICSE '84: Proceedings of the 7th international conference on Software engineering, IEEE Press, **1984** , pp. 408-417

[Pohl2005] Pohl, K., Böckle, G. & van der Linden, F.J., *Software Product Line Engineering: Foundations, Principles and Techniques* Springer-Verlag New York, Inc., **2005**

[Schmidt2006] Schmidt, D.C., *Guest Editor's Introduction: Model-Driven Engineering* Computer, IEEE Computer Society, **2006** , Vol. 39 (2) , pp. 25-31

[Seidewitz2003] Seidewitz, E., *What Models Mean* IEEE Softw., IEEE Computer Society Press, **2003** , Vol. 20 (5) , pp. 26-32

[Selic2003] Selic, B., *The Pragmatics of Model-Driven Development* IEEE Softw., IEEE Computer Society Press, **2003** , Vol. 20 (5) , pp. 19-25

[Shaw1977] Shaw, M., Wulf, W.A. & London, R.L., *Abstraction and verification in Alphard: defining and specifying iteration and generators* Commun. ACM, ACM, **1977** , Vol. 20 (8) , pp. 553-564

[Stahl2006] Stahl, T., Voelter, M. & Czarnecki, K., *Model-Driven Software Development: Technology, Engineering, Management* John Wiley & Sons, **2006**

[Taivalsaari1997] Taivalsaari, A., *Classes Versus Prototypes: Some Philosophical and Historical Observations* JOOP, **1997** , Vol. 10 (7) , pp. 44-50

[Vanderose2008] Vanderose & Habra, *Towards a generic framework for empirical studies of Model-Driven Engineering* Empirical Studies of Model-Driven Engineering (ESMDE), **2008**

[Visser2008] Visser, E., *WebDSL: A Case Study in Domain-Specific Language Engineering* Lammel, R., Saraiva, J. & Visser, J. *(ed.)*, Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Springer, **2008**

[Vliet2000] van Vliet, H., *Software engineering (2nd ed.): principles and practice* John Wiley & Sons, Inc., **2000**

[Wegner1976] Wegner, P., *Research paradigms in computer science* ICSE '76: Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press, **1976** , pp. 322-330

[Wimmer2007] Wimmer, M., Schauerhuber, A., Schwinger, W. & Kargl, H., *On the Integration of Web Modeling Languages.* Koch, N., Vallecillo, A. & Houben, G. *(ed.)*, MDWE, CEUR-WS.org, **2007** , Vol. 261

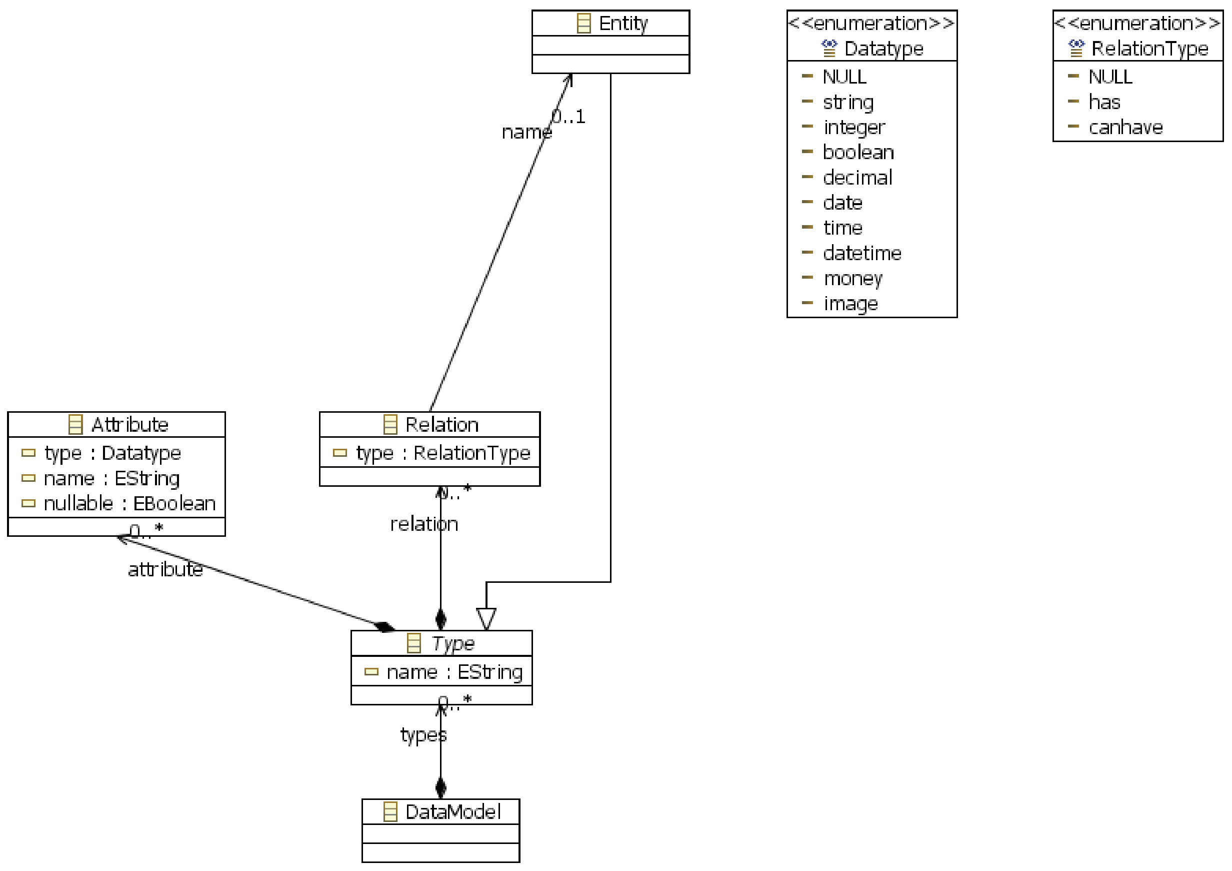
[Wirth2008] Wirth, N., *A Brief History of Software Engineering* IEEE Annals of the History of Computing, IEEE Computer Society, **2008** , Vol. 30 (3) , pp. 32-39

Online information, all links retrieved on 14-06-2009

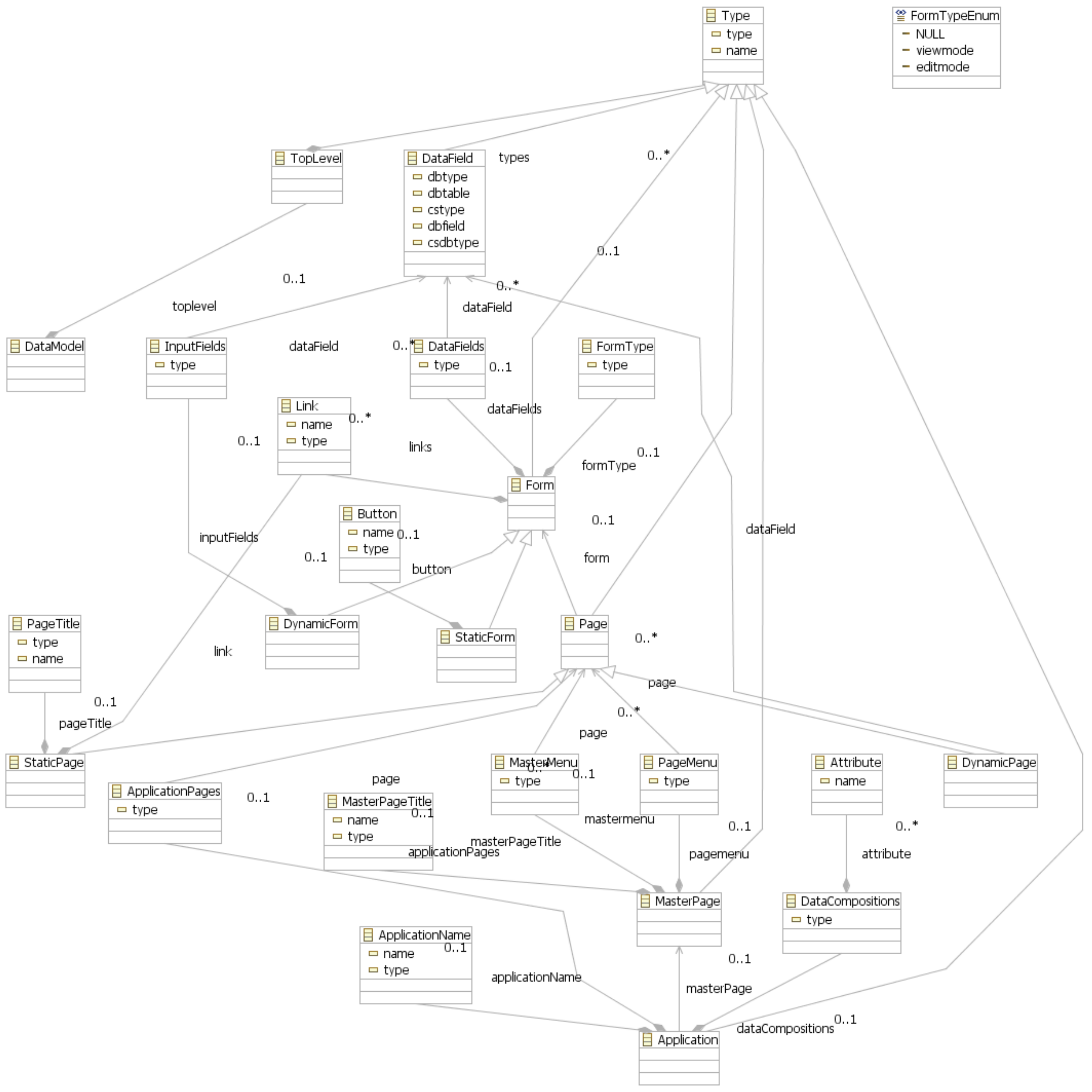
[AMPLE]	Aspect-Oriented, Model-Driven Product Line Engineering	http://ample.holos.pt/
[Andromda]	Andromda	http://www.andromda.org/
[Antlr]	Antlr	http://www.antlr.org/
[ASF+SDF]	Meta-Environment; a generalization of the ASF+SDF Meta-Environment	http://www.meta-environment.org/
[ASP_MVC]	ASP.NET MVC	http://www.asp.net/mvc/
[ATL]	Atlas Transformation Language	http://www.sciences.univ-nantes.fr/lina/atl/
[DSLTools]	Microsoft DSLTools	http://msdn.microsoft.com/en-us/library/bb126235.aspx
[EMF]	Eclipse Modeling Framework Project	http://eclipse.org/modeling/emf/
[EMP]	Eclipse Modeling Project	http://www.eclipse.org/modeling/
[Fowler2005]	Language Workbenches: The Killer-App for Domain Specific Languages?	http://martinfowler.com/articles/languageWorkbench.html
[MAPLE2009]	1st International Workshop on Model-driven Approaches in Software Product Line Engineering	http://www.lero.ie/maple2009/
[MDA_EXEC]	MDA Executive Overview 2009	http://www.omg.org/mda/executive_overview.htm
[METACASE]	Metacase	http://www.metacase.com/
[MODSE]	Model-Driven Software Evolution	http://swerl.tudelft.nl/bin/view/MoDSE/WebHome
[Nesma]	Nesma	http://www.nesma.nl/
[Northrop2004]	Achieving Product Qualities Through Software Architecture	http://www.sei.cmu.edu/architecture/cseet04.

	Practices.	pdf
[oAW]	openArchitectureWare	http://www.openarchitectureware.org/
[Petstore]	Java petstore	http://java.sun.com/developer/releases/petstore
[SEI]	Carnegie Mellon Software Engineering Institute: Software Technology Roadmap Articles:Halstead Complexity Measures, Cyclomatic Complexity, Maintainability Index Technique for Measuring Program Maintainability	http://www.sei.cmu.edu/str/str.pdf
[SEI_MDA]	Model-Driven Architecture (MDA)	http://www.sei.cmu.edu/isis/guide/technologies/mda.htm
[Stratego]	Stratego/XT	http://strategoxt.org/
[UWE]	UML based Web Engineering	http://uwe.pst.ifi.lmu.de/
[W3C]	The World Wide Web Consortium	http://www.w3.org/
[XMF]	XMF	http://www.ceteva.com/index.html
[XText]	TMF XText	http://www.eclipse.org/modeling/tmf/?project=xtext

APPENDIX A – Content DSL Metamodel



APPENDIX B – Presentation DSL Metamodel



APPENDIX C – Code Fragments

```
CONTENT: XTEXT

DataModel:
 (types+=Type)*;

Type:
 Entity;

Enum Datatype:
 string = "String" | integer = "Integer" | boolean = "Boolean" |
 decimal = "Decimal" | date = "Date" | time = "Time" | datetime = "Datetime" |
 money = "Money" | image = "Image";

Attribute:
 type = Datatype name=ID (nullable?="Nullable")?;

Enum RelationType:
	has = "Has" | canhave = "Canhave";

Relation:
 type = RelationType name = [Entity];

Entity:
 "entity" name=ID
 "{"
  (attribute+=Attribute ";")*
  (relation+=Relation";")*
 "}";
```

Code section 1 – Content XTEXT Fragment

```
CONTENT: XTEND

Field(dbdsl::Attribute this): "["+name.toLowerCase()+"] "+
                                    GetAttributeDbType()
                                    +" NOT NULL,";

FieldNULL(dbdsl::Attribute this): "["+name.toLowerCase()+"] "+
                                    GetAttributeDbType()
                                    +" NULL,";

GetAttributeDbType(dbdsl::Attribute this):
                                    switch (type.toString()){
                                            case "string"      : "[varchar] (255)"
                                            case "integer"     : "[int]"
                                            case "boolean"     : "[bit]"
                                            case "decimal"     : "[decimal](18, 0)"
                                            case "date"        : "[smalldatetime]"
                                            case "time"        : "[datetime]"
                                            case "datetime" : "[smalldatetime]"
                                            case "money"       : "[smallmoney]"
                                            case "image"       : "[image]"
                                            default            : "!unknown type!"
                                            };
```

Code section 2 - Content XTEND Fragment

```
CONTENT: XPAND

«IMPORT dbdsl»
«EXTENSION org::dbdsl::dsl::extensions::SqlServer»

«AROUND * FOR dbdsl::DataModel»
«FILE DBtablesFileName()-»
/* gen_id Database */
«FOREACH types.typeSelect(Entity) AS entity-»
«entity.BeginTable()»
«FOREACH entity.attribute AS attribute-»
        «IF attribute.nullable -»
        «attribute.FieldNULL()»
        «ELSE-»
        «attribute.Field()»
        «ENDIF-»
«ENDFOREACH-»
«FOREACH entity.relation AS relation-»
        «relation.Field()»
«ENDFOREACH-»
        «entity.MiddleTable()»
«FOREACH entity.relation AS relation-»
        «relation.ForeignKey(entity.name.toLowerCase())»
«ENDFOREACH-»
«entity.EndTable()»

«ENDFOREACH»
«ENDFILE»
«targetDef.proceed()»
«ENDAROUND»
```

Code section 3 - Content XPAND Fragment

```
CONTENT: DSL

entity Address
{
        String street;
        String zip;
        String city;
}

entity Person
{
        String lastName;
        String firstName;
        Integer phonenumber Nullable;
        String username;
        String password;

        Canhave Address;
}

entity Orderstatus
{
        String status;
}

entity Order
{
         Datetime orderdate;

         Has Person;

}
```

Code section 4 - Content DSL Fragment

```
CONTENT: GENERATED CODE - SQL

CREATE PROCEDURE GetAllAddress
AS
SELECT  [ID], [street], [zip], [city]
FROM [address]
ORDER BY 1
```

Code section 5 - Content Generated Code SQL Fragment

CONTENT: GENERATED CODE - C#

```
namespace MvcApp.Models
{
	public Address(int id, string street, string zip, string city)
			{
				ID = id;
				Street = street;
				Zip = zip;
				City = city;
			}

			public int ID
			{
				get;
				set;

			}
```

Code section 6 - Content Generated Code C# Fragment

CONTENT: GENERATED CODE - Presentation DSL

```
DataCompositions
{
	Address;
	Person;
	Orderstatus;
	Order;
	Productcategory;
	Product;
	Shoppingcart;
	Orderdetail;
	Pet;
}

DataField Address_ID{"Address", "ID", "int", "int", "SqlDbType.Int"};
DataField Address_Street{"Address", "Street", "[varchar] (255)", "string", "SqlDbType.NVarChar,
255"};
DataField Address_Zip{"Address", "Zip", "[varchar] (255)", "string", "SqlDbType.NVarChar, 255"};

DataField Address_City{"Address", "City", "[varchar] (255)", "string", "SqlDbType.NVarChar, 255"};
```

Code section 7 - Content Generated Presentation DSL SQL Fragment

PRESENTATION: XTEXT

```
DataModel:
 toplevel = TopLevel;

TopLevel:
 (types+=Type)*;

Type:
 Application | MasterPage | Page | Form | DataField;

Application:
 type = "Application" name=ID
 "{"
  applicationName=ApplicationName ";"
  masterPage=[MasterPage] ";"
  applicationPages=ApplicationPages
  dataCompositions=DataCompositions
 "}";

ApplicationName:
 type = "ApplicationName" "=" name=STRING;

MasterPage:
 type = "MasterPage" name=ID
 "{"
  masterPageTitle=MasterPageTitle ";"
  mastermenu = MasterMenu
  pagemenu = PageMenu

 "}";
```

Code section 8 – Presentation XTEXT Fragment

APPENDIX D – Introduction to Model Driven Software Development

DEFINITION, PURPOSE AND RELATED FIELDS

Model Driven Software Development (MDSD) is a software development approach in which abstract models are the primary artifacts, instead of the code in a classical approach. [Bezivin2005] compares the approach to object orientation and derives the basic MDSD principle that “Everything is a model” from the OO-principle that everything is an object. The models are used to describe applications and can be transformed into concrete implementations or directly used as such [France2007] [Selic2003][Bezivin2005]. The terms Model Driven Engineering and Model Driven Development refer to the same approach. The approach makes it possible to focus on the essence of the system and minimize the accidental complexity [Brooks1987] talks about.

The main purpose of MDSD is to improve productivity by reducing development effort. MDSD offers simplification by abstraction and automation to realize this. Models define what is variable in a system, and code generators produce the functionality that is common in the application domain [Deursen2007]. Besides productivity gain, also development costs and time to market can be reduced, and quality can be improved. The approach basically tries to offer a better way to deal with complexity.

MDSD has many related fields, e.g. software product lines, generative programming and domain specific languages. Looking at [Pohl2005] and [Czarnecki1998] one sees a clear overlap between these paradigms, sharing terms, practices and goals. The related fields are often combined, e.g. the workshop [MAPLE2009] focuses on Model-driven Approaches in Software Product Line Engineering.

[Bezivin2005] relates MDSD to language engineering and ontology engineering. Also a clear relation exists between modelware and grammarware, so called technical spaces [Kurtev2002]. [Klint2005] points out that grammarware engineering could be seen as an instance of MDSD, but differ in aspirations.

MODELS

The primary artifacts in MDSD are abstract models. Models are used as abstractions.

> **Abstraction**
>
> A view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information.
>
> [IEEE Std 610.12-1990]

In order to focus on some aspect of a system, or "separate a concern", other aspects must be left out or hidden. In this context [Booch1993] quotes [Shaw1977]: *We (humans) have developed an exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirely of an object, we choose to ignore its essential details, dealing with the generalized, idealized model of the object.* [Shaw1977] describes the use of abstraction in order to focus on certain aspects, a powerful technique in dealing with complexity. Besides this focus models also provide a means to separate concerns and add structure like hierarchies.

> **Model**
>
> A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language. [Miller2003]
>
> A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. [Bezivin2001]
>
> A model is a set of statements about some system under study. [Seidewitz2003]

Models are used throughout history for various purposes, e.g. paintings or roadmaps. Examples of models showing different concerns or views of one subject are blueprints of a house showing wiring or plumbing, or maps of a landscape showing altitudes, sorts of ground or climates. Within software development modeling has been used extensively to express parts of the system. Examples of languages used for this purpose are the Unified Modeling Language (UML), Archimate and the Business Modeling Notation (BPMN).

Models can be used in various ways. For example in [Fowler2003] Martin Fowler describes the use of models for sketching, blueprinting and programming. The different purposes have different levels of formality. In MDSD the models are the system, therefore the models must be formal enough to be interpreted by a computer. The models are used to "program" the application.

A models representation can be graphical, textual or a combination of both. Source code can also be seen as a model [Mens2005][Kleppe2008] [Favre2004]. Kleppe introduces the term "Mogram" to describe a product written in a software language. As programming or modeling languages are described by their grammar/metamodel, these also can be seen as models [Favre2004].

MODELING LANGUAGE

To create models modeling languages are required. Programming languages are usually textual and consist of an abstract syntax, a concrete syntax and semantics. Semantics are often not formally defines, and the abstract and concrete syntax are often described intertwined within a grammar, e.g. in a context free grammar. Textual versus graphical language specification formalisms can be categorized as tree versus graph formalisms. Modeling languages are more expressive when able to use both graphical and textual notations. A separation of abstract and concrete syntax is recommended, because different notations for different purposes add a lot of value. The metamodeling formalism provides a way to do this. For more information on this subject, see [Kleppe2008]. Kleppe states that a **metamodel** specifies a modeling language, and is usually the model of an abstract syntax.

Language descriptions are written in languages, e.g. EBNF. These languages are capable to describe languages and so are capable of describing themselves. In MDSD these languages are called **metametamodels**. [Bezivin2005] and [Favre2004] introduced the term **megamodel** to describe the model that uses concepts as model, metamodel and metametamodel, see figure 1. The layering in the picture is based on the four-layered architecture of the MOF standard from the OMG.

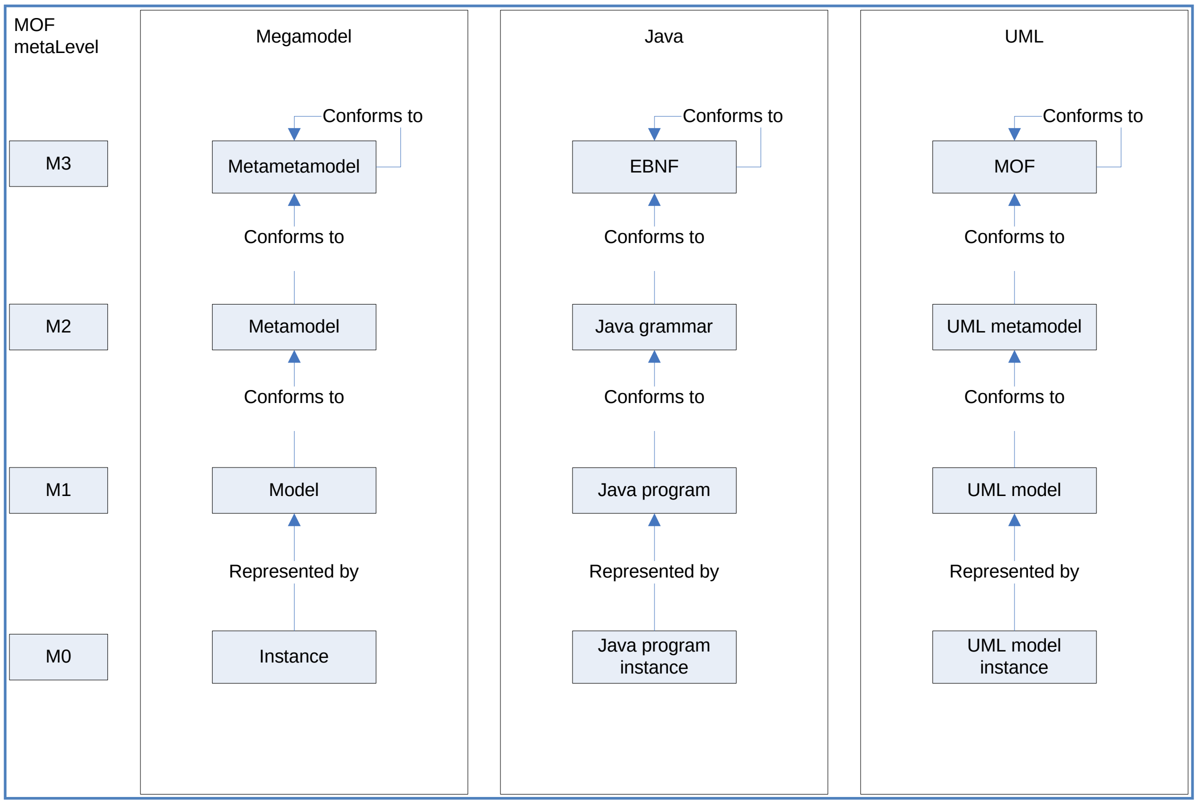


Figure 1 - Metamodeling architecture

TRANSFORMATION

In MDSD models can be transformed to models by applying transformations, forming the automation part of the paradigm. One model can be transformed to one or multiple models, and multiple models can be transformed into multiple models or one. The basic model transformation pattern used in MDA is shown below in figure 2.

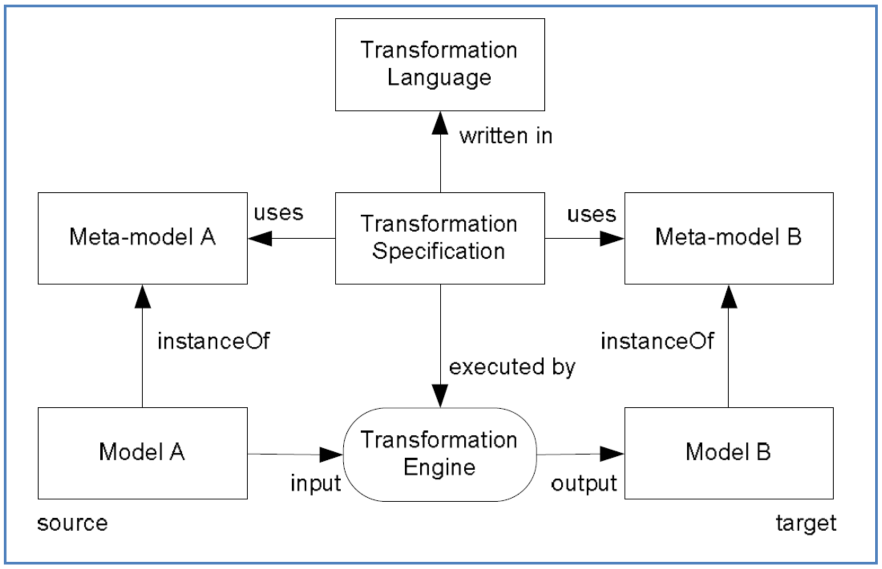
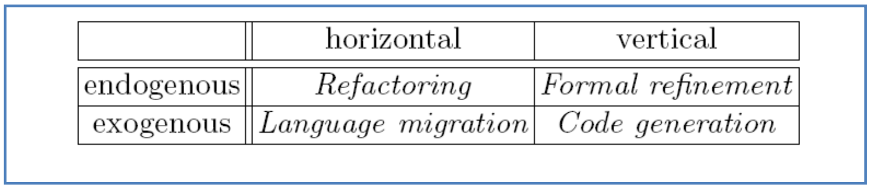


Figure 2 – MDA Transformation Pattern [Kurtev2005]

In MDSD practice most often the distinction is made between model and code, leading to the use of model to model transformations (m2m) and model to text/code transformations (m2t). The latter term is equal to code generation. The distinction between the two transformations comes down to whether or not a metamodel is used. In m2m transformations often a mapping between metamodels is used, as in m2t transformations this is mostly not the case, making the use of the metamodel implicit. One of the reasons for not using the metamodel might be because it is unavailable. Another reason might be that the use of transformation templates might be more pragmatic than explicitly mapping the metamodels.

[Mens2005] describes a taxonomy of model transformations with the dimensions as in figure 3.



	horizontal	vertical
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language migration*	*Code generation*

Figure 3 – Model Transformation Dimensions

Distinctions are made between horizontal and vertical transformations. A horizontal transformation transforms on the same abstraction level, a vertical transformation transforms to a different abstraction level. Also a separation between endogenous and exogenous is applied. Endogenous refers to transformations between models with different metamodels, exogenous to transformations between models with the same metamodel. Examples of the types of transformations are given in figure 3.

To realize these transformations languages are created solely for this purpose. Transformation languages are programming languages most often of the declarative kind. For m2m transformations examples are Stratego [Stratego], XTend [oAW], ATL [ATL] and implementations of OMG's QVT. For m2t transformations examples are XPand [oAW], JET and Acceleo [EMP].

DOMAIN SPECIFIC LANGUAGES

Domain specific language (DSL)

A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

[Deursen2000]

A DSL is the opposite of a general purpose language (GPL). DSLs are more specific and GPL's more generic. These are relative and subjective terms as same goes for the term domain. No formal specification can be given to determine the category a language falls into.

By scoping on a domain, its concepts can become language components. An example is music notation. The focus on the domain makes it possible to use a concise and easy interpretable notation instead. Music written down in plain English would be much harder to interpret. The notation of a DSL can be both graphical and textual, or a combination.

[Mernik2005] states that *"Many computer languages are domain-specific rather than general purpose. Domain-specific languages (DSLs) are also called application-oriented, special purpose, task-specific, problem-oriented, specialized, or application languages. DSLs trade generality for expressiveness in a limited domain."*

Examples of well-known DSLs are:

- TEX
- BNF
- HTML
- SQL
- XML

Within the field of MDSD examples can be found in transformation languages as ATL or XPand, Constraint languages as OCL or Check, metamodel specification languages like MOF or KM3, metadata-interchange with XMI and so one.

DSLs can be concise, enable reuse, can enhance productivity, reliability and maintenance, and improve communication with domain experts. On the other side the cost of designing, implementing and maintaining a DSL and educating the users can be high, especially because of the smaller user group compared to a GPL. [Mernik2005] [Deursen2000][Deursen1997]

Prerequisites for building a DSL are language development expertise, domain knowledge and a considerable code base for systems in the domain. [Mernik2005][Visser2008]

[Kleppe2008] distincts horizontal and vertical DSLs, where horizontal stands for a broad and technical orientation, whereas vertical complies with a smaller orientation within the business domain. The web application domain falls into the first category.

[Visser2008] describes two approaches of DSL development; deductive and inductive. The first is a top-down approach with an exhaustive domain analysis phase. The DSL is first fully designed, after which it is implemented. The second approach incrementally introduces abstractions, enabling a quick turn-around time for the development of such abstractions.

DSLs can be external or internal [Fowler2005]. External DSLs stand alone, internal DSL are embedded in a base language. Both types have their strengths and weaknesses. The external DSL offers freedom of implementation making high expressiveness possible, as well as meaningful error checking. This freedom comes at a high price as everything needs to be built from scratch. Building internal DSLs is cheaper because of the reuse. And because it uses the base language's syntax and structure the learning curve is shortened for developers who know the base language. But options are also limited by this fact, and errors are checked at the base language's level. For users who aren't familiar with the base language and the used IDE the learning curve might become higher.

In comparing DSLs to frameworks or API's [Kleppe2008] explains that all of them provide abstract concepts, often at a higher level of abstraction. But the DSL differs in the fact that it also provides a new syntax, which brings better options in dealing with complexity.

TOOLING

MDSD approaches can be realized in many different ways. For example models can be created using internal DSLs or external DSLs. A DSL's concrete syntax can be defined by a GPL like UML, e.g. by using profiles, or can be user defined. This can be graphical, textual or both.

Examples of MDSD tooling with a graphical focus are:

- AndroMDA
- Eclipse EMF/GMF
- Microsoft DSLTools
- MetaEdit+

The first two are UML based; the others make use of a user defined language.

Internal DSLs are realized within a base language, e.g. Ruby is a well suited base language for this purpose. The syntax and structure of the base language limits the DSLs possibilities [**Fowler2005]**. Therefore internal DSLs are mostly textual.

For external textual DSLs several tooling exists, e.g.:

- ANTLR
- ASF+SDF
- XMF
- openArchitectureWare(oAW)

The choice is made to use oAW for this project because of the following. According to [Gharavi2008] oAW is currently one of the leading open-sourceMDA generator frameworks. It is very extensible and supports model-to-model and model-to-text transformations. Also it provides all that is needed for a textual MDSD approach in one package.

For a textual MDSD approach oAW offers the following languages:

- XText (T2M)
- XTend (M2M)
- XPand (M2T)
- Check (constraints)
- Workflow (sequencing)

With XText an EBNF grammar can be written for a DSL from which an editor can be generated. This editor provides default features as:

- syntax highlighting
- code completion
- code folding
- error decoration

In this editor a model can be created. Check can be used for model validation. XPand is a template language that can be used to transform the model to code. XTend can be used for model to model transformations. The Workflow language coordinates the total transformation.

OAW also provides tooling for product line engineering and a recipe framework for programmer guidance. Because OAW is very extendible, it can use any concrete syntax, whether graphical or textual.

MODEL DRIVEN ARCHITECTURE

According to [Bezivin2005] MDA may be defined as the realization of model driven engineering principles around a set of OMG standards.

The Model Driven Architecture (MDA) provides an open, vendor-neutral approach to the challenge of interoperability, building upon and leveraging the value of OMG's established modeling standards: Unified Modeling Language (UML); Meta-Object Facility (MOF); and Common Warehouse Metamodel (CWM). Platform-independent Application descriptions built using these modeling standards can be realized using any major open or proprietary platform, including CORBA, Java, .NET, XMI/XML, and Web-based platforms.

[MDA_EXEC]

The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.

MDA provides an approach for and enables tools to be provided for:

- *specifying a system independently of the platform that supports it*
- *specifying platforms*
- *choosing a particular platform for the system*
- *transforming the system specification into one for a particular platform*

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns.

[Miller2003]

The goal of MDA is one that is often sought: to separate business and application logic from its underlying execution platform technology so that (1) changes in the underlying platform do not affect existing applications, and (2) business logic can evolve independently from the underlying technology

[SEI_MDA]

MDA uses three models:

- **CIM** – Computational Independent Model for describing the requirements
- **PIM** – Platform Independent Model for describing the system technology independent
- **PSM** – Platform Specific Model for describing the system technology dependent

MDA uses transformation rules to transform one model to the next, as shown in figure 4. In practice the CIM is often not formally used. The major steps in the MDA development process are the transformations from PIM to PSM and from PSM to code [Kleppe2003].

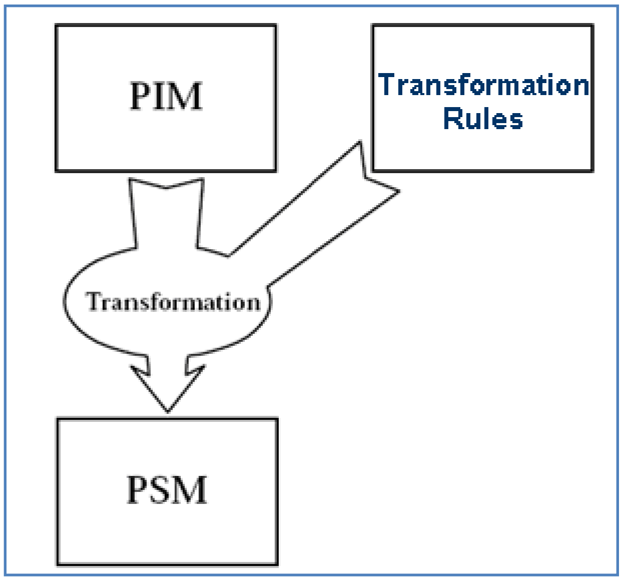


Figure 5 - MDA Model Transformation Process [SEI_MDA]

[Bezivin2005] explains that the real power of the MDA comes from the existence of level M3, the meta-metamodel level, see figure 1. This allows to build coordination between models, based on different metamodels, e.g. in the form of model transformations or model weaving.

The standards of the OMG play a major role in MDSD tooling. For example the Unified Modeling Language (UML), based on the MOF, is widely used for modeling, as well as XMI is used a lot for interchanging models. Other initiatives are undertaken on the basis of OMG's work. For example ATL of INRIA was developed to answer the QVT Request For Proposal. And the Eclipse Modeling Framework (EMF) started out as an implementation of the MOF, which lead to Eclipse's Ecore, a metametamodel which is used by many tools.

The OMG did put MDSD on the map by its MDA initiative. But there is also a lot of criticism on their work; Criticism on the standards, like the size and the lack of semantics of UML2.0 [France2006] [Hailpern2006][Greenfield2004] or the complexity of the MOF, on the changing of the standards (XMI), and also on the focus and principles of the approach. For example in [Stahl2006] the writers favor an approach which aims at increasing development efficiency, software quality, and reusability instead. The approach is focused on infrastructure code generation. [Greenfield2004] provides a product line based bottom up approach, where MDA is a more top down one or many approach.

The MDA is interesting because it is a major influence in the field and it provides some useful thoughts and terminology. But the standards and vision do not fit the purpose of this project. The project focuses on changeability, and complying with the MDA would probably influence this in a negative way. Standardization will come at the cost of flexibility. For example using the MOF will provide a lot of overhead, and MDA standards will restrict choices for tooling.

MDSD VS CASE TOOLS AND 4GL

MDSD and the Computer Aided Software Engineering (CASE) tools of the 80's seem to have a lot in common, both using abstraction and automation to industrialize software engineering. Case tools failed to live up to their promise. Why did CASE fail and in what way is MDSD different and will it stand a better chance of succeeding? And according to [Mernik2005] 4GL are usually DSLs for database applications. In what way do these languages differ from a MDSD approach?

[Stahl2006] points out that CASE adhered to the "one size fits all" dogma, leading to inflexible tooling. [Greenfield] mentions among others the low quality of generated code, caused by not taking advantage of platform-specific features. [Schmidt2006] points to the same two issues. He also mentions scalability problems by not supporting concurrent engineering and the targeting of proprietary execution environments. The issues with CASE seem to revolve around to a bad fit between the models and the application domains.

[Schmidt2006] describes that the difference between CASE and MDSD tooling lies in the provisioning of ways to tailor notations, transformations and validations specific to a domain, as can be seen in modern MDSD approaches as MDA [Miller2003], Software Factories [Greenfield2004] and MDSD [Stahl2006]. The big difference between CASE and MDSD is the shift in focus form a general to a domain specific perspective.

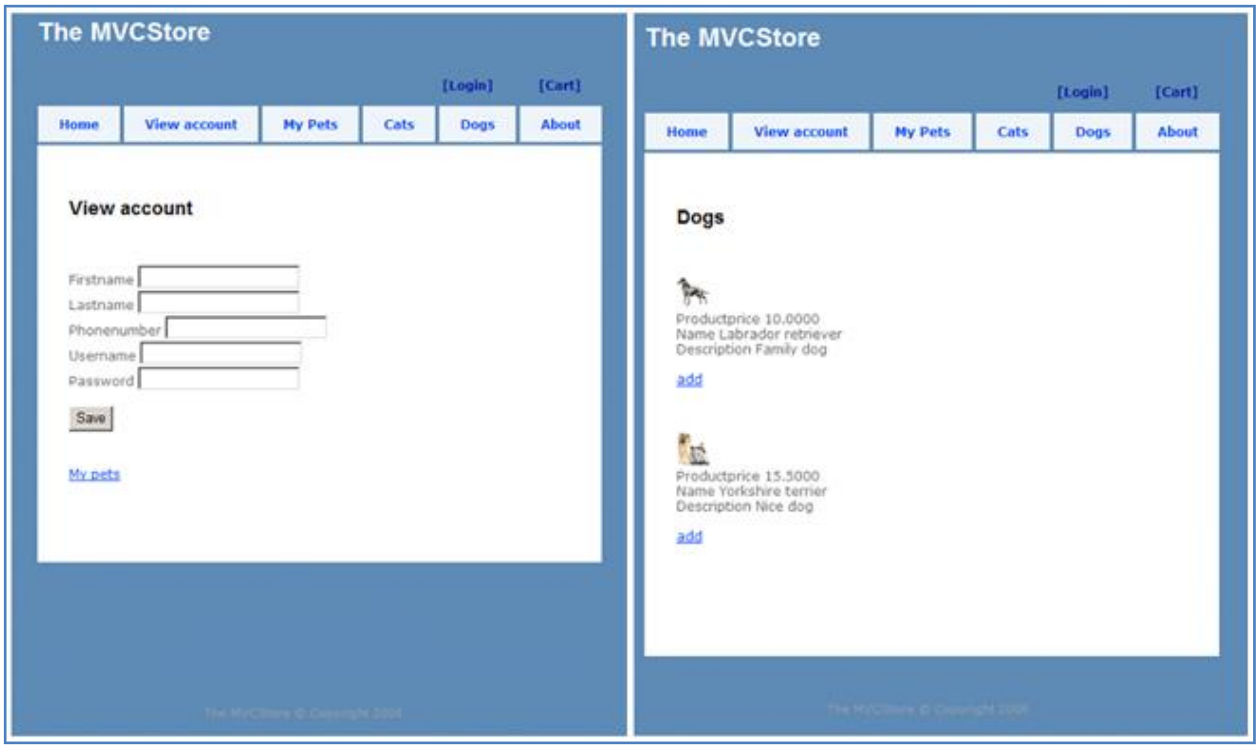
[Deursen2007] states that "*What distinguishes MDE from the 'fourth-generation' and domain-specific languages of the past is the aim at a systematic approach, with supporting technologies, to the construction of models and modeling languages such that these activities can be undertaken by the 'average' software developer and integrated in the software development process*."

MDSD AND EVOLUTION

While MDSD simplifies the “coding” of the system, complexity is added through the introduction of a multi dimensional evolution. [Deursen2007] defines the dimensions as follows:

*In **regular evolution**, the modeling language is used to make the changes. In **metamodel evolution**, changes are required to the modeling notation. In **platform evolution**, the code generators and application framework change to reflect new requirements on the target platform. Finally, in **abstraction evolution**, new modeling languages are added to the set of (modeling) languages to reflect increased understanding of a technical or business domain.*

APPENDIX E – User Interface



View account

Firstname
Lastname
Phonenumber
Username
Password
Save

My pets

The MVCStore © Copyright 2008

Dogs

Productprice 10.0000
Name Labrador retriever
Description Family dog
add

Productprice 15.5000
Name Yorkshire terrier
Description Nice dog
add

The MVCStore © Copyright 2008