UNIVERSITY OF OSLO
Department of Informatics

A practical evaluation of Software Factories and the Microsoft Domain Specific Language approach for developing Web Services

Master thesis

Erik Bråthen

22nd December 2005

Preface

This thesis has been written to fulfill my Master degree at the University of Oslo (UiO), Department of Informatics (IFI). The thesis has been written for SINTEF, Department of Information and Communication Technology, Cooperative and Trusted Systems. The work on this thesis has also been done at SINTEF.

I would like to thank my supervisor Dr. Arne-Jørgen Berre for his guidance through the entire process and especially in the final phase. I would also like to thank my fellow student Tuva Hassel Stang for giving me guidance on the writing process.

Erik Bråthen

Oslo,
December 22, 2005

Contents

1. INTRODUCTION 1

1.1. PROBLEM SCOPE 1
1.2. BACKGROUND AND MOTIVE 2
1.3. GOAL 2
1.4. STRUCTURE OF THIS THESIS 3

2. TECHNOLOGIES 4

2.1. MODEL DRIVEN DEVELOPMENT (MDD) 4
2.1.1. What is Model-driven development? 4
2.1.2. Defining models 5
2.1.3. Generating software 5
2.1.4. Automating 6
2.2. SOFTWARE FACTORIES 6
2.2.1. The Software Factory idea 7
2.2.2. What is a Software Factory? 8
2.2.2.1. Software Factory schemas 8
2.2.2.2. Product development 11
2.2.2.3. Mechanisms 11
2.2.2.4. Mass customization 12
2.2.3. Raising the level of abstraction 12
2.2.3.1. Abstraction 12
2.2.4. Reducing complexity 12
2.2.4.1. Problem of complexity 13
2.2.5. How to model software 13
2.3. DOMAIN SPECIFIC LANGUAGES AND TOOLS 14
2.3.1. Domain 15
2.3.2. Design 15
2.3.3. Creating domain specific languages 16
2.3.4. Automation 16
2.3.5. How does Domain Specific Models differ from plain UML models? 17
2.4. MODEL TO TEXT TRANSFORMATION 17
2.4.1. Custom Tool 17
2.4.2. Custom code 18
2.4.3. Custom code syntax 18
2.4.3.1. Control blocks 19
2.4.3.2. Standard directives 19
2.4.4. Transformation 20
2.5. VISUAL STUDIO 2005 20
2.5.1. DSL designer 20
2.6. WEB SERVICES 21
2.6.1. Basic concepts on web services and service oriented architecture (SOA) 22
2.6.1.1. Service 22
2.6.2. Usability and possibilities 22
2.6.3. SOAP 23
2.6.4. WSDL 23

3. DSL TRANSFORMATION REQUIREMENTS 24

3.1. MODEL TO MODEL REQUIREMENTS 24
3.1.1. Design 24
3.1.2. Specifications 25
3.1.3. Synchronization 25
3.1.4. Transformation 25

3.2. MODEL TO TEXT REQUIREMENTS 25
3.2.1. DSL tools text templating *26*
3.2.2. Reference to models *26*
3.3. OTHER POSSIBILITIES 26
3.3.1. Shape inside shape *26*

4. SOFTWARE FACTORIES AND DSL TOOLS WALKTHROUGH 27

4.1. RAISING THE LEVEL OF ABSTRACTION 27
4.1.1. Models *27*
4.1.2. Views *27*
4.2. REUSABILITY 27
4.3. DEALING WITH CHANGE 28
4.4. ERROR DETECTION ON DESIGN LEVEL 28
4.4.1. System logics *28*
4.5. REDUCING PRODUCTIVITY TIME 29
4.5.1. Application specific *29*
4.6. INCREASING RELIABILITY 29
4.7. INCREASING SECURITY 30

5. DEFINING DOMAIN SPECIFIC LANGUAGES AND TOOLS 31

5.1. DSL IN SHORT 31
5.2. VS 2005 DSL DESIGNER 31
5.2.1. Class *31*
5.2.2. Relationships *32*
5.2.2.1. Embedding 33
5.2.2.2. Reference 33
5.2.2.3. Inheritance 34
5.2.3. Models *34*
5.2.3.1. Shapes 34
5.2.3.2. Compartment models 35
5.2.4. Connectors *36*
5.2.4.1. Allowed models 37
5.2.4.2. Multiple connections 37
5.2.5. Enumerations *37*
5.3. DSL TOOLBOX 38
5.3.1. Names, Bitmaps and order *38*
5.3.2. Transformation *39*
5.3.2.1. Models and XML files 39
5.4. DSL DEPLOYMENT 39
5.4.1. Export Template *39*
5.4.2. Domain Specific Language Setup *40*
5.4.3. Install DSL *40*
5.5. “HELPER PLUG-INS” 41
5.5.1. DSL Dm->Dd *41*
5.5.2. T3Colorizer *42*

6. TRANSFORMING DSL MODELS TO USABLE WEB SERVICES 44

6.1. MODEL TO TEXT TRANSFORMATION 45
6.1.1. Templates *45*
6.1.1.1. Custom code 45
6.1.1.2. Defining output files 46
6.1.1.3. Multiple templates generating one solution 46
6.1.2. Model compartments *47*
6.2. MODEL EXPLORER 47
6.3. USING THE DOMAIN MODEL DESIGN IN OTHER PROJECTS 48
6.4. COMPILING GENERATED CODE 48

7. CASE STUDY 49

7.1. CASE DESCRIPTION 49
7.2. SCOPE 51
7.2.1. Requirements 51
7.2.2. Limitations 51
7.3. DESIGNING THE DOMAIN MODELS 52
7.3.1. Defining detail level 53
7.3.2. Compartment shapes 54
7.3.3. Connectors 54
7.4. USING THE DESIGNED MODELS 55
7.4.1. Designing a system 55
7.4.2. Adding methods and other application specific data 56
7.5. "PROGRAMMING" THE CODE GENERATORS 56
7.5.1. Custom tool 57
7.5.2. Custom code and Templates 57
7.5.3. Output 58
7.5.4. Getting compileable code 58
7.6. RESULTS 59
7.6.1. Web service (.asmx) 59
7.6.2. "Code behind" 59
7.6.3. WSDL 59
7.6.4. Other C# code 60
7.7. FULFILLING THE PROJECT 60
7.8. OTHER MICROSOFT DSL PROJECTS 60

8. OTHER SIMILAR TECHNOLOGIES 62

8.1. MODEL DRIVEN ARCHITECTURE (MDA) 62
8.1.1. Basic concepts 62
8.1.2. MDA vs. Software Factories and DSL tools 63
8.1.2.1. UML and MOF 64
8.1.2.2. Defining models 64
8.1.2.3. Platform 64
8.1.2.4. Model to text transformation 65
8.2. RAPID APPLICATION DEVELOPMENT (RAD) 66
8.2.1. RAD vs. Software Factories 66
8.3. UNIFIED SOFTWARE DEVELOPMENT PROCESS (UP) 66
8.3.1. UP vs. Software Factories 67
8.4. AGILE MODELING 67
8.4.1. AM vs. Software Factories 69
8.5. CODE TEMPLATES 69
8.6. METAEDIT+ 70

9. EVALUATION OF RESULTS 71

9.1. DOMAIN SPECIFIC LANGUAGES 72
9.2. SOFTWARE FACTORIES 72
9.3. DOMAIN MODEL DESIGN 73
9.4. USING DSL TOOLS 74
9.5. SOFTWARE FACTORIES AND DSL TOOLS VS. OTHER SOFTWARE DEVELOPMENT LANGUAGES AND TOOLS 74
9.6. DESIGNING WEB SERVICES WITH DSL 74
9.7. QUESTIONS AND ANSWERS 75
9.8. PROPOSALS FOR IMPROVEMENT 77
9.8.1. Two-way synchronization 77
9.8.2. Nested shapes 77
9.8.3. Custom code editor 77

10. CONCLUSIONS AND FUTURE WORK **78**

A. APPENDIX **80**

A.1. DOMAIN SPECIFIC LANGUAGE DESIGNER 81
A.1.1. The Domain Specific Language Design *81*
A.1.2. Domain Explorer *82*
A.1.3. Domain Model Designer XML code (designer.dsldd) *83*
A.2. DESIGNING WITH DOMAIN MODELS 93
A.2.1. Visual design *93*
A.2.2.2. WebServiceInteractivity *94*
A.3. CUSTOM CODE 95
A.3.1. Custom code for asmx files *95*
A.3.2. Code behind custom code *96*
A.3.3. Custom code for the rest of the modeling *97*
A.4. GENERATED FILES 99
A.4.1. No1OrderService asmx file *99*
A.4.2. No2OrderService asmx file *99*
A.4.3. TransportService asmx file *99*
A.4.4. Code behind classes *100*
A.4.5. Other generated code *101*

B. BIBLIOGRAPHY **103**

List of figures

Figure 1: Software Factory, collection of tools, languages and items 7
Figure 2: Software Factory Schema, Grid approach 9
Figure 3: Software Factory Schema, Graph approach 10
Figure 4: DSL, Increased productivity graph 14
Figure 5: Domains. Domains can overlap domains 15
Figure 6: Custom Tool 18
Figure 7: Custom code, file extension 18
Figure 8: Custom code, generated file 19
Figure 9: Custom code, Standard directives 20
Figure 10: Visual Studio 2005 Domain model designer 21
Figure 11: Domain model designer to a DSL model 24
Figure 12: VS DSL Designer, XML root 32
Figure 13: DSL Designer, relationships 33
Figure 14: DSL Designer, defining shape 34
Figure 15: DSL Designer, Compartment shapes 35
Figure 16: DSL Designer, Defining connectors. XML 36
Figure 17: DSL Designer, defining allowed connections 37
Figure 18: DSL Designer, defining toolbox 38
Figure 19: DSL Designer, Resource file 38
Figure 20: DSL Deployment, Exporting Templates 40
Figure 21: Installing the DSL 41
Figure 22: DSL Dm->Dd helper tool 42
Figure 23: T3Colorizer helper tool 42
Figure 24: Inline Web Service 44
Figure 25: Code behind Web Service 44
Figure 26: DSL Web Service design 45
Figure 27: Template files 46
Figure 28: Model Compartments 47
Figure 29: Modeling explorer 47
Figure 30: Consumer/producer relationship 50
Figure 31: DSL Design, Case 52
Figure 32: Compartment shapes 54
Figure 33: Different connections 55
Figure 34: Using models to build the case 56
Figure 35: Custom code, template generated files 57
Figure 36: Code behind Web Service 57
Figure 37: Generated Web Service code 58
Figure 38: Complexity against precision and functionality graph 71

Abstract

Software development has reached a new step in the evolution of software developing tools and languages. From the beginning of software development there has been major changes in how software is developed. From the basic of assembler programming, where every line of code would result in just a few CPU cycles, to programming with models, where just a few models could represent a whole application. The demand for new and more complex applications have been ever increasing, and the need to evolve new and better programming languages and tools has been needed to cover this demand. Many think that this “new step” has come with the so called Model-driven Development.

This thesis will try to explain the concept of model-driven development with a special focus on Domain Specific Languages in Visual Studio 2005 and how to produce Web Services in an easy and effective way. Microsoft’s idea on Software Factories, which takes the evolvement of software development even a step further, will also get a great deal of focus. Towards the end of this thesis I will also discuss other similar technologies which have their main focus on speeding up the development process.

1. Introduction

"*We should not build the future by piling on more tasks on people. Moore's Law improves because greater portions of the process are mechanized. We just need to push the boundary between manual and mechanical effort upstream which can be done only if we preserve the design intent of the subject matter experts*"

-Charles Simonyi

1.1. Problem scope

The world of software development is experiencing an ever increasing demand for software. Hiring more developers to deal with this is expensive, so software developers are counting on new development technologies to increase their productivity rate. There are many new technologies out there, and this thesis will try to explain the concepts of some of these technologies, with a special focus on Software Factories and Domain Specific Languages (DSL) for developing Web Services. How can Web Services be created with the use of DSLs, and is it the really the way to go? These questions should be answered through an actual implementation with the use of Microsoft's DSL Designer and tools. This case will also be one of the main parts in this thesis.

Questions that should be answered:

What is Model-driven Development (MDD) and what makes MDD so attractive for developers?

- How can we model software?
- Why is it important with an abstracting view when developing software?

What is the concept of Software Factories?

- How do we produce software with Software Factories?
- Can Software Factories be realized?

What is DSL?

- What are domains?
- How do we develop software with DSLs?
- How should DSLs be described?
- What are the differences between DSL and Model Driven Architecture (MDA) from Object Management Group (OMG)?
- How can Web Services be developed with the help of DSL?
- Is DSLs the way to go when developing Web Services?
- What is the future of DSL in Visual Studio 2005?

These questions are answered and summarized in chapter 9.7

1.2. Background and motive

Since the beginning of software development the complexities of the applications made have dramatically increased. As the complexity has increased there has been made counter measurements in the way of better programming languages and tools, letting the applications be more complex itself without getting more complex for the developer. Reducing the complexity on developing level also reduces the time needed to create applications, which is often a big problem today. Building a house starting at cutting down threes takes a lot longer time then getting finished modules that only needs to be nailed together. It might be slightly more costly, but the time to build the house will be so reduced that the cost will be the same or even less. The same thing can be applied to development of applications. The cost might here be seen as the efficiency or "speed" of the application. As hardware gets faster and faster this is not really a problem, and if we need to make some parts of the application optimized we can do this with the use of low level programming languages like assembler or C. Compilers for higher level of programming languages is also getting better, making the difference in the cost of optimization reduced greatly.

Lately there has been emerging a new kind of development method that tries to reduce the complexity even more by increasing the level of abstraction all the way up to using models when developing applications. This type of developing software with the use of models is called Model-driven Development (MDD). The basic concept of MDD is that the models can be transformed into text or code, and combinations of these models will result in an application.

The whole idea of using Model-driven development and other similar concepts is to save time developing our applications, avoid inventing the wheel over and over, reducing the complexity and making it easier to maintain applications after they are made. These are all great promises, but how does it really work when implementing the concepts of MDD into real languages and tools?

Object Management Group (OBG) and Microsoft have created some tools that try to implement the ideas of MDD. OMG has created Model Driven Architecture (MDA) which uses UML as a base for the modeling. Microsoft has implemented something that is called Domain Specific Language (DSL) Designer in their Visual Studio 2005 and the main focus in this thesis will be on the way that Microsoft implements Model-driven development.

1.3. Goal

The goal of this thesis is to figure out how MDD really works when looking at Microsoft's Visual Studio 2005 and their DSL Designer. Web services has the last few years been widely implemented, and as almost every application made nowadays have some kind of connection to the internet, whether it is an automatic update or search on the internet they need a good and reliable way of sending date over the internet. Web

services are a collection of several files and therefore it is a nice way of seeing how we can implement this in MDD. How to do this will also be a discussed in this thesis.

The main goal will be to evaluate Microsoft's Domain Specific Language approach in the case of MDD and evaluate how to use DSL's when developing Web services. There are many "new" ways of getting a faster and better way of developing software. Some of these "tools" for developing software have some great advantages, while others might seem to get you to the goal, but when you have reached the goal what then, maintenances. How DSL's cope with the problems that exist in software development is the one of the main goals to explore. We will also touch into how other similar tools or languages solve these problems, and how DSL's differ from these.

Questions that also should be answered are:

- What is the concept of Software Factories?
- Where is DSL and tools today and how does it work?
- Is developing Web services with DSL and tools the right way to go?
- When and where should we use MDD and especially DSL and tools?
- What other alternatives is there out there?

1.4. Structure of this thesis

This thesis mainly builds up towards a case of where DSL is used to design Web Services that interact with different companies. The thesis is mainly divided into these parts as follows:

Chapter 1 – 5 gives an introduction of the tools and concepts that is being used in the case. Here we will get an introduction to the development concepts Model-driven Development, Software Factories and Domain Specific Languages.

Chapter 6 will show how DSL models can be transformed into Web Services with the help of transformation scripts

Chapter 7 the thesis is written towards this chapter where the technologies and concepts that have been described in the chapters before are used to develop software. This case study will describe how DSL can be used to develop Web Services that is used by other companies.

Chapter 8 gives an introduction to the some of many other technologies that are out there that promise faster and better ways of developing software than today.

Chapter 9 – 10, this last part will try to summarize the work that has been done throughout this thesis and hopefully answer the questions that have been raised. My experience with the tools and languages will also be discussed.

2. Technologies

This section will focus on the technologies that are needed for creating applications with MDD in a Microsoft approach. Web services are also a large part of this thesis and therefore there will be some focus on the technologies behind Web services as well.

2.1. Model driven development (MDD)

Inventing the same thing over and over again is as everyone knows, not a very productive way of producing anything. The same thing can often be applied to software development, thus here we seem to invent the wheel over and over again. Software developers have finally seen this and are trying to do something to solve these issues. Model-driven development is not a new idea, but the technology has been a bit complex to evolve, it's just in the last few years we have seen "good" tools or languages that tries to do something with these problems.

2.1.1. What is Model-driven development?

Model-driven development is, as the name says, development of software with the use of models. Instead of using text you can draw you applications with the use of boxes, circles, connectors and so on.

UML has for a time now been used by software engineers to draw sketches of the system or parts of the system they are going to implement. This has been useful since it's much easier to see how a system works when getting a more abstract view than just plain code. Models normally give us a very good description of how software is built because we normally program software in modularized way. The problem with UML is not really a problem with UML itself, but the way that the implementation of the models normally results in a redesign of the modeling cause of discovered faults or impossibilities to implement as code. Since the models often only are used as a design of the system it's not many that will draw new models of the complete and finished system.

Model-driven development tries to address this problem by making modeling the main part of the development. Changes on the applications means changes to the modeling, making the models up-to-date at all stages.

An issue with model-driven development is whether to allow two-way synchronization or not. Meaning if we change something on the generated code shall these changes be displayed and added to the models. Is it even possible or at least some good ways to a chive this?

2.1.2. Defining models

One of the important aspects in MDD is how the models are defined. The definitions of the models need to be good and consistent so that it can easily be used for generating code.

Models can be easily made, but how to transform them into something useful is another matter. Somehow we need to be able to get a "grip" on the models and read what they are containing. How this is done is one of the important parts that let MDD be possible. So we need something that can define the models better than just the models them self's that we can get hold of when we want to transform the models into code later on. This "something" is normally some kind of text that is connected to the code. The text is often some kind of text that can be easily read by humans and computers, and therefore XML [XML03] is often used.

Defining how models can be connected to one and other is also a bit tricky question. Should every model be able to connect to any model and so on? Remember that the people that often are going to use the DSL's and tools might not understand which connections that is legal and which are not. They should not even have to know because only the legal connections should be able to be drawn. How these "rules" are also mostly defined in the same kind of XML environment as the models are. Here it could be defined which kind of models that can be connected, how they are connected and how many connections there can be of a certain connection type.

2.1.3. Generating software

Transforming the models to text is the goal for Model-driven development. Say that you got a model for a Web service [WS02], this can be just a box containing the web methods and what attributes it has. Then think you got systems that use these Web services (boxes), each of these systems is again only defined as a box. Connections can be made between the Web services and the systems, defining that one of the systems owns a web service and another system uses it.

How to generate the code from this might be a bit hard to see just looking at the "boxes" and their connections. The transformation scripts are the solution to this. When we got well defined models (see 2.1.2 Defining models) we can easily use the models and transform them into code. The models are "picked up" by transformation scripts and here we can basically put the models into any context we want. The idea though is to figure out a way to transform the models in a general way, meaning that no matter how we many models and how they are connected together we will still get the code that we are suppose to get.

Transformation scripts can be written in different ways. Often it can be optimized just to write a transformation script for just the modeling you have done and not a general one. This surely depend on how the models are defined as well, the less accurate you definition of your models are the more difficult it is to write a general transformation script.

Transformation is not necessarily from model-to-text, there can also be transformations from one type of models into new types of models. The transformation scripts can in fact transform your models into basically anything you want to translate them into; the only limits are how you write your transformation scripts. I will not go into more detail about that here, but I will discuss it a bit more later on in this thesis when I go more into detail about how DSL's are used in Visual Studio 2005.

2.1.4. Automating

When developers create software they do much of the same things over and over again. Creating a Web service for instance, the developer needs to create many different files that are basically the same every time, effectively inventing the wheel over and over again. Model-driven development can solve this by letting one model be a whole Web service and letting the transformation script (model-to-text) create the files that are needed, reducing the development time greatly. When the model is there, the only thing that is needs to be added is the application specifics. Another great thing with Model-driven development is that it lets you create "business specific" or "company specific" specifications. Meaning that if the application you are creating needs to have a specific kind of security or any other specific details that can be created for you. Each new application or application part you are making will have these specifications. Letting you focus on the problem, and not all the small details that has probably been solved hundreds of times before, again greatly reducing the time developing your software.

When your want to make changes to your application; MDD has some great ways of solving just that. When you already have developed the application with use of models you can simply add new models and/or change how the models are transformed into code (transformation scripts), and other applications that uses the same models can be recompiled with the new models or transformation script changes. Since modification and maintenances of applications often is something that takes a lot of time for companies this can really boost the time spent on just reading and trying to understand old code, or trying to figure out why or where the documentation is wrong. MDD is almost documentation in itself, and accordingly much easier to read and understand.

2.2. Software Factories

"The industrialization of software development" [GSCK04] [SF05], software development is experiencing an increasing demand for software, and something have to be done to speed up the developing process. Either if it is new software or maintains of older software the developers is soon reaching their capacity, and either there must be educated many more software developers, which will increase the cost greatly, or there must be done something about the way we develop and maintains the software. Having many developers creating software is not just expensive, but it often reduces the reliability of the end result. It's better to have some few experienced developers than having many that might not see how the whole application is put together.

2.2.1. The Software Factory idea

Software Factory is an old concept that had the idea that software could be produced like most other things, with the help of factories. A car factor might have many similar, but different models in their production. The different models, however, uses many of the same parts as the rest of the models. The same thing can be applied to development of software. According to [ESF87] software can be developed and marketed like any other ordinary industrial product. We are seeing many different car producers which all have their “special” way of manufacturing their cars. These companies can in the world of software be translated into domains of software, or software that is similar in the way they are built (software families [PAR76]). Having these kinds of software families allows software developers to have a collection of “parts” that can be “picked” and used to develop software, just like a car manufactory picks the parts to make exactly the car model they want to build. Software factories can bee seen as a collection of domain specific tools, modeling tools, content, architectures, templates, project structures, component frameworks, patterns that helps the software developers to develop software in a faster and better way.

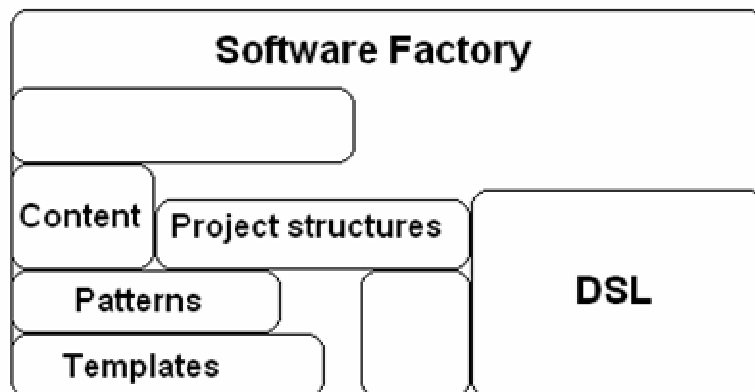


Figure 1: Software Factory, collection of tools, languages and items

The idea of Software factories is instead of building everything from scratch as we normally do today, as each thing we build is the first and only one of its kind. We recognize that we tend to build similar things. Many companies build games, and there should be possible to make some reusable parts for these games, tools that help to use them and guidelines that shows how to use the tools and the content. When a new developer comes he will instead of finding an empty project he will find everything he needs to get the application up and running. The nice thing with Software factories is that instead of heaving to invent everything from scratch; architecture, framework, discover through try and error what will scale and want will not, one can use the experience from developers that have done this in the past. Leaving the main focus on what’s unique about the particular instance of the application you are building.

2.2.2. What is a Software Factory?

Most developers build applications from scratch each time like it's the first application of its kind. Think if we put every thing that is the same with each application we make together and place it in a box. When making a new application we just open this "box" and we got every thing we need for making the application in the same way as the rest. If the firm you are programming for has some e.g. special security features then these security measures can be added in this "box". Effectively reducing the task to just focus on the problem that you where hired to solve.

Maintains of the software is also a great problem in software development today. It's often old and "forgotten" code that nobody really now how works that are used, and nobody dares to try to change. Often it results in a complete rewriting of the code witch is very time consuming, and yes, often exact same code is written twice. Software factories are trying to solve this in another way. If you found a hole in the security of your applications the ting you would normally have to do is go into each and every one of your applications and fix the problem. With the Software factories thinking you would only have to change the security hole one place. Either you can update the security you got in the "box" or you can rewrite it to meet future threads. Every application that is built on the basis of this "box" can then be changed accordingly.

A company has often very similar applications, and this is where the use of Software factories can be a great advantage. Software factories build on the idea of making similar but distinct software. As a company increases its software portfolio it also gets more and more similar applications. When knowing these similarities there can be made a "basic" application and then it's just for the developer to fill in what's different from the rest of the applications.

2.2.2.1. Software Factory schemas

A Software Factory Schema [GSCK04] [SFAP05] can be seen a document that is used to build and maintain a system. The Software Factory schema categorizes and summarizes artifacts such as XML documents, models, configuration files, scripts, source code, SQL files, test files and deployment scripts in a way that defines relations between them. All these relationships are defined in an orderly way such that it is easy to maintain consistency between the different items.

There are two common approaches when it comes to Software Factory Schemas; they are to use a grid or to use a graph.

When using a **Grid approach,** you define a grid which has rows that define abstraction level and columns that define concerns. Each cell then becomes a viewpoint from which we can build some aspect of the software.

	Business	Information	Application	Technology
Conceptual	▪Use Cases And Scenarios ▪Business Goals And Objectives	▪Business Entities And Relationships	▪Business Processes ▪Service Factoring	▪Service Distribution ▪Quality Of Service Strategy
Logical	▪Workflow Models ▪Role Definitions	▪Message Schemas And Document Specifications	▪Service Interactions ▪Service Definitions ▪Object Models	▪Logical Server Types ▪Service Mappings
Physical	▪Process Specification	▪Database Schemas ▪Data Access Strategy	▪Detailed Design ▪Technology Dependent Design	▪Physical Servers ▪Software Installed ▪Network Layout

Figure 2: Software Factory Schema, Grid approach

One cell might define some abstract level of some date while another cell might define the concrete viewpoint of the data. Ones the whole grid is defined it can be populated with the views that include the development artifacts for the software that is being constructed.

A grid can be used to build more than just the specified application or software that is was originally designed for. The grid defines the resources that is required to build the software, but before the application or software specific views are added there can be created any application within the "software family" that is described by the grid; the views which can consist of DSLs, patterns, frameworks, tools, etc are the things that make up the specific software.

Artifacts that are described by the viewpoints are often models, like DSL models, but it does not have to be models and frequently they can be any other source artifacts which can be based on a formal language like scripts, WSDL files, SQL files or any other programming language source files. The viewpoints define not only the languages that are used, but also the requirements on the views. These requirements can be expressed by the use of patterns or by the use of constraints. Two viewpoints can e.g. use the same DSL, and the requirements can define how they should be used in the two different views.

A **Graph approach** for representing a Software Factory Schema is a better way to go then with the grid approach. When using a grid approach we are basically simplifying the reality which in most cases isn't the way we want to go. Making adjustments based on

the tool's limitation when trying to "port" the reality into software often results in something that is only similar to the reality, which can be good in many cases, but the goal is of course to create something that is as close to the reality as possible. A directed graph that uses nodes as viewpoints and uses edges as computable relationships between the viewpoints can better illustrate the reality than a grid.

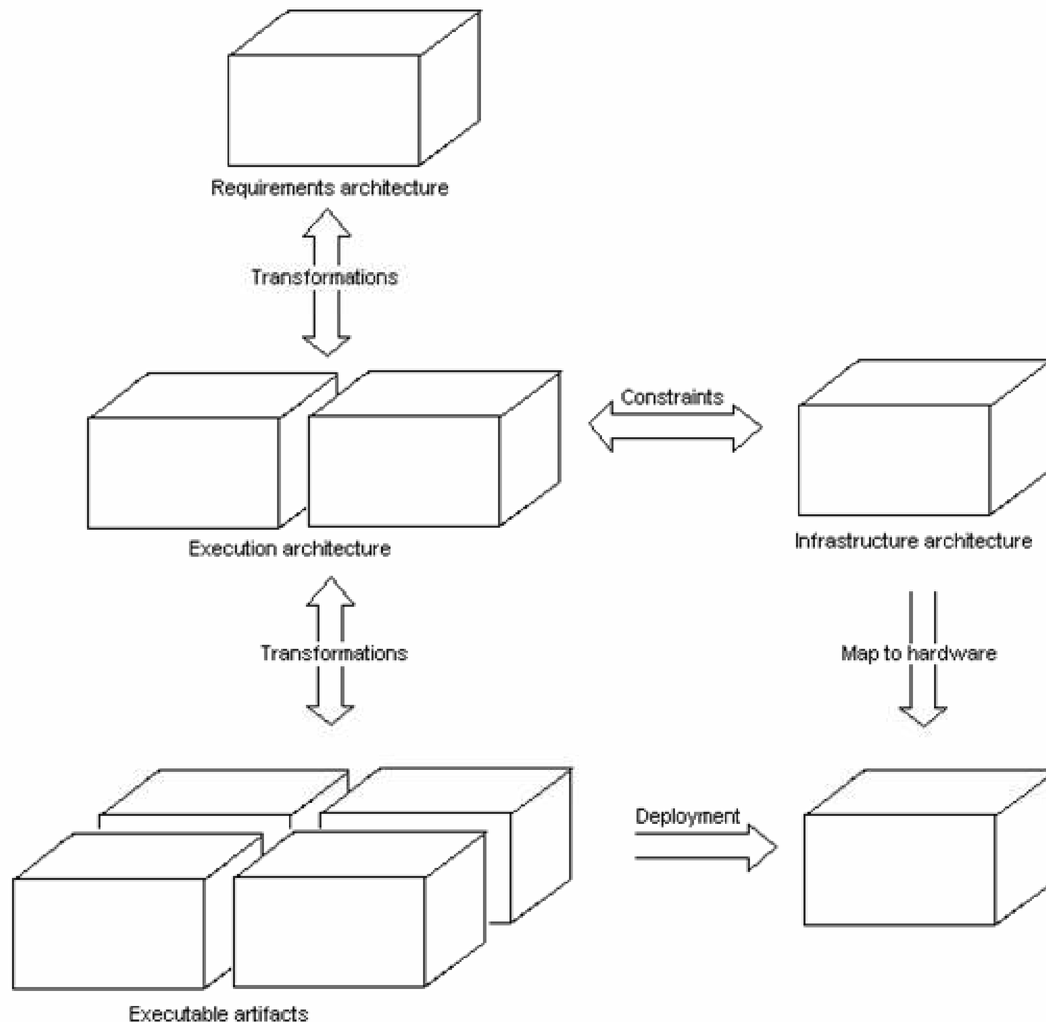


Figure 3: Software Factory Schema, Graph approach

When using a grid approach the viewpoints must fit into a classification scheme with rows and columns. The way that a graph approach can reflect the application or software architecture really improves the understanding of the schema.

Customization of Software Factory Schemas is an important part when building applications with Software Factories. An unprocessed Software Factory schema can be seen as a formula or recipe for building a family of applications. Building one application

however requires customization, depending on what special requirements the application needs it needs to be modified accordingly. Since the application or software families are built up in the same way it means that only the things that make the application unique needs to be changed or modified.

A Software Factory Schema normally contains both fixed and variable parts. The fixed parts are the parts that are the same for all members of the “software family”, while the variable parts can be changed to house the unique requirements that an application may have. Different parts of the customization can also be preformed at different times, meaning that it is possible to build the application even if some of the parts have not been modified.

2.2.2.2. Product development

Using Software Factory Schemas and thus creating application families has the goal of speeding up the development of applications within this family. When developing software with Software Factories it requires us to go through some steps to achieve a good and usable Software Factory.

- First thing to do is to determent how we can solve our problem. We have to analyze the problem and determine whether it is in the Software Factory domain. Is it a very unique application or could it be a part of an application family?

- Specifying the requirements that are needed in the family, which parts that are the same and which parts are variable.

- Designing the architecture.

- Customize to create an application that has unique features

- Deploying the application

- Testing, produce testes that ensures that the application doesn’t have logical or any other errors.

2.2.2.3. Mechanisms

A range of mechanisms can be used when specifying and implementing the application. The variation that is required limits or extends the mechanisms that can be used. This will let us use natural methods, at least if we see it in the old fashion way with hand-coding every little bit of software that is produced, when developing software. [CHE04] has described a range of variability mechanisms based on feature models [FMW01].

2.2.2.4. Mass customization

The idea of mass customization is that it should be easy to define an application and that it is made for you. Just like when you are buying a PC on the web, you can customize it easily on the web, deciding which parts you want, and then somebody builds it for you and sends it to you. In the same way it will be possible in the future to customize your own software. It is probably not going to happen very soon, but Software Factories will help make this possible. Some companies have specialized in selling cheap web page designs that can be easily be customized to get an individual look. This isn't the same, but it is a step in the right direction.

2.2.3. Raising the level of abstraction

Abstraction is different ways to look at a system, application or anything that has more or less details. Raising the abstraction level can be seen as just the same as reducing the complexity. When looking on e.g. a cell phone you can say that it just an item, but if you look closer or open it, reducing the abstraction level, you will see that it's built by many small pieces which again are built by even smaller pieces. To use the cell phone though you do not have to know how all the different parts are put together or how they even work separately. To use it you only need to know how the whole thing works. This is also one of the ideas with Software Factories and DSL tools.

2.2.3.1. Abstraction

UML [UML05] [UML04] has for a long time been used to draw systems or applications, but without a specific UML profile it is not really more than a sketching or drawing tool. UML gives us good diagrams illustrating how a system works, but it does not interact with the programming of the system itself. It therefore often is hidden away and never seen again after the software is developed and deployed. It's also often that we midways in the programming figure that there are some major logic errors in how the modeling is put together and at best the UML diagrams are redrawn to solve the problems. In most cases it's just forgotten and not used. The big problem is interactivity between the models and the code, and that's what Software factories try to handle.

We also got multiple platforms that we want our applications to run on. Platforms like .Net [.Net05], J2EE [J2EE05] and CORBA [CORBA05] are widely used, and larger applications often run on more than one of these. To do this we then need to program the exact same thing for each of the platforms we want to run our application on. It's time consuming and boring work to do the same thing many times, Software factories try to map the original application to the platforms desired; saving time and money.

2.2.4. Reducing complexity

When creating software on model level we immediately raise ourselves from the detailed code environment and lose sight of details like integers and strings. Boxes can define a class or even a whole system and we can draw connections between the different systems, defining how they interact with each other. Basically the Software factories

upgrades (or reduces, based on the eyes that see) the programmer to an architect rather than a programmer. He or she will still be a software engineer but on a whole new level. Bosses can now draw the whole system them self without having to know anything about integers, doubles or strings of any kind. The customer will also understand much more of how the application works by seeing the model instead of just seeing the code, and since the connection between model and code is real it reduces the problem that often happens between customer and developer, that they think they understand what the other part wants or needs, witch in many cases is not true.

2.2.4.1. Problem of complexity

Complexity is not easy to define precisely, but as we all know the more complex a task is the more difficult it is to solve it. In other words we can therefore say that reducing the complexity reduces the difficulty of solving the task.
Reducing the complexity is not just straight forward though. When reducing complexity we often also reduce our choices when developing software and that is not what we want. How can we reduce the complexity without reducing the performance of our applications? There is not one right answer for this, but creating better languages and better developing tools has been a way to do this. Another way is to see from another angle. You know that you are programming in one kind of domain and most likely you are not the first one that has done this. The idea of Software factories is to try to use the knowledge of previous development in this type of domain to develop perfectly adapted tool lists of the most used tools (By tools it's meant anything from classes to menus and buttons), and remove, hide or at least put the rest in the background.

2.2.5. How to model software

How can you model software? UML has been a way to describe how an application really works or how its different parts interact with each other. We can say that UML has been the closest thing to modeling software before the introduction of Model-driven development. UML is actually used as the modeling language for Model driven architecture (MDA) [MDA03]. The thing that separates UML from Model-driven development is the use of model-to-model and model-to-text transformation scripts. These scripts, you can say, "translates" or transform the models into new models or text. The nice thing with this translation or transformation is that we can decide how we want this transformation to be. If we want to translate a model into a plain class we can do that. You probably think this sounds a bit weird and it's not that way it's intended to be used either. To model software you need to have a good definition of your models, e.g. if you are modeling a connection between a Web service and an application you need to define this connection in a good way so that it will be easy to use when you are writing a translation script.

2.3. Domain specific languages and tools

Domain specific languages and tools is a way of reducing the complexity in software developing. You can think of it in the way that it can be created a language and tool that is perfectly adapted to the specific domain you want to develop your application. Reducing the chaos of numerous tools and possibilities to just the ones you want and can use.

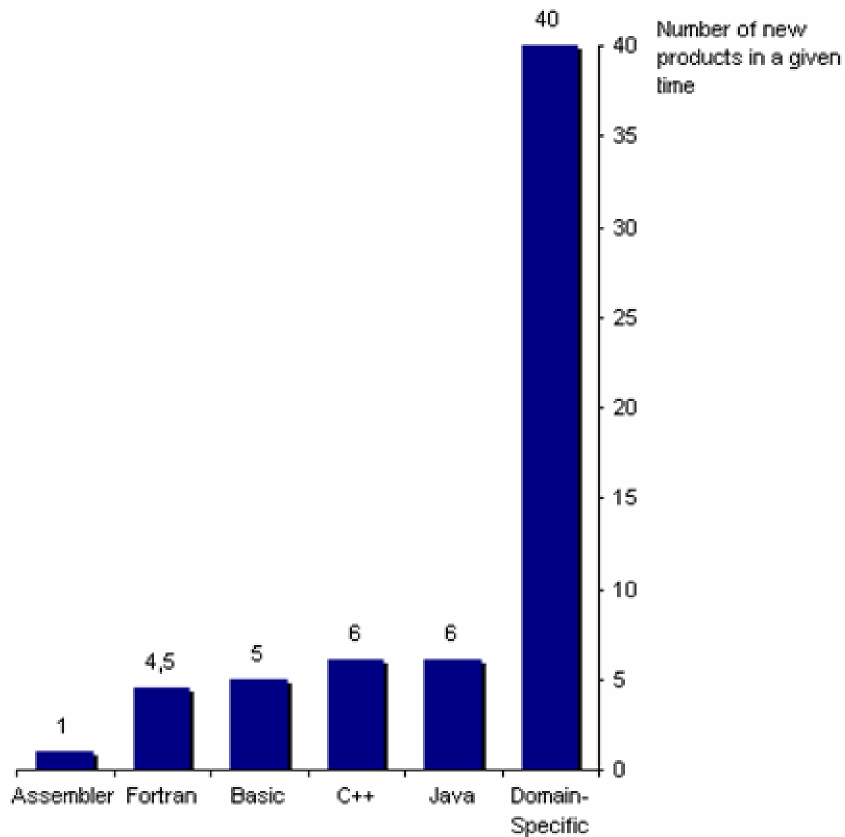


Figure 4: DSL, Increased productivity graph

2.3.1. Domain

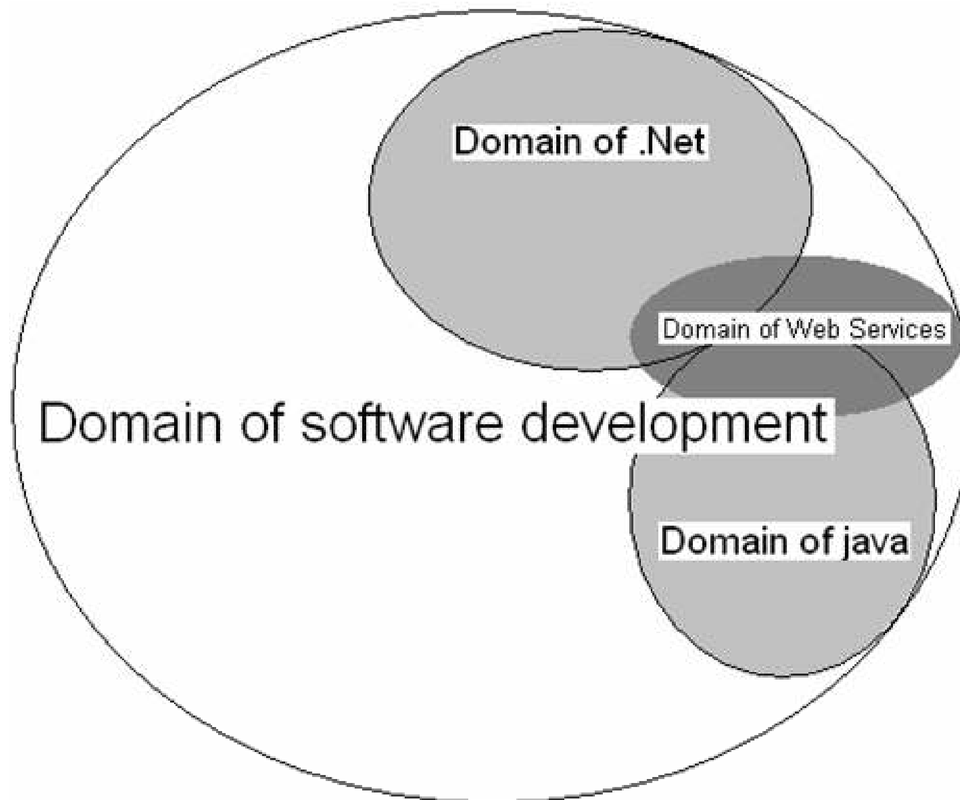


Figure 5: Domains. Domains can overlap domains

The word domain is often used, but it can be confusing what it really means. Sometimes a domain is referred to as web services, domain of web services. In the same context we also find the domain of all .Net applications. Meaning there are 2 domains covering the same web service. This might seem a bit confusing but it's really just the level of abstraction you use, or how far you go in detail when looking on the web services. A web service will also be in the domain of software engineering, meaning we can use a high level of abstraction when talking about some web services or we can have a domain that only covers one web service. We can therefore say that domains are recursive.

2.3.2. Design

When designing a domain specific language it's needed as much information on the domain that the DSL is created for. When designing DSL's for Web services its needed different expertise then designing a DSL for business applications. This is even one of the idea's that make DSL's so attractive. Developers who has deep knowledge about Web services are the ones that make the DSL's and since the experts knows about what's a good Web service design and what's not they will be able to make DSL's of Web services that is perfectly fitted to use on the domain that is wanted.

2.3.3. Creating domain specific languages

When creating a domain specific language we need to know how the system should be build, what kind of pieces (models) that is needed and how they should connect to each other and so on.

Now we know which parts we need when we are using the DSL, then we need to define how the different parts (models) are build and which ones that can connect to each other and so on. Designing each model basically, defining which attributes that it can have. If we are designing a Web Service we would here define that it can have web methods [WS02]. How the different models should look like and what type of connections that can be used would also be defined here.

Transformation of the models is written after they are defined. How they are translated is up to the designer. If we got our Web service we know that we need multiple files to be able to create a working Web service. The first things that are needed to do are to figure out what kind of files that are needed. Here the domain specific part comes into the picture. What kind of transformation that is needed depends on the kind of domain we want for our Web services. If most of them need a special authorization this can be put into the transformation script so that for every model that is transformed into a Web service will have this authorization code.

2.3.4. Automation

Automation and generation have become often used words in software development today. Along with ever increasingly better developing tools there has also been added wizards that allows you to make some choices and based on those choices there are generated code for you. Automation can mean many things in software development; one feature that has been added to many of the new developing tools that are made today lets you choose to e.g. change the name of a variable; the tool can then automatically change all instances of that variable to the new name.

When it comes to DSL's and automation we can see this in the same way as with a variable, if we change something with the model, that change can be directly made in the code as well. Automation also works in the way that when using the DSL and adding new models and connecting them can automatically update the code to fit the modeling. So that the application is basically generates the code as we are "drawing" the application. This will ensure that if there are any errors on the way that the application is "drawn" then we would get an immediate error upon trying to building it.

Automating with DSL's is however mostly seen as the ability to not reinvent the wheel all the time. Making a new application means, for most developers, starting with a completely blank file and typically start writing *static void main(..).* The whole idea with DSL's is to automate this, so that creating a new file we would start focusing on the problem we are solving and not how to just get the program to run as we normally would do. DSL's also lets us decide how we want to build our application. If you have used template files or pre-made solutions you know that these are static, meaning you can

either choose one template or another one but you cannot mix them together. Wizards can to some extend be used for that, but you normally haven't got that many choices. DSL's however lets you create any application you want if you have designed your DSL's right. You can basically pick any parts that are made and put them together. Again, if they are designed in the right way, only allowed combinations should be possible to make. Nothing is more annoying than having to focus on every little detail of the application. Letting the development software do all the small bits and pieces that we have done a 100 times before will help the developing time greatly reduced.

2.3.5. How does Domain Specific Models differ from plain UML models?

Unified Modeling Language tries to offer "one size fits all" concepts and generators, which as we all know cannot be easily done. Even if UML could create something that is that universal it would still be a problem which is not so easily solved with a universal language. When you try to cover every area with the same kind of models you lose the ability to see the same kind of pieces since, maybe not within the same company, but in a general way. Companies will make their own models and code, which probably will be different for each company, so we're back on creating the wheel again. The domain specific models try to see it from a different way, by focusing on a smaller domain and optimizing for that domain.

Domain specific models are usually created by experts or people that know very much about the domain that the domain specific models are going to be created for. Experts can take advantage of their knowledge about the requirements that are needed in the domain, and thus make better models, or model definitions. Higher level of abstraction is easier to be made, because of the ability to easier express the terms of domain specific models in a conceptual way than it is with UML models.

2.4. Model to text transformation

When you are using models you need a way to transform these models into some usable text or code. Transformation scripts are used to exactly this. Transformation script are not some pre-designed scripts that follows with the development software you are using, but a script you must write yourself. This gives us a very flexible way of programming with models. There can even be multiple scripts that separately give the same model different meanings.

2.4.1. Custom Tool

When transforming models into code or text we need something that can "translate" these models. For each file that is in a project we can add a custom tool, and based on that tool the models will be translated into text.

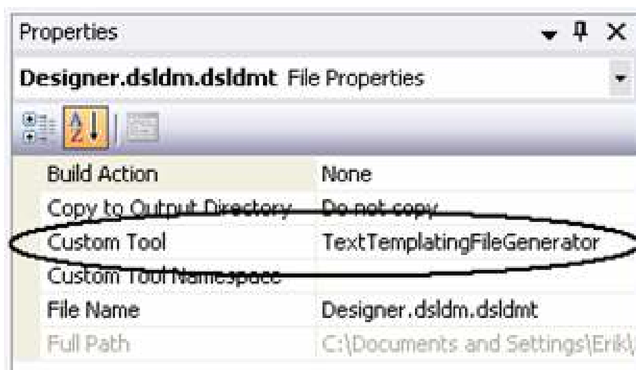


Figure 6: Custom Tool

2.4.2. Custom code

Custom code is the code that actually transform or translates the models into code. The definition of the models now shows the importance of being good so that the models are easy to use in the code. Determent on what the output file format is, the transformation of the custom code generates a file which can be opened in Visual Studio. Normally one template produces one file although there are methods for generating multiple files from one template. There can however be multiple templates attached to the modeling, which can produce different files by translating the models in different ways. One would maybe like to make a Web Service, and a Web Service needs multiple files to work properly, thus there need to be generated more than just one file.

2.4.3. Custom code syntax

When making a transformation script there are some specifications that defines how the script should be translated [T4M05]. First we need to define which type of file format we want on the file that is generated.

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTran
sformation"#>
<#@ output extension=".htm" #>
<#@ examplemodel processor="ReplenishingServiceDirectiveProcessor"
requires="fileName='Sample.ReS'" provides="ExampleModel=ExampleModel"
#>
```

Figure 7: Custom code, file extension

The output extension can basically be set to anything we want, but the extension, or file type will be used as part of the project or application so for it to work as a part of this it needs to be something that can be “read” properly by Visual Studio, thus needs to have the right file extension.

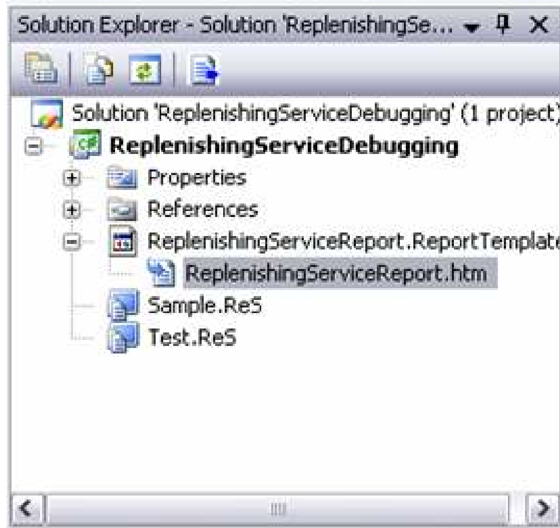


Figure 8: Custom code, generated file

As you can see from this figure the file is created with the same file extension as described in the custom code.

The custom code language [T4M05] is built up by tags similar to XML or html; the tags are static though and consist of these blocks:

2.4.3.1. Control blocks

- <# #> - Regular control block. These blocks control the flow that runs between them. Meaning that if you got the start of a loop in one control block you need to end it in another (or the same) block. Everything that comes in-between these two blocks will be done as many times as the loops runs for.

- <#= #> - Expression control block. This block will take some value or variable from the block (code) and be replaced with that value at generation time. The expression will automatically have a .ToString() appended to it.

- <#+ #> - Class features control block. This block allows you to add new methods, fields, properties, embedded classes etc. to the class derived from TextTransformation. You can then use these from regular and expression control blocks, making it easier to structure the code.

2.4.3.2. Standard directives

<#@ - is used for standard directives like import, defining custom code language, defining file extension and so on. If we e.g. import a class it can then be used inside the control blocks.

```
<#@ template inherits="MyNamespace.MyBaseClass" language="C#"
culture="en-US" debug="false" hostspecific="false" #>
<#@ output extension=".cs" #>
<#@ assembly name="System.Drawing.dll" #>
<#@ import namespace="System.Collections" #>
```

Figure 9: Custom code, Standard directives

2.4.4. Transformation

The transformation runs through the custom code from top to bottom and spits out text. Text that is outside blocks is just transferred directly to the generated file without any changes, except if it is between the start of a control block like a loop or an if-statement and the end of the statement (another block). Depending on the outcome of the statement the text in-between will transferred to the generated file accordingly.

The control blocks runs through the models that are created and can in this way get all information that is stored in the models, and if needed transferred into the generated files by using the expression control blocks to get variables, or other information that is stored in the models.

2.5. Visual Studio 2005

Microsoft Visual Studio 2005 [MVS05] has just been released. Visual Studio is a collection of tools that offers many benefits for developers. Tools that VS 2005 contains are among others Visual C#, Visual Basic, Visual J#. C# has been the new and rising programming language on the .Net platform which is the platform that Visual Studio uses. There are quite some new features in the new Visual Studio 2005, but I will only go into detail on the Domain model designer, and other new details that are relevant to this thesis and leave the rest of the details for you to discover.

2.5.1. DSL designer

The Domain model designer is a completely new feature in Visual Studio and has not been on any of the previous versions. The domain model designer lets you design your own domain specific language which you later on can use to develop applications.

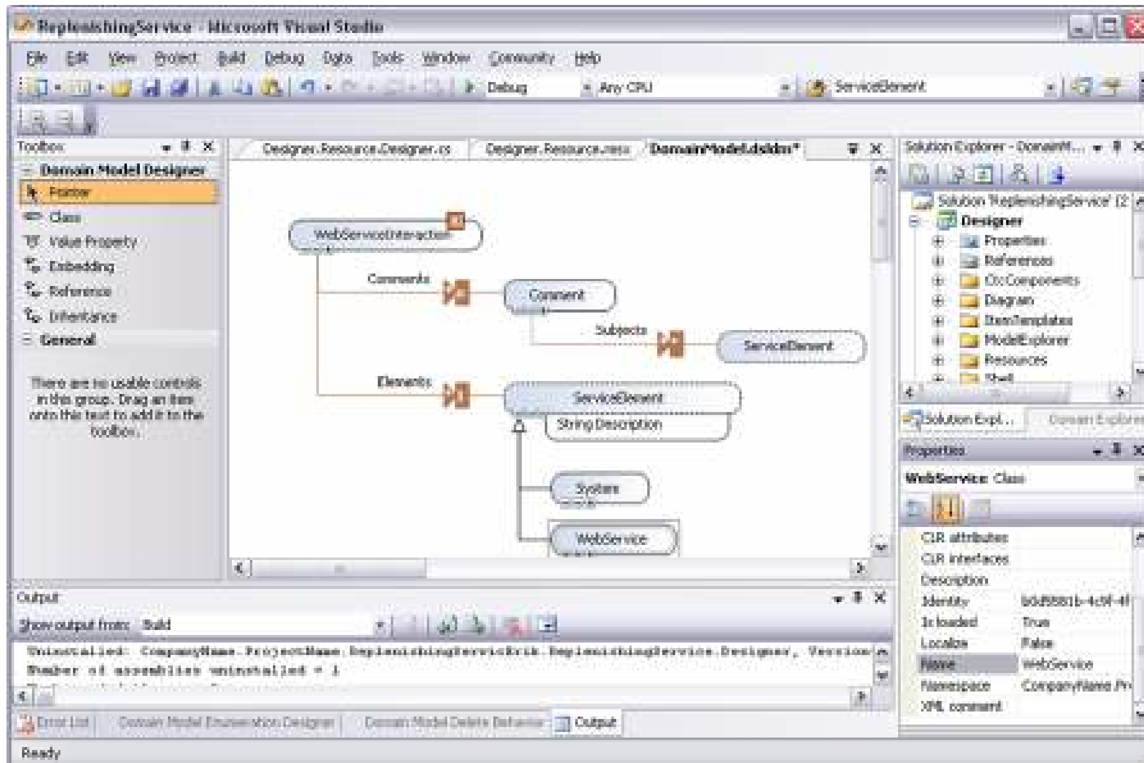


Figure 10: Visual Studio 2005 Domain model designer

The domain model designer has a toolbox on the left-hand where all the visual parts for developing a domain specific language are located. These can again be dragged onto the "designer file" which you can see in the center of the figure. On the right-hand you can see the solution explorer which shows all the projects and files that are used in a solution. Just below the solution explorer lays the property editor where the selected item (on the designer file) can be edited.

Once the Domain specific language is built we can use it by building and running the solution. A new Visual Studio will then be opened with a new solution. This new solution is based on the DSL that was defined, and lets you use this DSL to create applications. More on how this can be done, and what's needed to be able to transform the models into usable code will be explained later on in this thesis.

2.6. Web Services

Web Services was developed to create a good and reliable way of transferring date through the web. Web Service can be defined as programmatic interfaces that use XML as the base technology. The main idea behind Web services is that it is easy to read both for humans and computers, and especially that it can be made with one programming language and read by basically any other languages. [WSE02]

2.6.1. Basic concepts on web services and service oriented architecture (SOA)

Service oriented architecture is a concept where the use of services to meet requirements of applications or systems. These *services* are described in such a way that they are accessible in standard ways, meaning there are standard (basic) types that are defined that can easily be adapted or translated to other language specific types. They almost work like a normal function or method in programming languages. You send a request with some input data and then the service processes the data, before it gives a response. The input and respond data has to be well defined for the service can be used by platform and programming language independent calls. [SOA03]

2.6.1.1. Service

The word service is often used in software development, but it might be a bit confusing knowing what it really means. *Service* is a word that it often just outside software development as well and it might have some different meanings, but when used in context of software development we normally look at a service to be something that normally is platform independent, self contained and stateless. A Web Service would have an interface (WSDL) defining the types that should use, and what type that is returned. The Web Service would normally have a port that lets you connect to the service, and then it would have one or more access points, where the actual data would be sent back and forth. These access points would look like normal methods with specific arguments and return values.

2.6.2. Usability and possibilities

Service oriented architecture and especially Web Services has got more and more attention by software developers. The way that services can be used by different programming languages and tools at the same time has really boosted the use of these services. Imagine that you got a large system that is based on transmitting data across the net. The system might have numerous different connections for all sorts of management on the system. Let’s say that you had a part of the system that creates, modifies or deletes users that can log on to the system. The system is used by many different companies and some of these companies want to create their own type of “user management” which they might think is less complex than the methods that already exist.

If we build a service, and let’s say a Web Service in this case, we can basically create users in the system using any kind of programming language or tool that supports Web Services. This lets the user of the system have a more diverse way of building and using a “user management” on that system. Any tool or language can that supports Web Services can basically be used. Security measures can also be added like authorization, this can be normal “login” with user name and password.

2.6.3. SOAP

Web Services are described using Simple Object Access Protocol (SOAP) [SOAP03] [SOAP05]. SOAP is a text based description format and messaging protocol. Since it is text based it is much easier to get through heavy secured networks that uses firewalls and other security measurements. A SOAP message is also defined by XML which is supported by most modern programming languages.

SOAP is basically built up by a header and a body; where as the content of the message is normally placed in the body. The header is often not used, but this is the part that is parsed or "opened" first and thus it is a great place to put any authentication or authorization code.

2.6.4. WSDL

Web Service Definition Language (WSDL) [WSDL05] [WSDL01] is a XML based standard for describing a Web Service. WSDL can offer what kind of parameters that it has, what the name of the parameter is and what type it is. Types are normally just basic types like integer, double, string, boolean, date and time. The return type, if any, is also described in the WSDL.

3. DSL Transformation requirements

This section will reflect on the requirements both for the Domain specific languages and the Domain specific language transformation.

3.1. Model to model requirements

Defining the DSL require a transformation from one set of model (the domain model design) to another set of models (the domain specific language models).

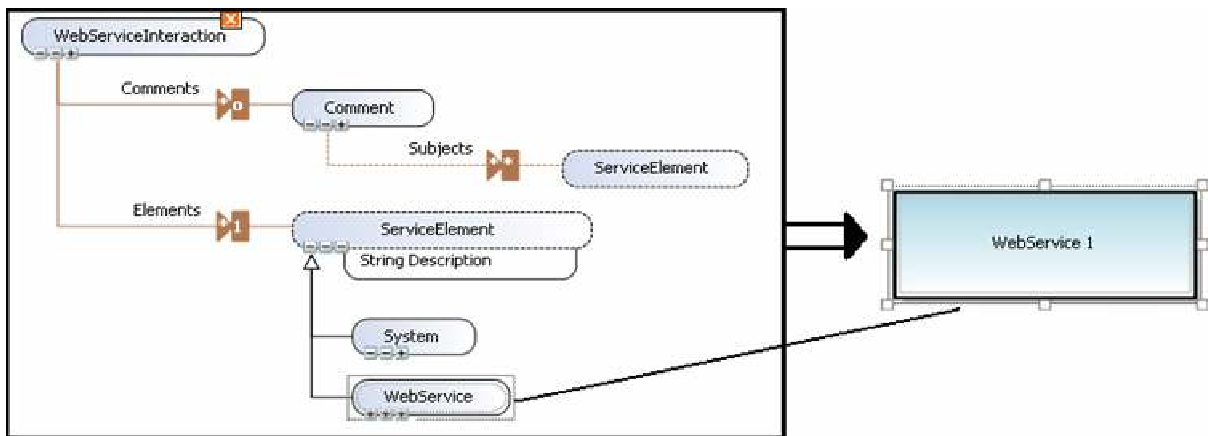


Figure 11: Domain model designer to a DSL model

In this figure we see an example on how a model-type can be generated from the domain model designer. Behind this generation there are a great deal of definitions and requirements and I will try to go more into detail about how this is done in the following sections.

3.1.1. Design

The domain model design is a lot more than just the models and arrows that can be put together in different combinations. The DSL not just has to be designed in the right way by the use of classes (see 5.2.1 Class) and connectors (see 5.2.2 Relationships); it also has to be combined with text that says how the model can react and how it can be connected to other models. This text file is a XML file that is directly connected with the modeling, here we can describe whether a "class" be translated into a model or not, how the model shall appear (square, circle, picture etc.), if it has classes "within" and so on.

Designing relationships is also required. There are 3 relationships that can be made (see 5.2.2 Relationships), where only the reference relationship can be transformed into connectors between models in the DSL. These connections need to be specified in the XML file. Here it's possible to define what kind of source and destination connections that can be made, which way that the connections can be set (from one model to another only, or both ways) and how many connections that are allowed from or to each model.

3.1.2. Specifications

Specifications on how a model should behave are specified in both the visual design and in the XML file. Getting a model that cannot be interpreted in one way is essential. Getting an ambiguous model can lead to misread or misuse of models upon using the models for developing applications. If the models aren't specified good enough both people that use them and when translating the models the machine can use them in the wrong way. A model that is of the type "box" is not easy to understand, and could thus be used in many different ways if the specifications isn't good enough. Specifying which models that can connect to which models, are therefore very important if there is someone that is unfamiliar to how the different parts can fit together. DSLs are often created so that software architects which might not know so much about how code is developed can use the DSL to develop software.

3.1.3. Synchronization

Synchronization between the visual design level and the XML file is also an important requirement. If these get unsynchronized and it's not detected we can easily discover strange build errors when trying to build the DSL. When creating a class that we want to show as a model we need to describe how this is done in the XML file, if the name of the class is misspelled in either the visual design level or in the XML file we will probably not even be able to transform the files and certainly not build the DSL.

3.1.4. Transformation

The transformation is done in two steps, first we need to transform the files (visual design, XML files and other files that are included). These files are transformed into code files that can be compiled, and thus makes them readable for the machine. If there are some illegal combinations there will be error messages that tell you that there are some changes needed to be made. When the transformation is successful you need to build the solution to get the completed Domain specific language, but also when trying to build the solution there might be some error that was not detected on the transformation level. Changes need to be made again to get the completed DSL.

3.2. Model to text requirements

By now we should have a working DSL and we can start focusing on the development of application or software the DSL was designed for. There are still quite a few challenges that remain to get a working application. Designing the application with the use of the DSL is one of the first things that can be done, but we can also start with creating the custom code (see 2.4.2 Custom code) that translates the models into code that can be compiled. If it isn't the first time that the DSL is used the custom code is probably already written, but still we might want to have a unique set of transformation and we might want to rewrite or add features to the custom code.

3.2.1. DSL tools text templating

DSL tools text templating or custom code is required to translate the DSL into something that can be read by the machine. Without the "transformation script" the DSL is really not more than a normal modeling language like UML although it is optimized for a certain domain.

Writing the custom code can be a bit difficult because it is required that the code works just as well for any application produced by DSL. When writing this custom code we need to make sure we know all the combinations of connections between the models. We must also be sure that we have added all the models and the connections to the custom code.

3.2.2. Reference to models

References to the models and making sure they are all covered in one way or another so that there can't be produced software that is not working as it was intended. When using multiple layers in the models, compartment models (see 5.2.3.2 Compartment models); we need to make sure that we "remember" to include these under-laying parts that are easy to miss.

3.3. Other possibilities

There are many other possibilities when transforming models into either new models or text. Since you can write the transformation scripts yourself, you can basically decide how you want a transformation to go.

3.3.1. Shape inside shape

One interesting way of programming with models is creating models inside models. This leads to an interesting way of programming with models. Recursive programming is an often used method when programming with code, it's also a useful way of reducing number of lines that needs to be written.

Defining shapes that are inside another shape can be seen as using different abstract levels. The outer shape would be a collection or the "result" of the inner shapes this way of programming with models can really improve the documentation of the applications that are made.

There are no directly supports for creating shapes that are inside shapes, but there are no limitations either that makes it impossible. There are added some "compartment shapes" in Visual Studio 2005 Domain model Designer, which can have some inner relations, but it's not directly shapes inside shapes. Defining your own shapes is possible, although it might be quite a lot work. There is expected to be available many more shapes as people start using the tool.

4. Software Factories and DSL tools walkthrough

Microsoft has released a new tool to their Visual Studio 2005 which builds on the idea of Software factories. This tool is called a domain model designer; here you can design your own Domain specific language and tools.

4.1. Raising the level of abstraction

When programming with models you need to figure out how to raise the level of abstraction in a way that gives the user a good experience and understanding in how the models can represent code or code parts.

4.1.1. Models

Models are defined in the way that they represent code-bits that can be defined in multiple ways or put together in different ways to create the application we want. The models need to be defined in such a way that they are easy to use and understand what they stand for.

4.1.2. Views

Programming with models is not like normal programming at all, there can be multiple different views. First you need to define how your models look like, how they work and how they are translated. Then these models can be translated into new models that again have to be translated.

4.2. Reusability

Reusability is a very important factor for using DSL and tools. Companies often have unique structure on their application that is the same on every application that is created. This can either be a security matter or some other special way that their applications are made. Normally programmers have to first program these things every time they make a new application, which is both boring, and often very time consuming.

Templates are “pre-coded” code that can be used to solve these problems. When using a template the basic code that you have typed in beforehand is the start of your new application. The “problem” with these templates is that they are static in the way that they are the same every time they are used. This is perfectly ok if every application is built this way, but if they are almost the same and they have some varieties this is not the ideal solution.

When using DSL and tools you can create small parts that are often used. This can be parts like a database connection, a special security code or other small parts that often are used. These parts can then be created as models, and when making a new application you can decide which parts you want to add. Almost just like when making dinner, you can

have meat, crossed tomatoes and a third ingredient, depending on what that third ingredient is you got many different possibilities for a dinner. The same thing can be used with the models; depending on the combination of the models you can create a special application. Templates cannot be used in this way and therefore DSL and tools gives us a much more dynamic way of building our applications.

4.3. Dealing with change

Plain UML has often been used as the modeling language for software (and of curse in many other areas), but plain UML has one major set back, and that is that it do not directly interact with the code. UML has therefore been used mostly as a drawing or sketching tool. The models that are made for an application has often been "wrong" in the way that they had to bee changed after developers have started programming on the application. Very seldom a UML diagram has survived from start of programming to the end results; there is almost always something that needs to be changed.
When programming with models like DSL and tools you are working directly with the code and every change made on the models means change on the code. Working directly on the code like this every little changes that are made can be directly transformed to code. Think that you got multiple applications made like this and you got a security feature on these applications, and the security feature is a model. Then you discover that there are some serious holes in this security. Normally this would result in a serious rewriting of every application by itself. Which is very time consuming and you are basically doing the same thing over and over. When programming with models you can change the definitions of the model and how it is translated into code. This means that when security holes are discovered you need only to change the model itself and each application will then again get these changes since the models are directly connected with the code.

4.4. Error detection on design level

The direct translation from model to code gives a unique way of detecting "errors" on design level. Normally when designing a UML diagram for an application it is change not one but several times because there are some unforeseen problems that reveal themselves when the code is actually written. Programming with models as prevent these unforeseen problems, because the code is generated at modeling time. So instead of doing quite a lot of modeling, then programming until there are some problems, then redrawing the modeling over and over again we can do this only one time.

4.4.1. System logics

A big problem when drawing models of a system is how it can be converted into actual code. There are often logical errors here. The tings that looks easy to program often turns out to be very difficult or even impossible. Programming with models reduces the chance of doing this greatly. The code is generated at modeling time and will give errors, if designed well, on mismatching modeling.

4.5. Reducing productivity time

One of the greatest advantages for Software Factories and DSL tools is that the production time is greatly reduced. As the applications we build are getting larger and larger and the complexity is increasing and the time to build them are as follows also increasing. To be able to split up this complexity into smaller parts reduces the complexity and it is much easier to see how the application is put together. Modeling gives us a higher abstract level of view and in this way can reduce some of the complexity.

When making a new application developers write the whole application from scratch and this takes a lot of time, effectively inventing the wheel multiple times. Let's say that we are building a car, and if we say that the wheel is built up by multiple parts. We can either buy each piece, try to figure out how to fit the pieces together each time we need a new wheel or we can just buy a whole new wheel. It's obvious what would be the easiest thing to do. The same thing can be applied to software development.

4.5.1. Application specific

After building many application we start to figure out that there is some similarities among the applications. These similarities can be "extracted" from these applications and reused when creating new applications that is in the same kind of domain. From these similarities we can build Domain specific languages and tools that are perfectly adapted for creating applications for that domain. When creating new applications using this DSL's and tools the only things that need to be done with the application is to add what is application specific. The idea is that instead of having to code the same things over and over, the only things needed to do are coding the solution for the application, and not the things that is written over and over again. Applications are getting more and more advanced and we are packing them with increasingly more "extra" features like increased security, automatic updating, special features like connections to the web etc.

4.6. Increasing reliability

Reliability and stability are keywords in software development today. Nobody wants to buy or use software that crashes all the time, and therefore software manufactures need to be able to create reliable and stable software. Manufactures use more and more money on testing their applications.

Software Factories and DSL tools has the idea that there is made some Domain specific languages and tools that are used by many developers and companies. By doing this developers might easier find errors, security holes and so on in these, and will therefore create more reliable and stable software.

4.7. Increasing security

Security is also a very important topic in software development. The need to protect sensitive data has only been increasing while the methods for getting to the sensitive data has only become better and better.

In the same way as when increasing reliability, security is also enhanced when more and more developers uses the same parts when developing software. When more people work with something the more easily e.g. security holes are detect.

Adding security features into models so that the developers do not have to think about security when they are developing the software can really boost the development rate a great deal. When the models are transformed into code, the built in security features in the models will also be added without the developer having to think about it at all.

5. Defining Domain specific languages and tools

Defining Domain specific languages and tools can be seen as trying to define how we in the best way can develop software within a specific domain.

5.1. DSL in short

DSL's and tools are used to help us when developing software within a specific domain. Normally when using languages and tools they are developed to be used to create basically any application. There are of course some programming languages that are based on mathematically solutions, small programs (scripts) and so on. The unique thing about DSL's and tools is that they can be created fairly easy and you do not have to be a compiler designer or anything like that to use it either and they focus on only one domain which makes them usable almost anywhere. UML has multiple domains that it cannot address because of the way that it tries to be usable in a general way. DSL and tools should however be created by experts on the domain that the DSL's and tools are made for. The more that is known about a domain the easier it is to build a good DSL that can be almost perfectly described accordingly to the domain description.

5.2. VS 2005 DSL Designer

With Visual Studio 2005 there has been released a new tool called a domain model designer. This domain model designer lets you create your own DSL in VS with the help of different objects that can describe the DSL. After the DSL is created you can start a new instance of VS 2005 where you can use the DSL that you just described to develop applications.

I will try to explain how this is done by going through the tool and the objects that can be used in defining the DSL.

5.2.1. Class

A class is an object that isn't really a class as we think of a class in normal programming languages, although, it can be as well, but I will go more into detail about that later on and it will become clearer when you see the whole tool in one picture.

A class does really not have to be anything when we are using the DSL it might not even be possible to find when using the DSL. I'll try to explain how classes are put together by starting at the top.

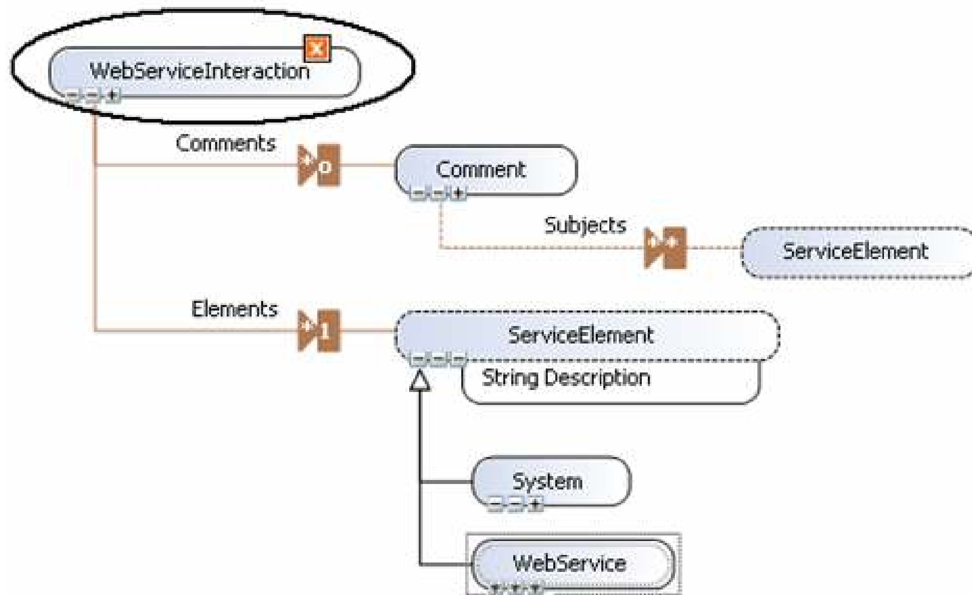


Figure 12: VS DSL Designer, XML root

The DSL has to start with a class that is the XML root, as I described earlier in this thesis DSL's are described with text, and Microsoft has chosen to use XML to do this. The "XML root" is also the start of the XML description of the DSL; you can almost say that it is the title of the DSL. Classes can then be added as we want to, but they will not be a part of the finished DSL unless they are connected to the XML root in some way or another.

5.2.2. Relationships

To tie the classes that are made together we need some way of making relationships between them. There are 3 different relations that can be made between the classes and some relations can even be made on the same class.

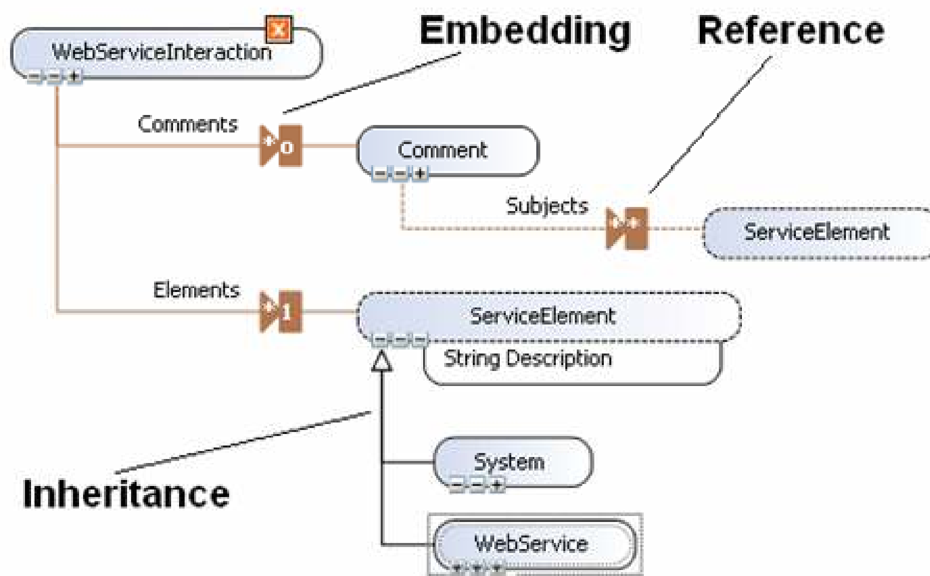


Figure 13: DSL Designer, relationships

5.2.2.1. Embedding

Embedding is used when we want to insert new classes that "naturally follow" the previous class. The consequence of this can, when using the DSL, is seen as something that is "inside" or owned by some other previous defined object. A class that is embedded from the XML root can be seen as it is owned by the XML root. A class that is embedded from a class later on can be seen as e.g. a compartment inside a model.

One class can only be embedded from one other class. This is though logical as we can't have models that are inside models that again has the first model inside itself again. There have been some discussions on this topic on the discussion forum for the Microsoft DSL tools [DSLFOR].

5.2.2.2. Reference

References can be used between classes or a class can have a reference to itself. References are actually the connectors that can be made between the models in the finished DSL. Meaning if you got to classes that can be translated into two different models you can make a reference from one of these to the other in the DSL designer and specify how they connect to each other by defining this in the XML description of the reference (See the 5.2.4 *Connectors* later in this chapter for more information about how this is done). You will then be able to draw connections between the two models when using the DSL.

5.2.2.3. Inheritance

Inheritance is used to inherit descriptions and other details from other classes. Inheritance really works just like inheritance works in normal programming languages like C# or Java, meaning the classes that inherits from another class receive or has all the same features as the inherited class.

5.2.3. Models

Models are created from classes and the XML definition of the model. There has to be both a class and a coherent definition of that class to get a complete model. The XML definition is actually the design of the model, here we can describe if we want a square, circle, compartment shape, etc.

5.2.3.1. Shapes

Shapes are created in separate files and are actually quite significant to make from scratch, so this is normally not done, but there are some pre-made shapes that can be used. When the shapes are defined they are easy to use there only has to be a reference to the file that creates the shape.

```
<shapeMap>
	<class>(…)/WebService</class>
	<iconMaps>
	</iconMaps>
	<melCollectionExpression>
		<roleExpression>
			<role>(…)/WebServiceInteraction/Elements</role>
		</roleExpression>
	</melCollectionExpression>
	<shape>(…)/Shapes/ExampleShape</shape>
	<textMaps>
		<shapeTextMap>
	<textDecorator>(…)/Shapes/WSShape/Decorators/Name</textDecorator>
		<valueExpression>
			<valuePropertyExpression>
			   <valueProperty>(…)/WebService/Name</valueProperty>
			</valuePropertyExpression>
		</valueExpression>
		</shapeTextMap>
	</textMaps>
</shapeMap>
```

Figure 14: DSL Designer, defining shape

5.2.3.2. Compartment models

Compartment shapes are shapes that have "compartments", meaning that you can insert classes in these compartments. Showing what a model contains isn't easily done with just normal squares. Describing a class in a normal programming language e.g. can be shown as just a square, but that does not give us any information about how the class is built, what it is containing and so on. Using this compartment shapes can therefore give us the same kind of abstraction level, while it still gives us a lot more information about the different models and what they contain.

```
<class>(…)/WebService</class>
    <compartmentMaps>
       <compartmentMap>
      <compartment>(…)/Shapes/WebServiceShape/Compartments/WebMethod</c
ompartment>
      <melCollectionExpression>
          <roleExpression>
(…)

<shapes>
  <compartmentShape name="WebServiceShape" initialWidth="2.0"
initialHeight="0.75" geometry="Rectangle">
    <decorators>
      <shapeText  name="Name" position="Center"
defaultTextId="WebServiceShapeNameDecorator"/>
    </decorators>
    <fillColor color="lightblue" variability="User"/>
    <outlineColor color="black" variability="User"/>
    <compartments>
       <listCompartment name="WebMethod" captionId="WebMethodes">
         <compartmentFillColor color="white" variability="Fixed"/>
         <titleFillColor color="lightblue" variability="Fixed"/>
       </listCompartment>
    </compartments>
  </compartmentShape>
```

Figure 15: DSL Designer, Compartment shapes

5.2.4. Connectors

Connectors are created by both the reference relationship and XML description. The XML description of the connector is the vital one, here we can decide which direction the connection has to be or if it can connect both ways. The design of the connection can also be chosen, and this has to be done by making a reference to a resource file in the project where the picture of the connector has to be described.

```
<connectorMaps>
      <connectorMap>
      <class>(…)ReplenishingService/CommentHasServiceElement</class>
            <connector>(…)/Connectors/ExampleConnector</connector>
            <sourceMap>
                  <modelNavigationExpression>
                        <roleExpression>
                          <role>(…)/ServiceElement/Comments</role>
                        </roleExpression>
                  </modelNavigationExpression>
            </sourceMap>

            <targetMap>
                  <modelNavigationExpression>
                        <roleExpression>
                           <role>(…)/Comment/Subjects</role>
                        </roleExpression>
            </targetMap>
      </connectorMap>
</connectorMaps>
```

Figure 16: DSL Designer, Defining connectors. XML

The figure shows how connectors are defined with a sourceMap and a targetMap, which both describes the allowed models that are source and target.

5.2.4.1. Allowed models

A reference relationship might have relations to many different classes, if there is a reference relationship with a class that has many other classes that inherits from it, we can adjust the relationships by setting which classes that the class can be connected to. This is done in the XML file as seen in Figure 17.

```
<connector name="ExampleConnector">
      <color variability="User" color="black" />
      <dashStyle variability="User" dashStyle="dash"/>
      <decorators>
            <connectorText name="Label" position="TargetBottom"
defaultTextId="DefaultLabelText"/>
      </decorators>
      <source>
            <permittedShapes>
                  <shape>(…)/Shapes/ExampleShape</shape>
            </permittedShapes>
      </source>
      <target arrowStyle="EmptyArrow">
            <permittedShapes>
                  <shape>(…)/Shapes/ExampleShape</shape>
            </permittedShapes>
      </target>
</connector>
```

Figure 17: DSL Designer, defining allowed connections

5.2.4.2. Multiple connections

There are possible to have multiple connections between models. By multiple connections it not only meant that one model can connect to many different models of one certain kind, but if we define our DSL in the right way we can let models connect to many different kind of models. This as many thing else need synchronization between the visual design and the XML design. For each model we want to connect to we need a reference by the use of the reference relationship (remember inheritance also counts on reference, meaning if we have a relationship to a class that other classes inherits from they will also have the same relationship). We also need to edit the XML file and set the permitted shapes as seen in Figure 17. Here we can add the shapes we want to have a connection with.

5.2.5. Enumerations

An enumeration tool is also available in the DSL Designer. This tool lets you create your own types, which is a great when designing DSLs. Instead of letting the user type in what e.g. a Web Service is going to return, we can let the user only be allowed to chose between some predefined types. These enumerations lets us build a more user-friendly DSL which also is much easier to transform into code, since it reduces the chance of create models that doesn't use valid types.

5.3. DSL toolbox

When we are going to use the DSL we need to have a toolbox from where we can select the models that we are building the application. These have to bee added to the toolbox somehow, and it's done by adding the models and connections under the tag *toolbox* in the XML file.

```
<toolbox>
  <items>
    <shapeTool iconId="ExampleShapeToolBitmap"
captionId="WSShapeToolboxCaption"
contextSensitiveHelpId="ExampleShapeHelpId" order="0">
        <shape>(…)/Shapes/WebServiceShape</shape>
     </shapeTool>
     <shapeTool iconId="CommentShapeToolBitmap"
captionId="CommentCaption" contextSensitiveHelpId="NoteHelpId"
order="1">
         <shape>(…)/Shapes/CommentShape</shape>
      </shapeTool>
    </items>
</toolbox>
```

Figure 18: DSL Designer, defining toolbox

5.3.1. Names, Bitmaps and order

As you can see of the figure above there are some variables that need to be set. These variables are used to determent the icon of the model, the name that the model or connection has in the toolbox, and the order of the models and connections. The first is linked to a bitmap file which is the icon that will be used in the toolbox. The second is connected to a resource file, *Designer.Resource.res,* which again we must set the output text to the key that we defined in the XML file (see the figure below). The *order* is simply the order that the different models and connections are shown in the toolbox.

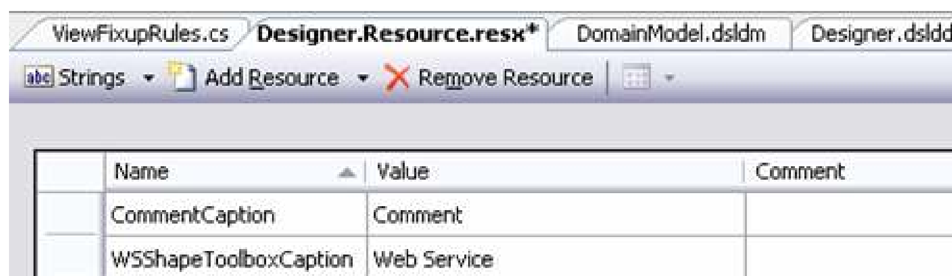


Figure 19: DSL Designer, Resource file

5.3.2. Transformation

The transformation of the DSL design to the actual DSL is done by first generating code from the DSL design and then to compile that code. There are a lot of errors that can be done in case of both synchronization between the XML file and the visual designing of the DSL.

5.3.2.1. Models and XML files

The models need to have the right synchronization between both visual design and XML file. If the model (shape or connector) is only defined as a model in the XML file but never connected to any class or relationship, we will get the model in the toolbox. Using the model however will be impossible since it is not connected to the XML root (See 5.2.1 Class) in any way there is no way of using it in the design. Only models that are under the XML root can be used in the design.

5.4. DSL deployment

When you have created your DSL and written the transformation script(s) you might want others to be able to use the DSL as well. To do this it’s needed to deploy the DSL and the transformation script. It can then be installed on other computers that use Visual Studio 2005.

5.4.1. Export Template

The first things that need to be done is, of curse, to build the DSL and transformation scripts. When this is done we need to export the templates, and this is done by using “Export Template” wizard from the debugger instance of Visual Studio. The wizard will, based on your choices, create a .zip file containing what’s needed for deploying the DSL.

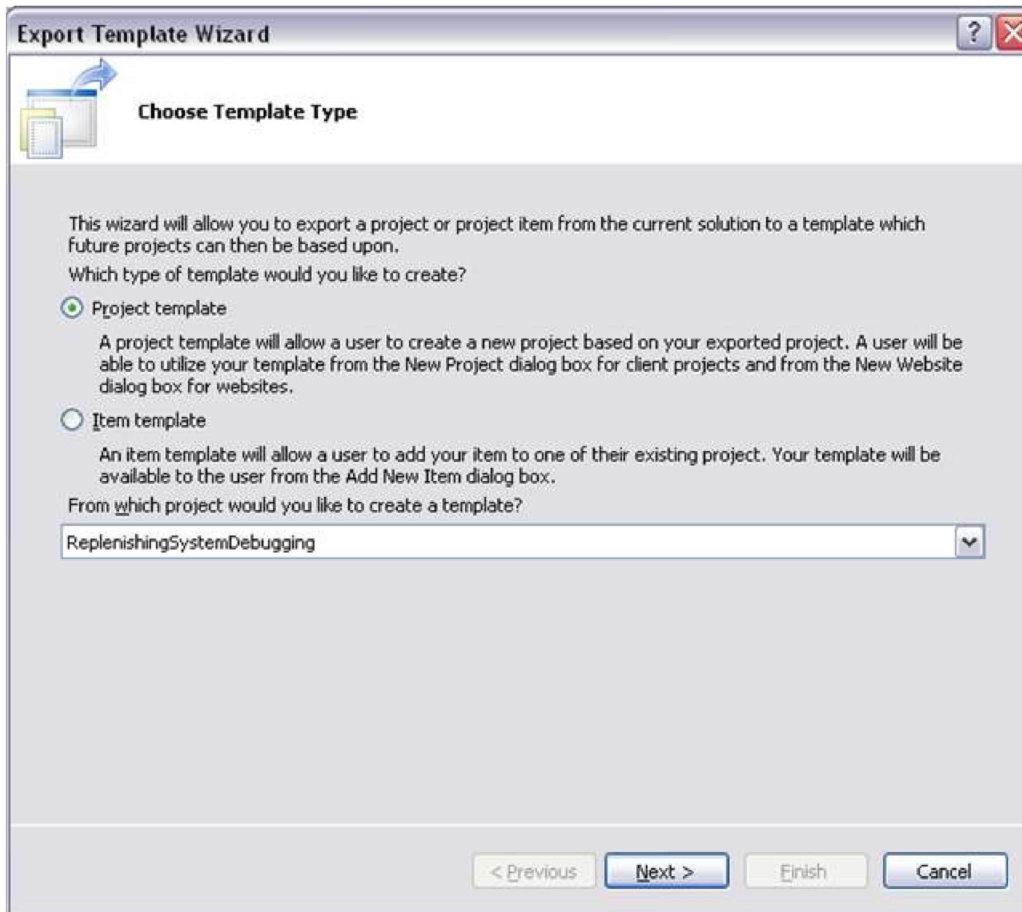


Figure 20: DSL Deployment, Exporting Templates

.

5.4.2. Domain Specific Language Setup

After the "Export Template" wizard has been finished and we got our .zip file, we need to create a Domain Specific Language setup project. This project needs to be added with the DSL solution (along with the two already existing projects). The .zip file we got from the wizard needs to be added to this setup project and other configurations can be done. The setup project is build with templates as well, and thus the custom tool needs to be run on these templates to transform them into code. The setup project is now ready to be built, whereas 5 files are created. These files can be used to install the DSL on any other PC with Visual Studio 2005.

5.4.3. Install DSL

Installing the DSL is done by running the setup file that was created by the setup project of the DSL solution. To start a project we need to we need to look under templates when creating a new project.

To uninstall the DSL we need to go to the *control panel* and *Add or Remove Programs* where we can find the DSL that we installed.

Figure 21: Installing the DSL

5.5. "Helper plug-ins"

The DSL designer in Microsoft's Visual Studio 2005 is quite new so there is of course some improvements that can be made still and thus there have been made some helper tools that can be installed with VS 2005 which improves the DSL designer.

5.5.1. DSL Dm->Dd

The DSL Dm->Dd tool (Dm is short for a file with the .dsldm extension, while Dd is short for .dsldd. These files are the visual designer and XML design file) complements the Microsoft DSL designer by helping maintaining the consistency of the .dsldd and the resource files from information that is located in the .dsldm file.

It might be easy to get lost with all the XML which connects to the different models. This tool [DSLDM] helps in the way that it translates the XML into figures that are much easier to read than raw XML.

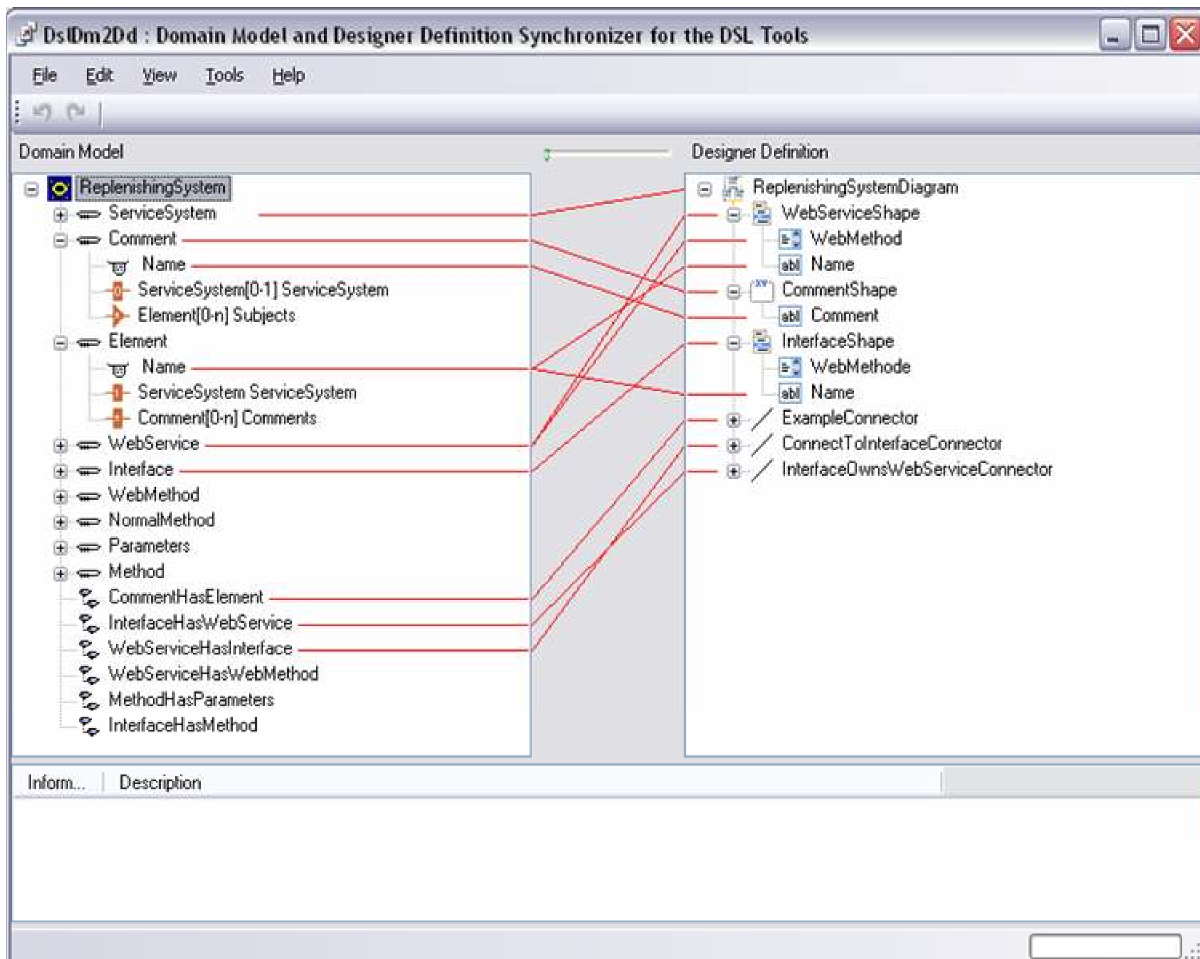


Figure 22: DSL Dm->Dd helper tool

5.5.2. T3Colorizer

There is not been added any real editor for the text template transformation files in VS 2005, the files are just text files which really are as easy as editing a file in notepad. There have been created this T3Colorizer [T3C] which shows the output text in blue background while the template coding is white. Text template transformation keywords have a green color.

```
<h3>ConceptBs</h3>
<#
        foreach ( Element b in this.Modele.Elements )
        {
#>
        <p>Name: <#=b.Name#></p>
<#
        }
#>
</body>
</html>
```

Figure 23: T3Colorizer helper tool

These small improvements help a lot to get an overview of the files, but it isn't perfect. Microsoft is however planning on releasing a text template transformation editor which will improve the look and feel a great deal, thus improving the programming speed.

6. Transforming DSL Models to usable Web Services

Designing a Web Service with the help of DSL can be done in many ways. A Web Service can be built in many different ways, and it can have dependencies to other files as well.

When building a Web Service (from the .Net point of view) you has two possibilities, either you create an *inline* Web Service or one that has a *code behind* file. Inline means that everything is put into just one file (.asmx file), and since Web Services in .Net uses so-called *Just-in-time* compiling it can't interact with other classes in the same way as *code behind* Web Services can. These are compiled beforehand and thus can interact with other classes just as a normal class library can. The asmx file will only reefer to the *code behind* file so when it's being compiled it has all the relations that it needs, in the pre-compiled code.

Figure 24: Inline Web Service

The figure above shows an inline Web Service, and as you can see, everything is written inside this one file.

Figure 25: Code behind Web Service

The figure above shows the whole asmx file for a Web Service. All the rest of the code is written in the codeBehind.cs file, and is pre-compiled.

When it comes to designing Web Services in the DSL Designer, I found out that the best way of designing them, either I would use *inline* Web Services or the ones with code behind, was to just be a box with the ability to add *web methods*.

Figure 26: DSL Web Service design

Then I could let the writing of the custom code design how the Web Service should be translated into code. If this is the best way to go I shall not say, but at least it does not set any limitations on how the author of the transformation scripts can transform the models into code. Domain Specific Languages have the purpose of letting you describe any kind of domain in the way you self chose. So when trying to describe a general Web Service the figure above might be the answer, but if there is a company that has its own special way of building their Web Services, depending on how specific we want to be, one could design it in a whole different way.

6.1. Model to text transformation

Transforming the models into actual Web Services is done by the use of *custom code*. This script translates the models to code by using the information that is stored inside the models. Depending on how we want to design our Web Services, *inline* or *code behind*, we first need to create some files where the output from the transformation will be placed.

6.1.1. Templates

Custom code is written in template files, and for each template file that is used on the modeling there will be generated a new file to the project based on the custom code. These are the "translation" between the visual modeling and abstract view to concrete code.

6.1.1.1. Custom code

When transforming the models into code we need to figure out how we want to transform the model into actual code. The nice thing with using domain specific models is that we can basically put any information we want into these models. Using this information is essential when writing the custom code.

When designing output code for Web Service, we must of course figure out how we want to design our Web Services. Any special features that we want to add to every one that is built? If we want to add some special security features or if we want to add authorization in the case of people having to log in to be able to use the Web Service, this can be done easily. Problems will occur if we want to build some without these special features with

the same DSLs. This is the nature of **domain specific** development; you build tools or languages that are perfectly fitted to a special way of developing software or a specific software family.

6.1.1.2. Defining output files

Let's say that the one model is one Web Service. That means that for each model we need to generate at least one new file. We also need to figure out what kind of files we want. Normally a template only produces one single file, but there are ways of making new files within the custom code. This is done like we normally do when creating new files in with the help of the programming language. This isn't the best way of solving the issue of multiple files, but there will hopefully be added some better alternatives by Microsoft in the future.

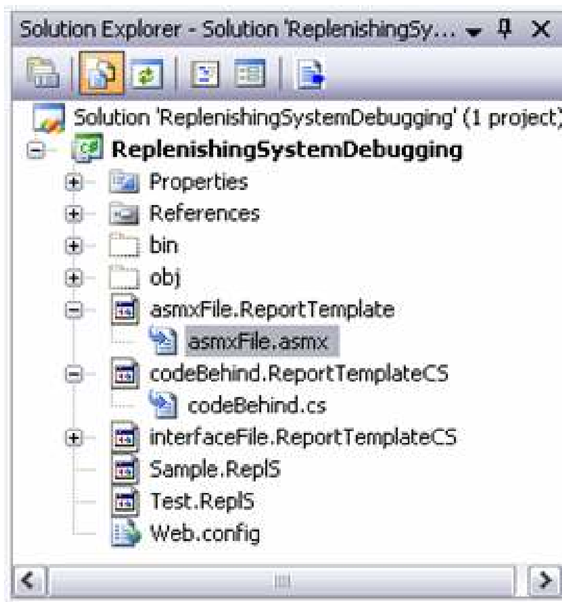


Figure 27: Template files

When creating more than one Web Service in our project we need to add files for each Web Service that we want to use. If we decide to create *inline* Web Services we need only to add an asmx file for each new Web Service. If we use *code behind* files we still need to add a new asmx file for each new Web Service, but as of the *code behind* files we do not need to add new files, but we can just add new classes to the same file and refer to the class.

6.1.1.3. Multiple templates generating one solution

There can be multiple template files that can generate one project or solution. Basically there are many files that can be in one project and for each of these files there can typically be a template file. For generating Web Services we can have different template files for asmx and C# files.

6.1.2. Model compartments

Using compartment shapes (see 5.2.3.2 Compartment Models) for defining the web methods is a great way of getting an abstract but also informative model.

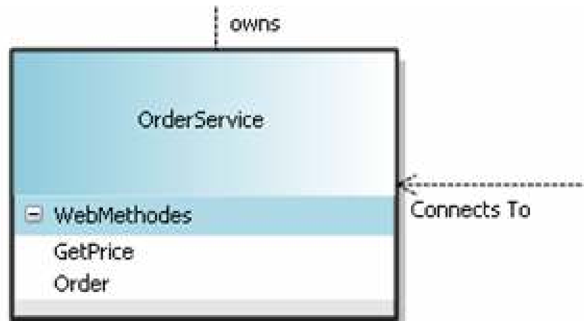


Figure 28: Model Compartments

This model shows how easy it can be to describe a Web Service and which *web methods* it has.

6.2. Model explorer

The visual or graphical design can and should also be aided by the modeling explorer. Not everything in a project is easy to describe with models and are often better described with text.

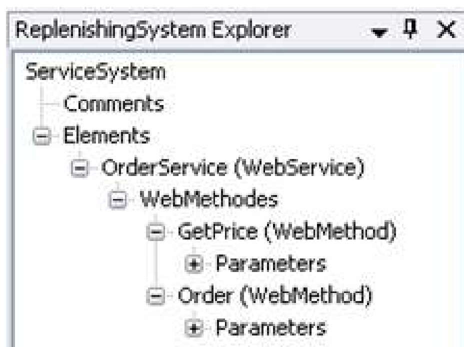


Figure 29: Modeling explorer

As you can see from the figure above the whole modeling will be described in a hierarchy way. “Parameters” as shown on the figure will not be visual on the model of the Web Service; it will only be visible in the modeling explorer. This also increases the abstraction level of the modeling, with not letting every little detail be visible in the models.

6.3. Using the Domain model design in other projects

When the DSL design is finished and the custom code for the modeling is written then we can export the template (see 5.4.1 Export template). This means that we can now, when starting a new instance of Visual Studio 2005, start a new project with the Domain Specific Language that we have described and exported. The new project will now have the templates that generate code and a model designer.

6.4. Compiling generated code

When we have built our Web Service we need to build the code. The files that are generated are automatically added to the project, and will thus be added in a build. When building Web Services in Visual Studio we need to configure the project. Normally when creating Web Services in Visual Studio there is a special project type that can be selected that fix everything with building and web server. This has do be done manually with the DSL type of Web Service. As of now we cannot select or configure that the DSL should be a Web Application, this will hopefully be added in future versions.

To build an executable Web Service we must first change the dll output path, and then we need to configure IIS.

7. Case study

This chapter illustrates the use of Microsoft's DSL tools in a case study. A typical business case with a producer and a consumer will be described.

7.1. Case description

Companies have various ways of ordering new things when their inventory is running low. Administrating when to order and how the "things" should be delivered can often be time consuming. There are seldom very good solutions on this subject that are easy and fast to use.

The idea of this case is to create a relationship between consumer and producers via Web Services. A customer typical has many producers where he can select which products he wants, and he can also easily check and compare prices.

In this case we will look at a system that handles the relationship between a car producer and a few car-seat producers and a transport ordering service.

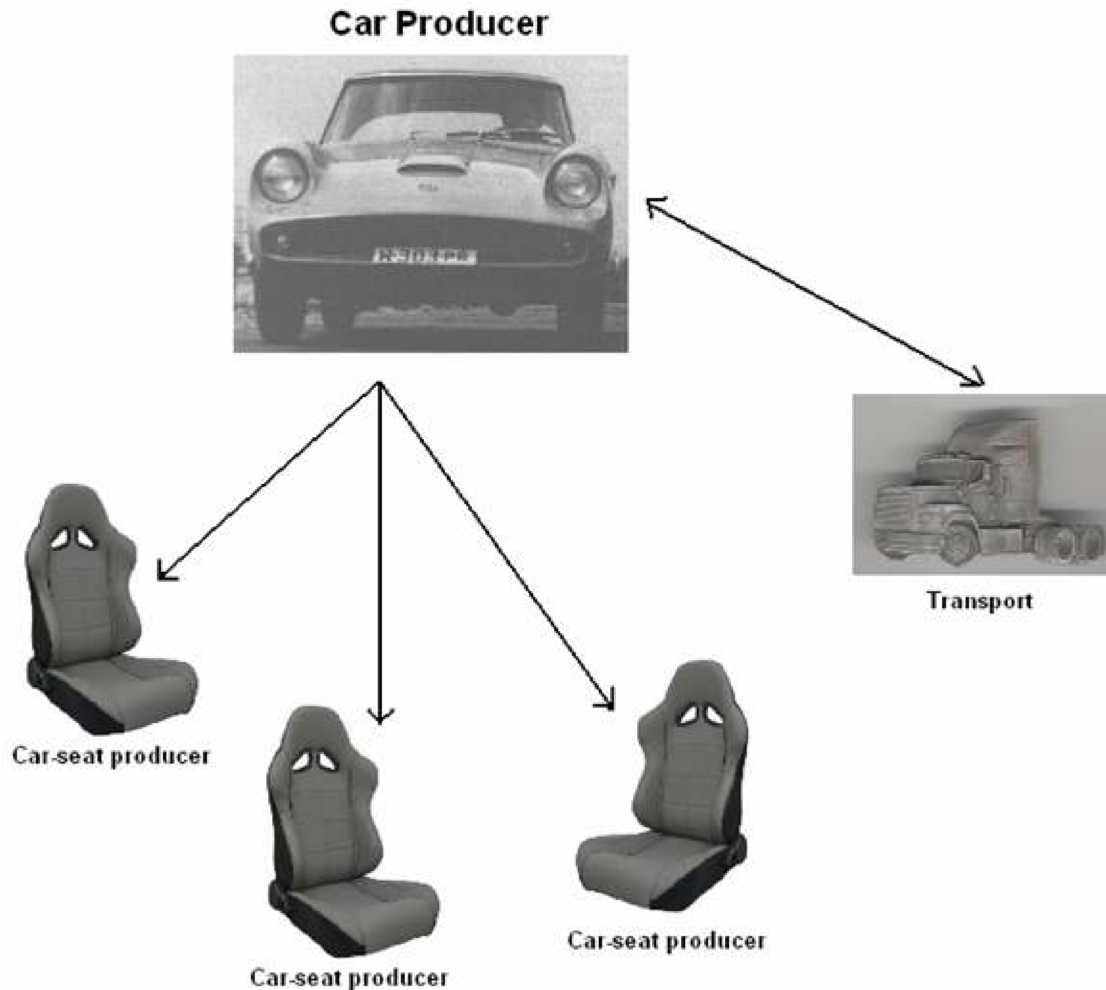


Figure 30: Consumer/producer relationship

A typical scenario could be divided into these stages:

1. A low inventory alert goes of telling the system that there is a need for new car-seats.

2. The system sends out a message to the car-seat producers that it has in its database and asks for a price.

3. The system compares the prices from the different car-seat producers and figure out which one that got the lowest price.

4. When the lowest price is found the system orders new seats from the car-seat producer through its Web Service.

5. When the order is confirmed, the system orders transportation for the seats through a transportation company.

Of course we could also check the combined price on both the seats and the cost for delivering them. That is not implemented in this case.

7.2. Scope

The development of the software for this case will consist of the different Web Services that are used by each party that is involved. Interaction between the different systems will also be documented. The methods that shall call the Web Service will not directly be connected with the Web Service; it will however have everything it needs to be able to do the actual call.

7.2.1. Requirements

The following requirements have been added:

1. Authorization on the Web Services. There can be added a security measurement with a login part on the Web Service.

2. Web Services that works according to the [WS02] standard.

3. Interaction between Web Services and another system or application.

7.2.2. Limitations

The following limitations are assumed in the case:

1. Calls between a system and a Web Service are not actually done, from which method or when the Web Service should be called should be done manually after the code is generated. Adding the connection between the system and the Web Service is though made so that it is ready to be used.

2. Manual configuration of the web server is needed to be able to use the Web Services.

3. Details about how orders or information sent with the Web Services is not relevant to the case and therefore not described.

4. When and why a Web Service is called is not specified, neither is the content of any method.

5. The different producers are not made as single files, but backed together in the same file, but with different namespaces and classes. The main focus is on the Web Services.

6. WSDL documents are generated automatically be .Net. To see a WSDL document that is on the .Net platform you can use to internet explorer and type the address

to the Web Server followed by ?WSDL (e.g. http://aplace.not/OrderService.asmx?WSDL).

7.3. Designing the domain models

Designing the Domain Specific models for this case have been done in a more general way than it has to be. I have focused on designing models that can basically be used to develop any type of Web Service.

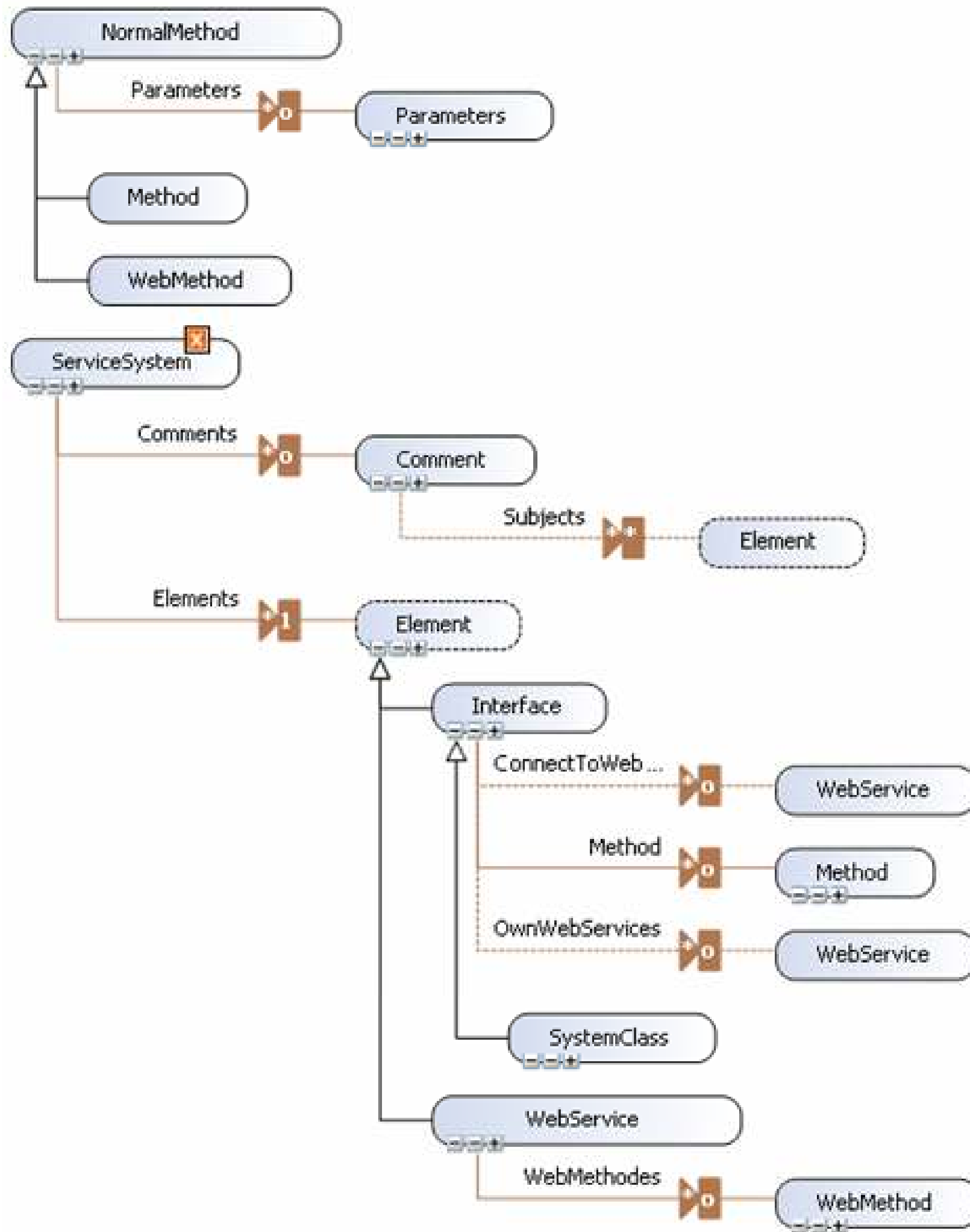


Figure 31: DSL Design, Case

7.3.1. Defining detail level

DSL tools have been made so that it would be possible to create real domain specific modeling. This means that we can basically design the DSL in the detail that we want. More detailed DSL can better give a good and very detailed description of the software that we are developing.

In our case we are just designing the use of Web Services, and thus we do not need extreme detailed models. A Web Service can be described in a good way by a shape which shows which *web methods* it has.

I have chosen to design the DSL models like on the figure above (Figure 31: DSL Design, Case) because it I think it will be easy to understand how the different models can connect to each other and how they work. I also think that the DSL gives a good abstract view of how it would be if it were just normal code.

I will give a short explanation on why I used the different classes and relations between them:

ServiceSystem is the XML root, which means that everything that follows from this class is in the context of the ServiceSystem. It also means that our finished DSL is called ServiceSystem and all modeling is done here.

Comment is a model that I've chosen to add so that the users of the DSL can comment their models. Comment refers to the class Element which means that it can connect to Element types (Comment elements).

Element is not actually a model in the DSL, but I've chosen to add it because all classes that inherits from it will inherit the same behavior (*Comment* can be added to all models that inherits from *Element*).

WebService is a model which I have defined to be a compartment shape (model design) in the XML description, which means that each class that is embedded from this *WebService* can be inserted into the compartments in the finished shape.

Interface is a model which in the same way as *WebService* inherits from *Element*. *Interface* is used to describe a real interface of a class. *Interface* is also defined as a compartment shape, and classes that are embedding from the *Interface* can be added in compartments in the DSL model.

SystemClass is a model which inherits from *Interface*.

NormalMethod is as you can see not connected to *ServiceSystem* in any forward way; this means that *NormalMethod* can not be used as a model in our DSL. It still plays an

important part of the DSL design though because other classes can inherit from it like *WebMethode* and *Methode* do.

WebMethod and **Method** are defined as compartments in *WebService* and *Interface*/*SystemClass*. This will give us models that fairly easy to understand. A Web Service can have many web methods, and these web methods are identified by *WebMethod* compartments in the *WebService* model.

7.3.2. Compartment shapes

For the methods in both a Web Service and a system class I have used compartment shapes. The compartments lets us give an abstract design that is easier to read than if we use a new model for each method that is used in a class.

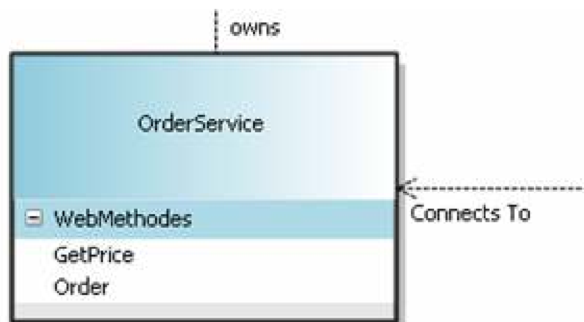


Figure 32: Compartment shapes

7.3.3. Connectors

The connectors that I have defined are connections between a system and a Web Service and between a comment model and any other model. A Web Service can be owned by one and only one system, but many different systems can connect to the same Web Service.

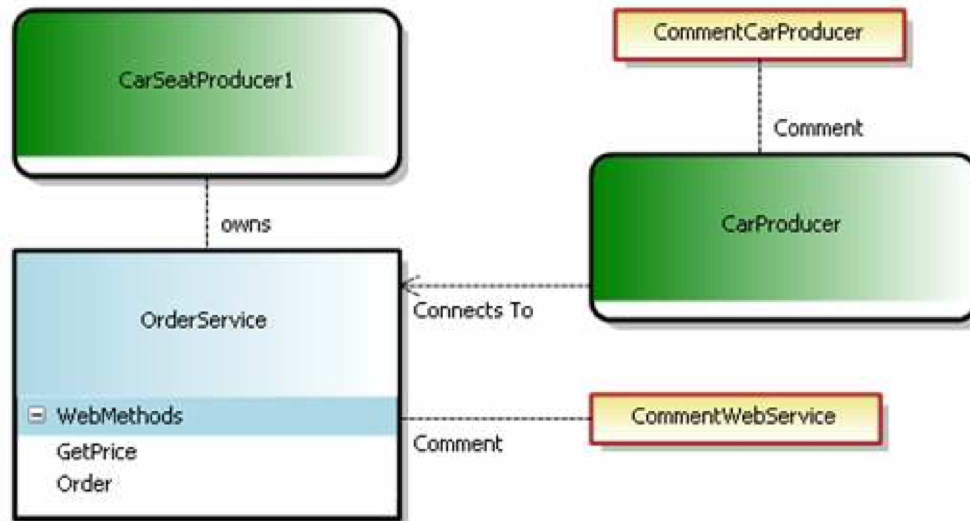


Figure 33: Different connections

7.4. Using the designed models

When all the models and connections are designed in the Domain Specific Model Designer we can start a new instance of Visual Studio and use these models and connections to design software.

7.4.1. Designing a system

Designing the whole system and how the different models connect to each other is basically the "programming" of the system.

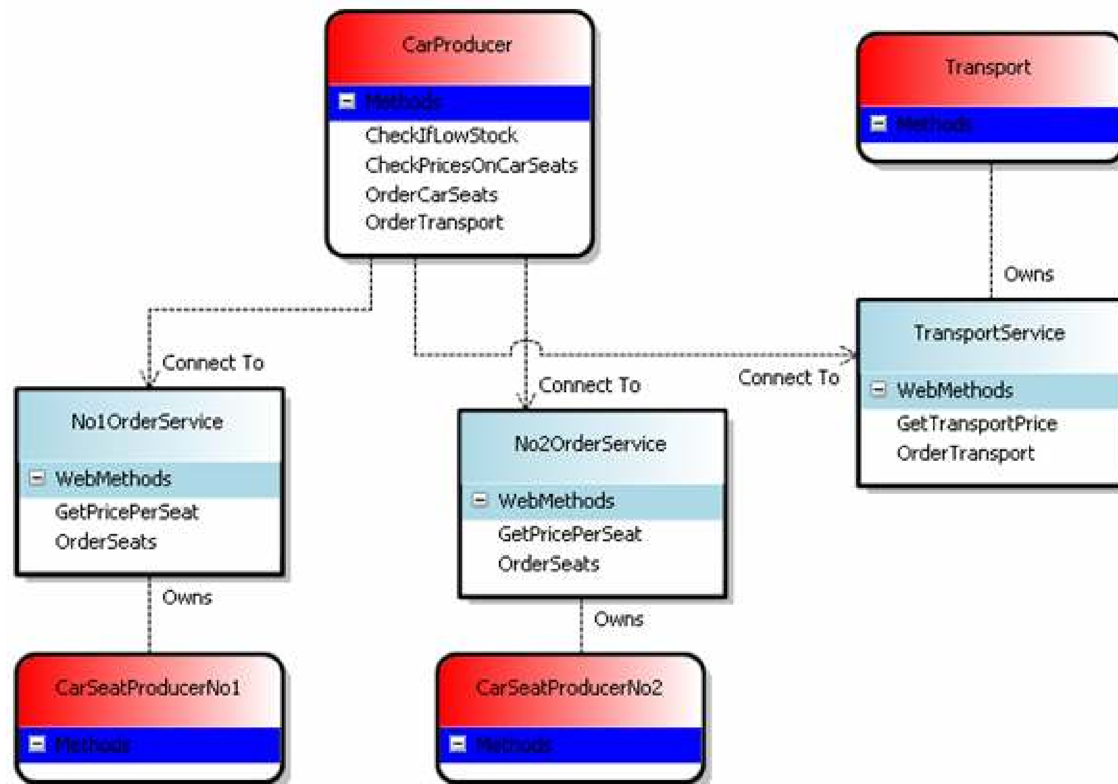


Figure 34: Using models to build the case

As you can see from the modeling above we do not stress how the different other systems are designed (CarSeatProducers and Transport). The main focus on this case is to show how different systems can communicate between each other with the help of Web Services.

7.4.2. Adding methods and other application specific data

Both the Web Services and the system class models are so called compartment shapes, which mean that there can be added inner information in these compartments. Compartments are perfect for describing which web methods that is available for the different Web Services.

Other configurations like authorization on the Web Service can also be configured, but these details are better of hiding from the abstract view of the modeling. These details can be found in the model explorer and properties section.

7.5. "Programming" the code generators

The transformation of the modeling is done by transformation scripts. These scripts transform the models into code that can be compiled.

7.5.1. Custom tool

Files that we want to generate code from needs to have a custom tool that can translate the model into code. TextTemplatingFileGenerator is the name of the custom tool that is used for translating the models in this case.

7.5.2. Custom code and Templates

The custom code that needs to be written I have chosen to divide up in different templates. This is mainly do to the fact that we got several different file types, and using one template for each file-type gives us a nice a structured way when designing our custom code.

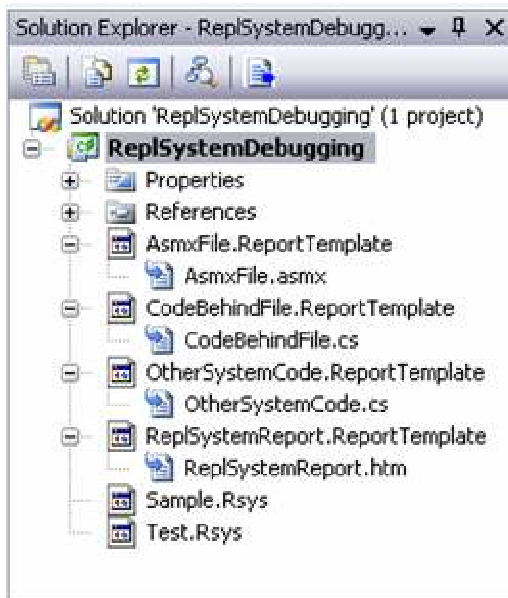


Figure 35: Custom code, template generated files

As you can see from the figure above I have chosen to use four different template files (including the pre-made report file that is an .htm file). The reason for using so many templates is to get a better view of the different files that are used. There are basically one template file for each file "type".

Explanation for each template file:

AsmxFile creates a *code behind* asmx file, which means that it only refers to a file and a class in that file.

Figure 36: Code behind Web Service

The asmx file refers to the file codeBehindFile.cs where it again uses the class TransportService class which lays in the namespace Transport.

CodeBehindFile generate the actual Web Services with the *web methods* and other information that is stored in the models.

OtherSystemCode generates the rest of the code that is needed in the project. This is code like plain classes from the car seat producers. They will get a namespace though which will be connected to the Web Services that they own.

ReplSystemReport is a sample template which follows with the project. This has no meaning for any of the code. It will only generate an htm file.

7.5.3. Output

When we run the custom tool, the models are generated into code

```
using System;
using System.Web.Services;

namespace CarSeatProducerNo1
{
    public class No1OrderService : WebService
    {
        [WebMethod]
        public void GetPricePerSeat()
        {
            throw new NotImplementedException("Not implemented yet");
        }
        [WebMethod]
        public void OrderSeats()
        {
            throw new NotImplementedException("Not implemented yet");
        }
    }
}
namespace CarSeatProducerNo2
{
    public class No2OrderService : WebService
    {
```

Figure 37: Generated Web Service code

As we can see of this model I place every Web Service (code behind) classes in the same file while separating them with namespaces that their owner also will have.

7.5.4. Getting compileable code

Since some of the files that we are producing are Web Service files, we need an environment in which these Web Services can be executed and are available for customers. This will include a web server on which the Web Services can be placed. When creating Web Services with the built in web application projects in Visual Studio 2005 every little configuration that is needed to be done both in the way that the Web

Services are built and the way that the web server should handle it. All this has to be done manually when creating Web Services like we do with the use of DSLs. Hopefully there will be added features later on that lets you decide if you want to use a web application project or just a normal project.

7.6. Results

The results of the transformation are different files that can be compiled into a system that lets customers contact each other and order products when needed through the Web Services

7.6.1. Web service (.asmx)

The Web Service and the .asmx file I found out had to use pre-compiled code in order to communicate with the other files in the project. Thus we needed an .asmx file and a code behind file for each Web Service that is made. In this case I made one asmx file for each Web Service and one single *code behind* file that contains all the classes of the different Web Services. That means that every asmx file refers to the same file, but uses different classes and namespaces within that file.

7.6.2. “Code behind”

The *Code behind* files is used when we want to pre-compile the Web Service. A normal Web Service uses *just-in-time* compiling which means that it is compiled the first time anyone tries to use it.

The *code behind* files does not have to be separate files. There can be separate classes within the same file as I showed in one of the figures above. In this case I have placed all these Web Service classes within one file. This is mainly because there is no good way of creating multiple files from one single template file.

7.6.3. WSDL

The Web Service Description Language files can be generated when we got a finished and working Web Service. An own template or translation script is not needed for this, at least not in our case. WSDLs are used to describe a Web Service, and is normally only designed if we want to use so called *Contract first* [TTWS] [CFWS] services, which means that a customer can use this WSDL while the Web Service itself can be change as long at it sticks to the definition of the WSDL. They basically have the same functions as *interfaces* which can make a more abstract connection between the customers that uses the Web Service and the Web Service itself.

7.6.4. Other C# code

Other code that is developed is the code for the producers and consumers, the ones that use the Web Services. This is not all very important code for this case, as this case is used to show how Web Services can be used between different consumers and producers.

7.7. Fulfilling the project

As I have explained before the level of specification that can be made with the DSL Designer is enormous. We can basically describe every little code sentence with the help of models. This would thus not be very useful, it would be better to just write the code manually. The main goal is do try to build the framework of the solution which has all the relations (namespaces and what needs to be included) already present when we start on the project.

What is missing in to get the system up and working? Visual Studio 2005 normally has its own special "Web Service templates" that automatically places the Web Service and its dependencies on the web server, and configure it. I have not found a way to this with the DSLs so this have to be done manually. To get the Web Services to work we need to add the location of the Web Services to the web server, and we also need to set the folder in which the Web Services are to a *web application*.

The Web Services will now work as normal Web Services, but as I said, the web methods have been left empty. These have to be filled with code.

The rest of the files are created in the same way as the Web Services, meaning that for every relationship that is going to be used there have been added or included what's necessary for the relationship. If we have designed that one class is going to use a certain Web Service the reference is already defined, so to use it we do not have to figure out where it is and so on.

7.8. Other Microsoft DSL projects

There are of course not to many DSL projects that have been released since tool hasn't been around for that long yet. There have only been a few days since the release of the full version of the tool, so people have not started populating their DSL designs yet. Some projects have been populated though, and there are 4 different DSL designs following the Microsoft DSL Designer. These are:

Activity Diagrams – Allows you to model activities in the sense of float diagrams. You would typically have a start and a finish point, and have signals that come in and out of this system.

Class Diagrams – Allows you to design classes with interfaces and associations and interfaces. It is in many ways just like modeling classes in normal UML.

Minimal Language – Creates a simple language for you from where you can build your own DSL. This Minimal Language only has a root class and a embedded class that refers to itself.

Use Case Diagrams – this is basically a normal use case diagram as seen in many other modeling languages. The thing that makes this use case diagram unique is that you can decide how you want to translate it into actual code.

These DSL designs are only the design of the DSL, there has not been added any custom code that transform the models that you might make into any code. These transformation scripts are left for you to write.

8. Other similar technologies

There are many other technologies as well that tries to decrease the time developers has to use creating applications, improve the possibility to maintenances the software that is already built and improve the way that applications are developed by using different abstraction levels.

In this chapter there will be shown how some of the tools and languages that are out there work, although I will not go in to much detail.

8.1. Model Driven Architecture (MDA)

MDA was originally started by Object Management Group (OMG) [OMG05] back in 2000. MDA is based on modeling, just like DSL. MDA however use the already defined language of UML [UML04] which also is developed by OMG.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns [MDA03]

8.1.1. Basic concepts

The basic concepts of MDA can more easily be explained if we try to split it into the main parts that it is build up by and around. I will only touch into the parts but you can read more about it in the MDA guide [MDA03].

Platform has become more than it was before. When you got your operating system you got your platform, but it does not work that way anymore. Now there can be many platforms that build on top of an operating system. Examples of these are .Net, J2EE or CORBA. MDA lets you build both platform independent (PIM) and platform specific models (PSM). PSM are used when we want to generate code to the specified platform.

Architecture is taken from designers that design buildings, bridges and so on. As the abstraction level continues to rise we start to see the similarity between a normal architect and a software architect. The software architect draws the software with the help of models and connectors and then the transformation of the models see to it that the application is made.

Model is normally something that can be just a figure or almost every thing. When you model a car you can say that every little part is a figure and you can place all these different parts together to make a complete model of a car. But if there is no information behind or connected to these parts, we really haven't done anything but draw the car. If you then would make the car you would then try to figure out how each part work, and you can find out that some parts does not really go to good together and so on. If you tie each part together with information about how they work at the design level it will be much easier to build the car after the modeling is done. It is also easier to see which parts that goes together and which does not because this is already defined with the

information that is tied to the models. In the same way we can draw an application with UML.

By **Model-driven** it's meant that instead of developing software with the use of normal programming languages we use models as primary artifacts for developing the software. The models are defined in such a way that it's possible to use them to develop applications or parts of applications; it's also possible to develop entire frameworks with the help of the models.

Views can mean different things within MDA, but the most important are the level of abstraction and focusing on a particular part of the application. Different view can give a show us different abstraction level of application that we are working on. This zooming affect is very useful to be able to see the application as a whole or just focusing on a specific part.

Platform Independent models (PIM) are normally the starting point in the MDA process. Information about platforms is usually not present at this point. PIMs are more likely to be used as the architecture part of MDA and are normally defined by the use of the modeling language UML.

Platform specific models (PSM) have a less abstract level than PIMs and are usually the result of one or more PIMs that has been transformed. These PSM can have detailed information about the platform that they are going to be ported on to. The functionality from the PIM should also be present and mixed with the functionality that is described in the PSM.

Transformation is when going from one model into something new. The transformation could be from model to model, as when going from PIM to PSM, or it can be model-to-text transformation. The goal MDA is to develop applications or at least parts of applications and to do this it's needed to be generated code in some way or another. This generation from model-to-text is done by the use of MOFScripts that basically translates all the models into text or code that again can be compiled or at least be edited.

8.1.2. MDA vs. Software Factories and DSL tools

MDA and Software Factories use the same idea, but there are some important differences that should be important to know about. OMG has developed UML and has therefore it's natural for them to use this as a basis for any modeling whether it's Model-driven development or not. Microsoft has not developed any standard for modeling, at least not in the same degree as UML. OMG also normally operates on more than one platforms, Microsoft has as most people know by now built the operating system called Windows, and Microsoft has also developed a developing framework for windows that is called .Net, so Microsoft (almost) only operate on one platform and they do not need to think so much about platform portability and so on. MDA has also therefore made so called model-to-model transformations that allow you to generate new models from your other models. Microsoft has not introduced such a feature, except from DSL Designer models

to DSL models. The use of model-to-model isn't that current in Microsoft DSL since it has not support for multiplatform. There are no restrictions on what you can do with the models though, if you want to transform the models into new models, and want to write probably very much code, you can do that.

8.1.2.1. UML and MOF

MDA is specifically designed for building applications in UML. This would separate MDA from Software Factories as an approach if we would use plain UML, since UML originally was designed for sketching.

MDA has to use models that can hold more information that just plain UML models. MDA therefore uses MOF and metamodels to define their models. A MOF is a description of a metamodel which again holds the information of a model. According to [MDDD04] MOF defines how models can be accessed and interchanged, in terms of e.g. interfaces defined using OMG's XML Metadata Interchange (XMI) [XMI05].

Models are defined in a similar ways in Microsoft DSL Designer, where we got a visual design of the model and a XML description.

Software Factories can use UML for sketching but will not use UML as source artifacts for that Software Factories use DSL because of their characteristics, smaller languages that are very focused on specific tasks or problems with very well defined semantics which can be very easily tooled.

8.1.2.2. Defining models

There are some big conceptual differences when it comes to how MDA and Software Factories and DSL define their models. As I mentioned above MDA uses platform independent models (PIM) and platform specific models (PSM) when defining models. Software Factories does not use UML the way that MDA uses it, instead of using UML and PIM/PSM the Software Factory figured out that they where better of using Domain Specific Languages to define their models. DSL has the ability to focus on the small details on a specific domain while UML has a more general way of seeing the world. The way that UML has of generalizing makes it impossible to use in certain domains. Software Factories begins with designing the domain specific language, and thus it can be designed to fit any domain.

8.1.2.3. Platform

MDA claims that they can support cross platform portability, "write once, run anywhere" like the claim Java uses. By saying this they effectively claim that UML is a universal programming language, but as most developers that uses UML knows that the "U" in UML stands for Unified. Even java, which in fact is a programming language and not a modeling language, says that they support cross platform portability, but they not achieve portability except at the most basic level. Extensions like Swing has provided a way to

achieve portability, but the cost of not only performance but also usability, you cannot create something that is not supported on all platforms. Also platform integration is reduced greatly.

Although there might be many larger or smaller issues with the use of platform portability it is a great idea, but as platforms are build in very various ways it is almost impossible to make it realized. Think of just making a form like a windows form, here you got some choices, do you want to use the already defined forms that windows uses or do you want to build the forms from scratch. The last choice will result in a form that would not be configurable by the platform it is build on. Let's say that a user has a configuration that all window title areas should be green, all the forms made in the platform specific way will then get green title areas while the forms that are build from scratch would not be changed. This might not seem as a big issue, but there is a lot of ways that platform specified forms behave that the ones that are build from scratch don't. It isn't the way that they are built, but that forms that are built from scratch do not get the same "connection" with the platform as the platform specified form get. If we try to build forms in the platform specific way, we will get trouble in the way that forms from one platform does have some special features that are used and others do not. Making this work properly will result in some serious complex mapping and functions that some platforms might have that others does not, has to either not be used or tried to be integrated one way or another which is not easy.

No matter how you try to solve these issues it is going to be complex, and the more complex it gets the less performance we will probably get. Usability and platform will also most likely be reduced as of the way that I described above.

The question if we still want to support platform portability is then up to the ones that develop languages and tools. What are the important thing, portability or performance and usability?

8.1.2.4. Model to text transformation

Getting something that can be deployed in one way or another is of course the main issue whether we talk about MDA or Software Factories and DSL. How we transform our models into text or code is not just seen as how we write the transformation scripts, but also how we define the models that we use. MDA uses something that they call PIM (Platform Independent Model) and PSM (Platform Specific Model). PIM's are, in short, used to define how the modeling is on a platform independent level, while PSM's goes down on a platform specific level and describes how the modeling should be translated into code to be used on that particular platform.

The transformation scripts are however very similar. MDA uses what they call MOFScripts which in Visual Studio 2005 is called Custom code or Templates. These scripts basically works in the same way with translating the models into text, and they basically use the same syntax, with the important difference that output code is written inside the tags and script-code on the outside in MOFScripts, while in Visual Studio 2005

and ReportTemplate uses the exact opposite notation with the output code outside the tags and the script code between the tags.

8.2. Rapid Application Development (RAD)

Rapid Application Development was originally developed to increase the speed of the developing applications and increase the quality of the applications. RAD was initially developed by James Martin in the 1980s. It used Computer-Aided Software Engineering (CASE) tools which mean that there where developed software tools that helped developers to develop and maintenance software.

Typical CASE tools are:

- Code generation tools
- UML
- Re-factoring tools
- Configuration management tools

When using these languages and tools it’s possible to increase the development speed. The goal of RAD is to capture requirements and turn them into code that can be used as quickly as possible.

The quality of the applications is meet by which degree an application meets the requirements of the user. The quality is increased by involving the end-user in the analysis and design stages.

RAD has some disadvantages however. Scalability is reduced because RAD developed applications starts as a prototype which is then developed into a finished application. Reduced features is also a disadvantage with RAD, this will occur due to features are pushed to later versions to be able to finish a new release in a shortest possible time amount.

8.2.1. RAD vs. Software Factories

Software Factory is actually more or less essentially a domain specific RAD environment. A Software Factory uses used Computer-Aided Software Engineering (CASE) tools just like RAD. The primary difference lays in the way that software information is captured. RAD uses only logical information about the software that is captured, while a Software Factory also uses conceptual information that is captured by the DSL’s.

8.3. Unified Software Development Process (UP)

The Unified Software Development Process (UP) [JBR99] makes extensive use of models with a practical focus on the modeling using the Unified Modeling Language (UML). Unified Software Development Process uses a range of different techniques for developing software. UP is a component based process that makes extent use of use-

cases, making the result to be use-case driven, architecture-centric, interactive and incremental. It uses UML as the modeling language, where UP constructs use-cases, actors, subsystems, classes, active classes, interfaces, processes, threads and modes with the help of UML. Relations are also described in the context of models.

8.3.1. UP vs. Software Factories

UP provides a prescriptive process, while Software Factories provide a more non-prescriptive framework. According to [GSCK04] UP is known as a fixed process it is actually a process model that is accompanied by a fixed process based on that model. It could therefore in theory be used to define domain specific product development processes. It would, however, need some rewriting in terms of constraints, micro-processes and focus on a specific domain.

8.4. Agile Modeling

Agile Modeling (AM) [AJ03] is based on modeling software that can be applied on software development projects. According to [AGMO] Agile Modeling consist of a collection of **values**, **principles** and **practices** for modeling software in an effective and quality way of developing software.

Values for AM:

- **Communication**. Communications between not just the team that are developing the application is important, but also contiguous communication between the developers and the project stakeholder.

- **Simplicity**. Keeping an overview of the application that is being built can be hard when getting thousands of lines with code. Using models or diagrams can easily describe an abstract level of the code.

- **Feedback**. The problem with software engineering for someone is that you might think you know what the stakeholder wants, which often can be wrong. Communication and feedback from the stakeholder is therefore very important. Enabling the stakeholder to give feedback quickly can be enabled with using diagrams or models to demonstrate your ideas.

- **Courage**. Courage to make the right decisions if the previous decisions prove to be wrong. Have the courage to either discard or at least re-factor the code to suit the requirements.

- **Humility**. Many developers think they know everything about developing software, but it is often the best developers that are modest and try to see if there exist better solutions than the one that they are thinking on, that are the best developers.

Agile modeling defines a collection for both core and supplementary principles that can be applied for software development.

Some core principles:

- Model with a purpose. Many do not see the fully purpose of modeling the software system or application they are developing, and it is often a boring work for the developers. Taking a step back and reflect on why you are modeling in the first place can be a good idea. Getting an abstract view does not just make it easier for you to understand, it also makes it much easier for others that are going to use/maintain/evolve it later on.

- Maximize stakeholders' investment. Stakeholders have chosen your company, maximizing the investment is not just important to him or her but also your own company. Are the stakeholder not satisfied with he's investment with you he will probably take he's investment somewhere else next time.

- Rapid feedback. Time between action and feedback is crucial. When working with models you can get near instant feedback on your ideas. Bringing the customer to participate get their needs and analyze is important to speed up the developing process.

- Embrace change. Project can often change over time; new ideas can develop when the project is moving ahead. This can mean that the project environment can change over time, and consequently an agile approach is needed to handle these changes.

Some Supplementary principles:

- Content is more important than representation. You take the advantage of the benefits of modeling without bringing upon yourself with expenses of creating and maintaining documentation.

- Open and honest communication. Developing in a team there need to be good communication between the participants, but there also needs to be opened up for critical comments or ideas for improvement. Having an open and good environment usually gives the developers more motivation to solve the problems then if they just got a recipe on what they had to do.

Some Practices:

- Model with others. Modeling was originally done on paper and was done to try to understand how a system or application could be built. Trying to find a solution to these problems is therefore easiest done if the problems are discussed and maybe seen from more than just one angle.

- Use the simplest tools. As most people that have experience with modeling know that you usually just can't model the system perfectly or directly from the start. Often it can be much faster to use simpler methods when starting on the basic modeling. This could be tools like normal paper. It is usually much faster to draw some easy models on the paper than making it in a software tool. The first models could be designed on paper, and when a structure of the system starts to take shape it could be transferred to the development tool.

- Display models publicly. Putting the models out on the net or anywhere else that people might have the possibility to comment them can be a very good idea. If they are public people are often more honest and are not so afraid to criticize as they might be if they where a member of the developing team.

8.4.1. AM vs. Software Factories

Software Factories and Agile Modeling has actually much in common. They both use modeling primarily as means for developing software. Agile Modeling does not, however, try to use the models programmatically as Software Factories, but every model that is created in AM is directly related to swift construction of working software. There are, however, two apparent differences when looking at both Agile Modeling and Software Factories.

Software Factories has it focus on developing application or product families, while Agile Modeling focuses on one-off development.

Agile Modeling uses models purely as a form of documentation, while Software Factories uses them as source artifacts that can be transformed into text or code.

8.5. Code Templates

Code Templates are basically pre-designed code that is either a working application or just pieces of the application that needs to be edited before the application can be compiled.

Code templates are used to a "jump start" on the application. Application or product families are normally built in the same way and have its basic building blocks that are equal on every application within the family. Essentially you can say that code templates is a pre-made and very simple application which only has the basic needs, which are used over and over again for every new application that are made on the template.

Code templates have the downside that it is very static, meaning that they are the same every time that they are used. Normally they cannot be mixed together either, which might help a lot when developing larger systems. The models in Software Factories can almost be seen in many cases as code templates that can be changed and put together accordingly to the requirements.

Wizard for generating code has been developed that lets the user make some choices and based on these choices there are generated code. It might bee an improvement to code templates, but it takes time to go through these wizards and when the wizard is done we cannot go back and re-do any of the choices that we have made, thus we might lose the overview of the application.

8.6. MetaEdit+

MetaEdit+ [ME+] is a tool for developing domain specific models. It offers full Computer-Aided Software Engineering (See 8.2 Rapid Application Development (RAD), and definition on CASE) support for your methods. It allows you to switch between views, brows designs with filters, apply components, link models to other designs, and check models with various predefined reports. The end result can be published on the web or word processors and generated to code. There are some predefined code generators for Smalltalk, C++, Java, Delphi, SQL and CORBA IDL.

9. Evaluation of results

This chapter will evaluate the results from this thesis. This will include both theoretical and my experience on the different technologies.

Model-driven development is being used ever more, and there are quite a few ways of using and defining models. Model Driven Architecture and the Domain Specific Language approach have basically the same ideas when it comes to modeling and transforming code. There are some major conceptual differences though, whereas MDA in its language has a more general way of describing software than DSL. DSLs try to see the world as it is, with all the different domains, possibilities and restrictions that are out there. DSL will thus divide the world into different smaller pieces which can explain each domain better than a general tool that tries to cover all domains at the same time. Trying to perform a surgery with a butcher knife might be possible, but I don't think anyone would want to be operated without knowing that the tools that are used are designed just for the purpose of being used for surgery. The same can be applied to software development, using tools and languages that aren't perfectly adapted to what ever is going to be developed, might not get as good result as they might have done if they used specified tools for the task.

The argument against this is whether we need so specialized tools when developing software or not. There are many arguments for and against creating tools and languages that are too specialized. As a general idea we can say that the more complex a development tool or language is, the more possibilities it has. The problem is that complex tools and languages are often very hard to understand and use.

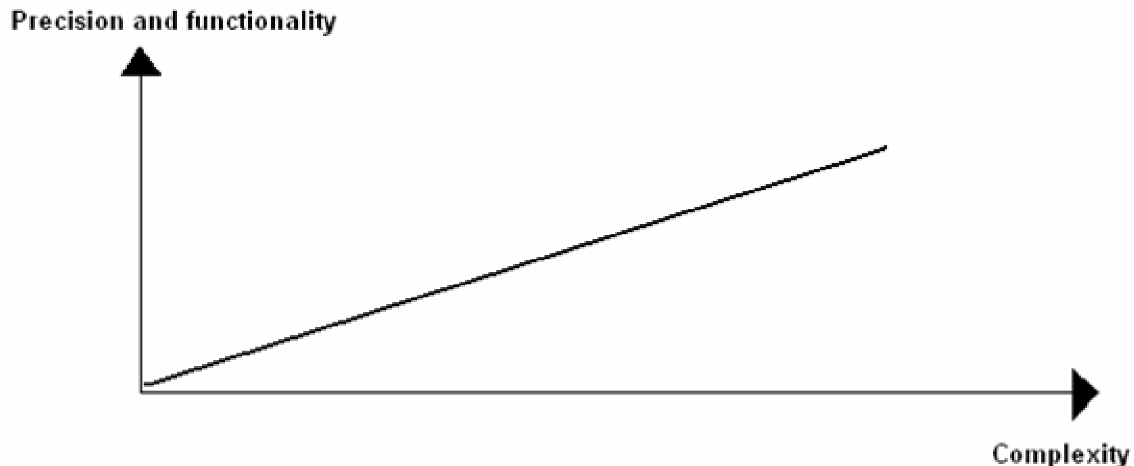


Figure 38: Complexity against precision and functionality graph

9.1. Domain Specific Languages

Domain Specific Languages are really useful when getting in to complex domains, but they also have a great advantage since they can be designed fairly easy and that the designer can decide how specific or detailed he wants the DSL to be. Since DSLs are designed around a domain and its limitations and advantages it is important to use people that knows as much as possible on the domain to design the DSL. Letting experts take care of the design will often also result in an easier-to-understand DSL as well.

When I started on this thesis I chose Microsoft's DSL Designer for developing DSLs. This was a great risk to take since it was early in the beta testing back then. There have been several releases in the tools I have used (Visual Studio and DSL designer). When I first started out back in August/September both Visual Studio 2005 and DSL Designer was still beta. Here is a list of the versions that I've used.

Visual Studio 2005 Beta 2
DSL Designer May 2005 CTP
DSL Designer September 2005 CTP

Visual Studio 2005
DSL Designer November 2005 CTP

The last release, DSL Designer November CTP, actually came out just two weeks before this thesis was delivered. So I have not had a lot of time to explore the new things that have been added to this version.

As we have seen in this thesis the DSLs have great potential, both in the way that models are designed and the way they are translated into code. As I see it there is also a lot of improvements that can be made, and I think it will continue to evolve a great deal. There are some limitations today that probably will be solved in the near future; it is however very complex to develop these tools and languages, and I've seen that there are a lot of small bugs that needs to be fixed as well.

9.2. Software factories

Software Factories is a great concept, and implementing would give software developers a great advantage when developing software. Although it is a great concept it is very complex to actually develop something that could be called a software factory. Microsoft has, with the DSL Designer for Visual Studio 2005, started to go in the direction of Software Factories. If I got the question if we where there yet; I would have to say no. Microsoft made a great leap, however, with the introduction of Domain Specific Languages in their Visual Studio 2005.

Software Factories is a collection of many different tools, items and languages. These are brought together so that building an application can almost bee seen as picking parts of the shelf and assembly them together. It might seem like it's too good to be true, but as

we develop more and more applications we have seen that we tend to build many of the applications in the same way. A car producer often has quite a few different models that they produce, but all the different models are basically build up with the same parts except for a few which makes the model unique. The same things are happening to the software development. We tend to build more and more similar applications, but we build them from scratch every time, because we haven't got a toolbox or parts that can be picked from the shelf. We can basically say that we effectively are inventing the wheel every time we build a new application, which of course isn't very effective at all.

Similar applications or applications that are inside a domain are often called an application or software *families*. It is these *families* that we can see the real potential of Software Factories. Software Factories is based on that applications are built in similar ways or at least uses similar parts. Having a way of just picking out the parts we need from a toolbox would really boost the software development speed.

Will Software Factories be the revolution that people is hoping that it will be; that remains to see. It will, however, have a great potential when it comes to applications that are similar. If applications will tend to be similar built in the future, is most certain. The days when everyone developed their own "standards" has passed (or at least gotten much better).

9.3. Domain model design

Designing DSL models can be a bit difficult when first starting out. You have only got a class model, and some relationships that can be used to connect the classes together. Then you got this massive XML file where you also need to add information about the model behavior. As I explained earlier, to be able to have tools and languages that gives us many opportunities and good functionality are often complex. The DSL Designer might seem very complex but you will soon find out that it really is very logically built when you first get the hang of it.

The main idea when designing domain specific models is to get models that can easily be used to model a application or any software, but it also has to easy for the transformation scripts to get the hold of the models and their information.

There have been many changes since the first version of the DSL Designer in Visual Studio 2005. When the compartment shape (see 5.2.3.2 Compartment models) came it was a big breakthrough. It was easy to store much information in just one model. Changes that are expected to come are shapes that are inside shapes, meaning you can have two different abstraction levels or more. This would really make the tool more usable in many situations. A whole software system can e.g. be reduced to one single model, and if we want to see what the system contains or how it is built, we can zoom down in detail.

9.4. Using DSL tools

When the models are designed, we can use them in just the same way as we would model with UML. These models can, however, be transformed into code by the use of transformation scripts. When you have completed the modeling you can select to realize it with running the custom tool. The transformation scripts that have been written will then translate the models for you.

If you have designed the DSL yourself you would need to write the transformation scripts as well to be able to get any transformations. When writing these scripts you stand freely to translate the models in any way you want. And the concept of DSLs is that these two things combined (design of the models and transformation scripts) give us a very good and usable development tool.

9.5. Software factories and DSL tools vs. other Software development languages and tools

Comparing Software Factories and DSL tools against other similar development tools and languages is not as easy as it might seem. There are a lot of tools out there which are all specialized in one way or another.

The DSL has the advantage that it can be used in basically any domain that we might find, making it both an “all-round” tool and a tool that can describe very specific details. MDA uses UML as the basis for the modeling, and thus gets a more general way of describing the world. Since different DSL can describe the whole world in great detail, it would not be wrong to say that DSLs are better in that way. MDA allows you to design models that can be used on different platforms. Platform independent models (PIM) let you design the system on a platform independent point of view, while platform specific models (PSM) let you design directly on to a platform. Models can easily be transformed to many different platforms; while Microsoft’s DSL Designer only lets you create models that run on their platform, .Net. Both tools have their advantages and disadvantages, and to try to decide which one of these two tools is the better one would almost be impossible at this time. Both tools are though still under development, and there might be surprises yet to come which might favorite one of them.

9.6. Designing Web Services with DSL

How good is the domain specific language approach for developing Web Services? DSL are very good for describing Web Services, but when it comes to implementing them in a Visual Studio project we face some problems. A normal Web Service project in Visual Studio does a lot of configuration on both the web server and the way that the project is built. These configurations need to be done manually when we are creating a Web Service from scratch.

These configurations are not so easy to do without having knowledge to both the web server and the way that Web Services are built. Hopefully there will be some adjustments on these cases in future releases of the DSL Designer.

9.7. Questions and answers

These questions where asked in chapter 1.1 and are answered and summarized here.

Model-driven Development
What is Model-driven Development (MDD) and what makes MDD so attractive for developers? Model-driven development is conceptual way of developing software with the use of models. MDD is attractive to developers because it is easy to use and it gives a more abstract view of the application then with code. See chapter 2.1 for more details
How can we model software? Software is often built in a modular way with classes and methods. These "modules" can easily be described with the use of models. These models can again have connections like when we e.g. call a method. When we have a way of modeling the software should behave, we need to transform the models into real code. This can be done by using transformation scripts See chapter 2.1 – 2.1.4 for more details
Why is it important with an abstracting view when developing software? Large applications can be very complex, and difficult to understand how they really work. When we raise the abstraction level we remove some of the details that aren't important in the big picture. This could be private variables, or content of methods based on the level of abstraction we want. See chapter 2.2.3 for more details

Software Factories
What is the concept of Software Factories? The concept of Software Factories is that it should be possible to produce software in the same way as any other industrial products. "Picking parts directly from the shelf and assembly them with help of tools and languages". A Software Factory is a collection of tools, languages, patterns, templates and many other things that helps producing software. See chapter 2.2.1 for more details
How do we produce software with Software Factories? Software is produced with the help of many different tools and technologies. Application families are developed which has the same type of building blocks. Making a new application within the family will then be easy, only the unique aspects of the application needs developed. See chapter 2.2.2 – 2.2.2.4 for more details
Can Software Factories be realized? Software Factories will probably be realized in the future. It is a very complex concept,

but also a very defuse one. DSL is a big leap towards Software Factories, and I think we will see a steady improvement of these types of tools, which sooner or later can be called Software Factories.

Domain Specific Languages
What are domains? Domains can be seen as the level of abstraction we use when we look at a certain thing. A Web Service we can say is a part of the domain of Web Service, but we can also say that it is a part of domain of software. See chapter 2.3.1 for more details
How do we develop software with DSLs? Software is developed with first defining the DSL (models), then model the software with the DSL models and connections, and then you have to write the transformation scripts that transform the models into code. See chapter 2.3 and chapter 5 for more details
How should DSLs be described? DSL are described in such a way that they are easy to use, and more importantly easy to understand. You are basically inventing a modeling language, and thus you must make it as close to the reality as possible. See chapter 2.3.2 for more details
What are the differences between DSL and Model Driven Architecture (MDA) from Object Management Group (OMG)? DSL and MDA are very similar in the way that they both use models for describing applications, whereas these models are generated into code. There are some conceptual differences though, whereas MDA tries to describe platform independent models, while DSL (in Visual Studio 2005) only uses the .Net platform. Microsoft has its own DSL editor which is perfectly adapted to the DSL design. As this is written MDA has not got any good editors for their designed models. There are many other differences as well and many of them are discussed on chapter 8.1. See chapter 8.1.2 for more details
How can Web Services be developed with the help of DSL? Web Services can easily be designed in the DSL Designer, and in chapter 6 I have in closes details described how Web Services can be developed with DSL. The transformations scripts are an important part here. See chapter 6 for more details
Is DSLs the way to go when developing Web Services? It is certainly easy to create Web Services once the DSL (models) are defined, but deploying them is another matter. Since the project type is a normal project and not a web application we need to manually configure both how the project is built and the web server needs to be configured to be a web application. There will hopefully be some better ways of solving these issues in the future.
What is the future of DSL in Visual Studio 2005? The future of DSLs in Visual Studio is very exciting; the tool has only been available in

this last version of Visual Studio. The first full version was just released early in December 2005. There are many exciting new ideas on the drawing board as this is written. The future looks bright for DSL, but even if the ideas are good it remains to see if people will use the tool or not.

9.8. Proposals for improvement

The DSL Designer in Visual Studio 2005 has not been out for long, and there are plenty of suggestions on new features that can be added to the tool. I will discuss some of the suggested new features below.

9.8.1. Two-way synchronization

Two-way synchronization between the model and the code has been proposed as a feature. This means that if we change something in the code, it will also automatically be changed in the model. The idea is very good, but getting this to work is another matter. Models have to be very specific to let changes in the code affect the model. I do not think we will see two-way synchronization for some time yet, but it might be a possibility in the future.

9.8.2. Nested shapes

Nested shapes are a possibility that some people have suggested. It sounds great to be able to create recursive modeling. How this should be done is another matter though. Recursive modeling is not very easy to do, and it is not so easy to understand if we have models that have a circular connection between each other.

9.8.3. Custom code editor

There has not been made any good editor for the custom code. When writing the custom code it is just like the old days when coding in notepad, no visual effects at all. Custom code is also very difficult to read since we are basically coding two files at once (the custom file and the file that the template should generate).

10. Conclusions and future work

This thesis has basically been about Software Factories and Model Driven Development with the main focus on Domain Specific Languages in Visual Studio 2005 when developing Web Services.

The concept of Software Factories is to have a collection of languages, tools and other items which makes the development of software like picking parts of the shelf and assembly them. Software Factories is a great concept, but has not been realized because it is a very complex concept with many complicated combinations of strategies. The use of DSLs is a start of realizing the concept of Software Factories, but there is still quite a long way to go before reaching the goal. The future of Software Factories and Domain Model Languages is very interesting, and Microsoft has nearly scratched the surface of what the tools can achieve. The Domain Model Designer in Visual Studio 2005 has still a great deal of limitations, but since the tool is relative new we haven't seen the fully potential of it yet.

The DSL Designer in Visual Studio 2005 has a lot of possibilities and opportunities that we haven't seen before in any tools or languages. The way that models can be used to design software can really revolutionize the way that we build software today. Software families are getting more and more normal as we seem to seek the best ways of developing our software. Standards is an important way of securing compatibility between systems, and with the use of software families that are built with the same kind of parts will ensure more compatibility between the applications in the application families.

Microsoft DSL Designer and MDA are very similar, at least in a conceptual way, when it comes to developing software. They both use models to describe the application or software that is being built. MDA and DSL also go through almost the same phase in the development process. They both start with defining the models that is going to be used in the development, then the models are used to design some software, before the models are translated into code that can be compiled. There are some differences though, MDA tries to have multiplatform support by first modeling in a platform independent way. They are only going into platform specific details when deploying the software to a specific platform. Microsoft DSL has not these platform stages.

MDA is defined by OMG, but OMG does not build the tools for using MDA, this is left to other producers. Microsoft offers tools for both the design of the DSL, but also an environment where the designed DSL can be used and deployed. This is a great advantage for the users, they have the same kind of environment from start to end of the development process.

Designing the Web Services and the other parts needed for the case in chapter 7 was fairly easy after I had used the DSL Designer in other projects. The DSL that I created was a relative simple one that didn't have too many built-in options. The case wasn't designed for that purpose either. It was designed for the purpose to show that once the

DSL had been described and the transformation scripts had been written a working Web Service with all its web methods could be created in just a few seconds with the help of the DSL.

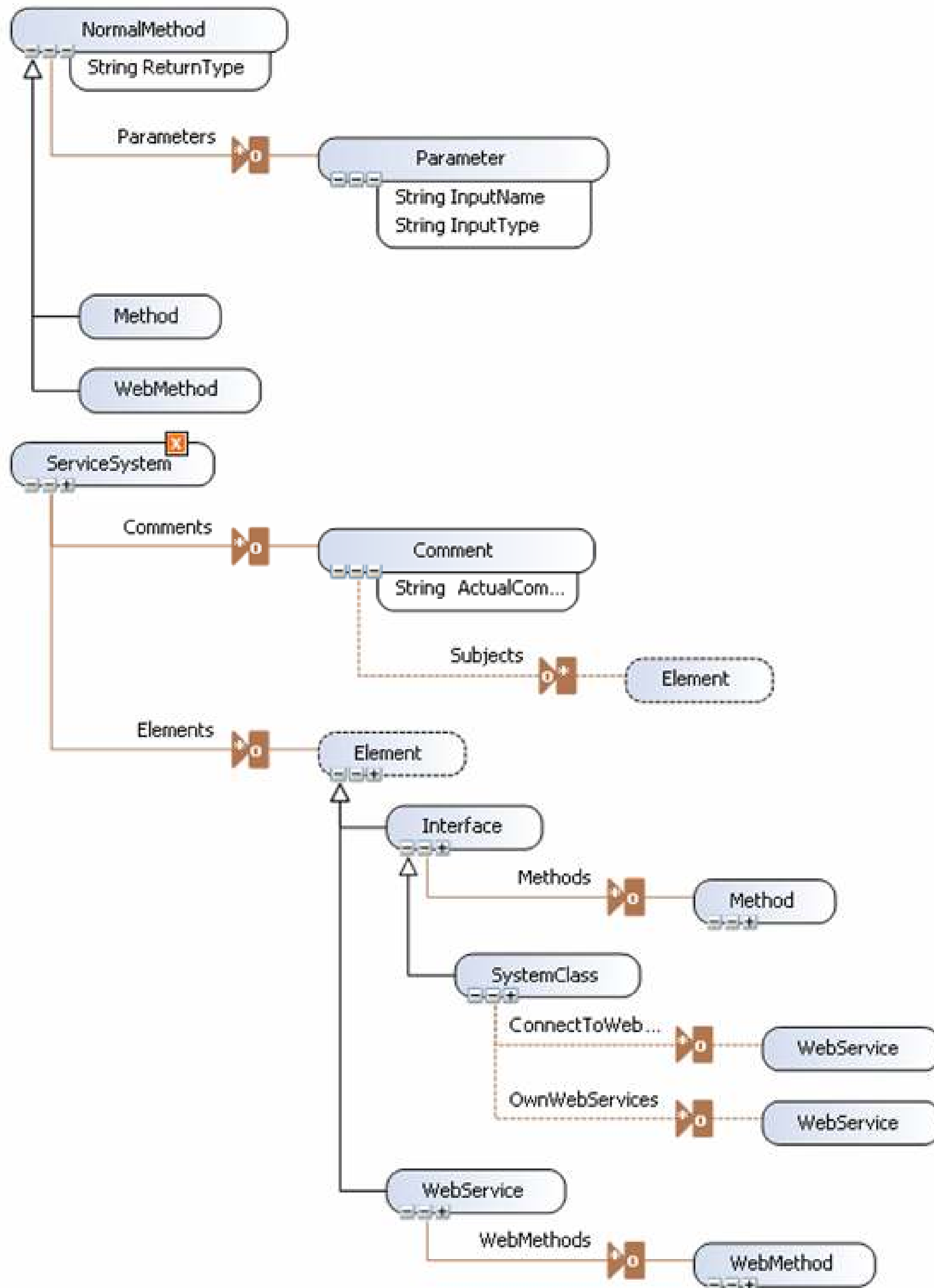
When I was finished with designing the DSL, writing the transformation script and was about to test my first Web Service that I had created with the DSL I found out that it could not be directly used. It built like it should, but I did not have the right configurations on neither the project output location nor the web server. When creating a normal Web Service project with Visual Studio 2005 all configurations are done for you. When we make a Web Service from scratch we need to configure everything manually. I tried to figure out if there was any way to start a normal Web Service project with the DSL configuration, but it did not seem to be supported. It will hopefully be in future releases of the DSL Designer. The configurations that I needed to do was to let the project be a *class library* and the output code had to be placed in the /bin/ folder and not the /bin/debug/ folder as it normally is. These configurations could be "locked" to the project when it exported and installed on other PCs. Configurations on the web server (IIS) is, however, needed for every new application that is made with the DSL. Adding the location of the Web Service is needed, and it is also needed to make the folder that the Web Service is located in to be a web application. Making sure we use the right web application version (ASP.Net) is also vital to get the Web Service up and running.

Although there are some configurations that has to be made when developing Web Services with DSL it is still many times faster than making a Web Service the old fashion way. As I said many times already, the DSL Designer and tools in Visual Studio 2005 is brand new, and there are many ideas that haven't been added yet that will come in later versions. I can only say that the future looks very bright and exciting for the DSL Designer and tools in Visual Studio 2005.

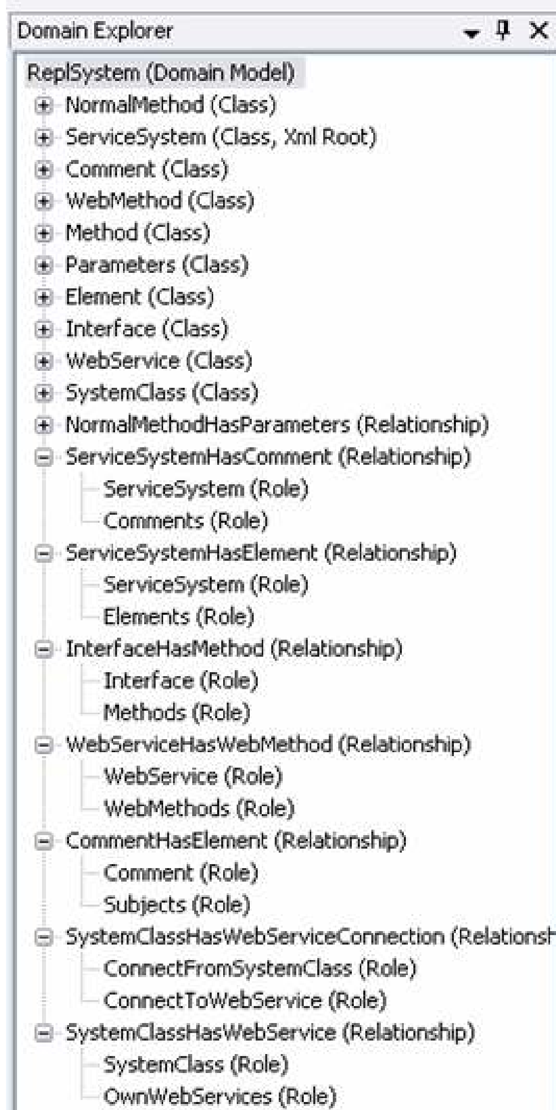
A. Appendix

A.1. Domain Specific Language Designer

A.1.1. The Domain Specific Language Design



A.1.2. Domain Explorer



A.1.3. Domain Model Designer XML code (designer.dsldd)

```
<?xml version="1.0"?>
<designerDefinition namespace="Erik.WebServiceInteraction.Designer"
name="WebServiceInteraction" fileExtension="wsi"
companyName="CompanyName" productName="WebServiceInteractionDesigner"
packageName="WebServiceInteractionDesigner" productVersion="1.0.0.0"
xmlns="http://schemas.microsoft.com/dsltools/dsldd">
    <explorer>
        <defaultRenderings>
        </defaultRenderings>
        <roots>
            <elementNode useDefaultRendering="true">
                <childCollections/>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Ser
viceSystem</class>
            </elementNode>
        </roots>
    </explorer>
    <notation>
        <diagramMaps>
            <diagramMap>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Ser
viceSystem</class>
                <connectorMaps>


                  <connectorMap>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Com
mentHasElement</class>

<connector>Erik.WebServiceInteraction.Designer.WebServiceInteractionDia
gram/Connectors/CommentConnector</connector>
                    <sourceMap>
                      <modelNavigationExpression>
                        <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Elem
ent/Comment</role>
                        </roleExpression>
                      </modelNavigationExpression>
                    </sourceMap>
                    <targetMap>
                      <modelNavigationExpression>
                        <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Comm
ent/Subjects</role>
                        </roleExpression>
                      </modelNavigationExpression>
                    </targetMap>
                  </connectorMap>
```

```
                  <connectorMap>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Sys
temClassHasWebService</class>

<connector>Erik.WebServiceInteraction.Designer.WebServiceInteractionDia
gram/Connectors/OwnsWebServiceConnector</connector>
                    <sourceMap>
                      <modelNavigationExpression>
                        <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/WebS
ervice/SystemClass</role>
                        </roleExpression>
                      </modelNavigationExpression>
                    </sourceMap>
                    <targetMap>
                      <modelNavigationExpression>
                        <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Syst
emClass/OwnWebServices</role>
                        </roleExpression>
                      </modelNavigationExpression>
                    </targetMap>
                  </connectorMap>

                  <connectorMap>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Sys
temClassHasWebServiceConnection</class>

<connector>Erik.WebServiceInteraction.Designer.WebServiceInteractionDia
gram/Connectors/ConnectToWebServiceConnector</connector>
                    <sourceMap>
                      <modelNavigationExpression>
                        <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/WebS
ervice/ConnectFromSystemClass</role>
                        </roleExpression>
                      </modelNavigationExpression>
                    </sourceMap>
                    <targetMap>
                      <modelNavigationExpression>
                        <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Syst
emClass/ConnectToWebService</role>
                        </roleExpression>
                      </modelNavigationExpression>
                    </targetMap>
                  </connectorMap>

                </connectorMaps>
```

```
      <diagram>Erik.WebServiceInteraction.Designer.WebServiceInteractio
nDiagram</diagram>
                        <shapeMaps>

          <shapeMap>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Web
Service</class>
            <compartmentMaps>
              <compartmentMap>

<compartment>Erik.WebServiceInteraction.Designer.WebServiceInteractionD
iagram/Shapes/WebServiceShape/Compartments/WebMethod</compartment>
                <melCollectionExpression>
                  <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/WebS
ervice/WebMethods</role>
                  </roleExpression>
                </melCollectionExpression>
                <valueExpression>
                  <valuePropertyExpression>

<valueProperty>Erik.WebServiceInteraction.DomainModel.WebServiceInterac
tion/WebMethod/Name</valueProperty>
                  </valuePropertyExpression>
                </valueExpression>
              </compartmentMap>
            </compartmentMaps>
            <iconMaps>
            </iconMaps>
            <melCollectionExpression>
              <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Serv
iceSystem/Elements</role>
              </roleExpression>
            </melCollectionExpression>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/WebServiceShape</shape>
            <textMaps>
              <shapeTextMap>

<textDecorator>Erik.WebServiceInteraction.Designer.WebServiceInteractio
nDiagram/Shapes/WebServiceShape/Decorators/Name</textDecorator>
                <valueExpression>
                  <valuePropertyExpression>

<valueProperty>Erik.WebServiceInteraction.DomainModel.WebServiceInterac
tion/WebService/Name</valueProperty>
                  </valuePropertyExpression>
                </valueExpression>
              </shapeTextMap>
            </textMaps>
          </shapeMap>
```

```
          <shapeMap>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Int
erface</class>
            <compartmentMaps>
              <compartmentMap>

<compartment>Erik.WebServiceInteraction.Designer.WebServiceInteractionD
iagram/Shapes/InterfaceShape/Compartments/Method</compartment>
                <melCollectionExpression>
                  <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Inte
rface/Methods</role>
                  </roleExpression>
                </melCollectionExpression>
                <valueExpression>
                  <valuePropertyExpression>

<valueProperty>Erik.WebServiceInteraction.DomainModel.WebServiceInterac
tion/Method/Name</valueProperty>
                  </valuePropertyExpression>
                </valueExpression>
              </compartmentMap>
            </compartmentMaps>
            <iconMaps>
            </iconMaps>
            <melCollectionExpression>
              <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Serv
iceSystem/Elements</role>
              </roleExpression>
            </melCollectionExpression>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/InterfaceShape</shape>
            <textMaps>
              <shapeTextMap>

<textDecorator>Erik.WebServiceInteraction.Designer.WebServiceInteractio
nDiagram/Shapes/InterfaceShape/Decorators/Name</textDecorator>
                <valueExpression>
                  <valuePropertyExpression>

<valueProperty>Erik.WebServiceInteraction.DomainModel.WebServiceInterac
tion/Interface/Name</valueProperty>
                  </valuePropertyExpression>
                </valueExpression>
              </shapeTextMap>
            </textMaps>
          </shapeMap>


          <shapeMap>
```

```
<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Sys
temClass</class>
            <compartmentMaps>
              <compartmentMap>

<compartment>Erik.WebServiceInteraction.Designer.WebServiceInteractionD
iagram/Shapes/SystemClassShape/Compartments/Method</compartment>
                <melCollectionExpression>
                  <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Syst
emClass/Methods</role>
                  </roleExpression>
                </melCollectionExpression>
                <valueExpression>
                  <valuePropertyExpression>

<valueProperty>Erik.WebServiceInteraction.DomainModel.WebServiceInterac
tion/Method/Name</valueProperty>
                  </valuePropertyExpression>
                </valueExpression>
              </compartmentMap>
            </compartmentMaps>
            <iconMaps>
            </iconMaps>
            <melCollectionExpression>
              <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Serv
iceSystem/Elements</role>
              </roleExpression>
            </melCollectionExpression>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/SystemClassShape</shape>
            <textMaps>
              <shapeTextMap>

<textDecorator>Erik.WebServiceInteraction.Designer.WebServiceInteractio
nDiagram/Shapes/SystemClassShape/Decorators/Name</textDecorator>
                <valueExpression>
                  <valuePropertyExpression>

<valueProperty>Erik.WebServiceInteraction.DomainModel.WebServiceInterac
tion/SystemClass/Name</valueProperty>
                  </valuePropertyExpression>
                </valueExpression>
              </shapeTextMap>
            </textMaps>
          </shapeMap>


          <shapeMap>

<class>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Com
ment</class>
```

```
            <iconMaps>
            </iconMaps>
            <melCollectionExpression>
              <roleExpression>

<role>Erik.WebServiceInteraction.DomainModel.WebServiceInteraction/Serv
iceSystem/Comments</role>
              </roleExpression>
            </melCollectionExpression>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/CommentShape</shape>
            <textMaps>
              <shapeTextMap>

<textDecorator>Erik.WebServiceInteraction.Designer.WebServiceInteractio
nDiagram/Shapes/CommentShape/Decorators/Comment</textDecorator>
                <valueExpression>
                  <valuePropertyExpression>

<valueProperty>Erik.WebServiceInteraction.DomainModel.WebServiceInterac
tion/Comment/Name</valueProperty>
                  </valuePropertyExpression>
                </valueExpression>
              </shapeTextMap>
            </textMaps>
          </shapeMap>

                        </shapeMaps>
                  </diagramMap>
            </diagramMaps>
            <diagrams>
                  <diagram name="WebServiceInteractionDiagram">
                        <connectors>


          <connector name="CommentConnector">
            <color variability="User" color="black" />
            <dashStyle variability="User" dashStyle="dash"/>
            <decorators>
              <connectorText name="Label" position="TargetBottom"
defaultTextId="Comment"/>
            </decorators>
            <source>
              <permittedShapes>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/CommentShape</shape>
              </permittedShapes>
            </source>
            <target arrowStyle="None">
              <permittedShapes>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/InterfaceShape</shape>
```

```
<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/WebServiceShape</shape>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/SystemClassShape</shape>
              </permittedShapes>
            </target>
          </connector>

          <connector name="ConnectToWebServiceConnector">
            <color variability="User" color="black" />
            <dashStyle variability="User" dashStyle="dash"/>
            <decorators>
              <connectorText name="Label" position="TargetBottom"
defaultTextId="Connect To"/>
            </decorators>
            <source>
              <permittedShapes>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/SystemClassShape</shape>
              </permittedShapes>
            </source>
            <target arrowStyle="EmptyArrow">
              <permittedShapes>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/WebServiceShape</shape>
              </permittedShapes>
            </target>
          </connector>

          <connector name="OwnsWebServiceConnector">
            <color variability="User" color="black" />
            <dashStyle variability="User" dashStyle="dash"/>
            <decorators>
              <connectorText name="Label" position="TargetBottom"
defaultTextId="Owns"/>
            </decorators>
            <source>
              <permittedShapes>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/SystemClassShape</shape>
              </permittedShapes>
            </source>
            <target arrowStyle="None">
              <permittedShapes>

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/WebServiceShape</shape>
              </permittedShapes>
            </target>
          </connector>
```

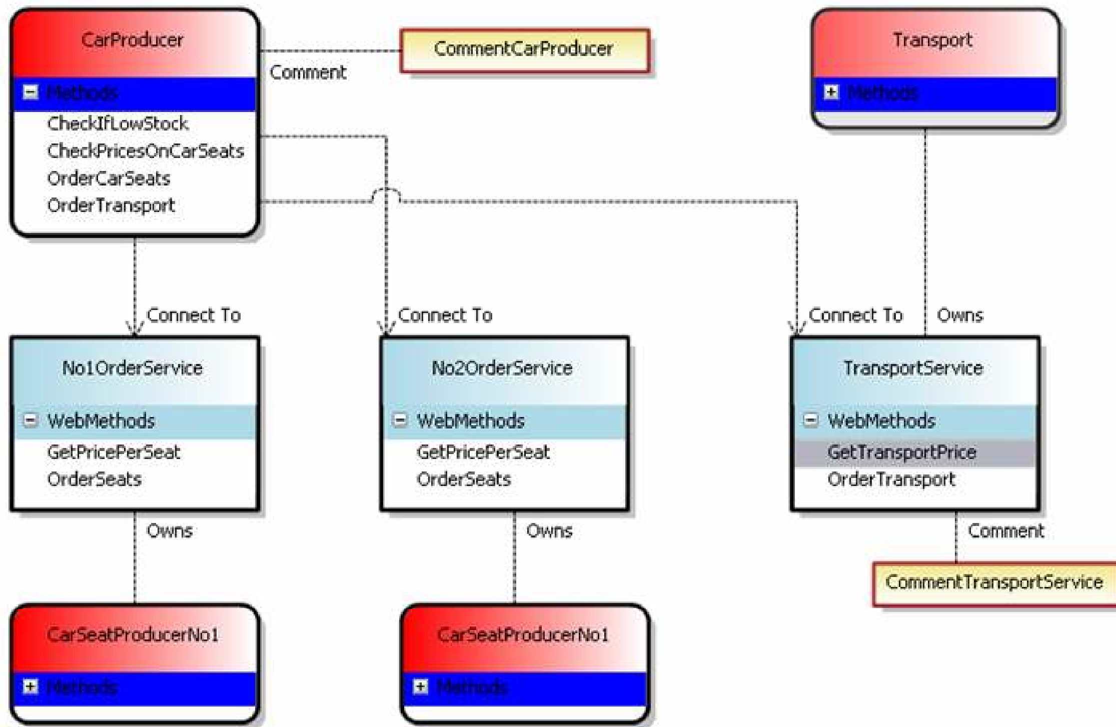
```
                        </connectors>
        <shapes>

          <compartmentShape name="WebServiceShape" initialWidth="1.5"
initialHeight="0.4" geometry="Rectangle">
            <decorators>
              <shapeText  name="Name" position="Center"
defaultTextId="WebServiceShapeNameDecorator"/>
            </decorators>
            <fillColor color="lightblue" variability="User"/>
            <outlineColor color="black" variability="User"/>
            <compartments>
              <listCompartment name="WebMethod" captionId="WebMethods">
                <compartmentFillColor color="white" variability="Fixed"
/>
                <titleFillColor color="lightblue" variability="Fixed"/>
              </listCompartment>
            </compartments>
          </compartmentShape>

          <geometryShape name="CommentShape" initialWidth="1.5"
initialHeight="0.3" geometry="Rectangle">
            <decorators>
              <shapeText name="Comment" position="Center"
defaultTextId="CommentShapeCommentDecorator"/>
            </decorators>
            <fillColor color="khaki" variability="User"/>
            <outlineColor color="brown" variability="Fixed"/>
          </geometryShape>

          <compartmentShape name="InterfaceShape" initialWidth="1.5"
initialHeight="0.4" geometry="RoundedRectangle">
            <decorators>
              <shapeText  name="Name" position="Center"
defaultTextId="InterfaceShapeNameDecorator"/>
            </decorators>
            <fillColor color="green" variability="User"/>
            <outlineColor color="black" variability="User"/>
            <compartments>
              <listCompartment name="Method" captionId="Methods">
                <compartmentFillColor color="white" variability="Fixed"
/>
                <titleFillColor color="red" variability="Fixed"/>
              </listCompartment>
            </compartments>
          </compartmentShape>

          <compartmentShape name="SystemClassShape" initialWidth="1.5"
initialHeight="0.4" geometry="RoundedRectangle">
            <decorators>
              <shapeText  name="Name" position="Center"
defaultTextId="SystemClassShapeNameDecorator"/>
            </decorators>
            <fillColor color="red" variability="User"/>
            <outlineColor color="black" variability="User"/>
            <compartments>
              <listCompartment name="Method" captionId="Methods">
```

```
                <compartmentFillColor color="white" variability="Fixed"
/>
                <titleFillColor color="blue" variability="Fixed"/>
              </listCompartment>
            </compartments>
          </compartmentShape>
        </shapes>
                        <toolbox>
                              <items>
            <shapeTool iconId="ExampleShapeToolBitmap"
captionId="WebServiceShapeToolboxCaption"
contextSensitiveHelpId="ExampleShapeHelpId" order="0">

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/WebServiceShape</shape>
            </shapeTool>
            <shapeTool iconId="ExampleShapeToolBitmap"
captionId="CommentShapeToolboxCaption"
contextSensitiveHelpId="ExampleShapeHelpId" order="1">

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/CommentShape</shape>
            </shapeTool>
            <shapeTool iconId="ExampleShapeToolBitmap"
captionId="InterfaceShapeToolboxCaption"
contextSensitiveHelpId="ExampleShapeHelpId" order="2">

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/InterfaceShape</shape>
            </shapeTool>
            <shapeTool iconId="ExampleShapeToolBitmap"
captionId="SystemClassShapeToolboxCaption"
contextSensitiveHelpId="ExampleShapeHelpId" order="3">

<shape>Erik.WebServiceInteraction.Designer.WebServiceInteractionDiagram
/Shapes/SystemClassShape</shape>
            </shapeTool>

            <connectorTool iconId="ExampleConnectorToolBitmap"
captionId="CommentConnectorToolboxCaption"
contextSensitiveHelpId="ExampleConnectorHelpId" order="4">

<connector>Erik.WebServiceInteraction.Designer.WebServiceInteractionDia
gram/Connectors/CommentConnector</connector>
            </connectorTool>
            <connectorTool iconId="ExampleConnectorToolBitmap"
captionId="ConnectToWebServiceConnectorToolboxCaption"
contextSensitiveHelpId="ExampleConnectorHelpId" order="5">

<connector>Erik.WebServiceInteraction.Designer.WebServiceInteractionDia
gram/Connectors/ConnectToWebServiceConnector</connector>
            </connectorTool>
            <connectorTool iconId="ExampleConnectorToolBitmap"
captionId="OwnsWebServiceConnectorToolboxCaption"
contextSensitiveHelpId="ExampleConnectorHelpId" order="6">
```

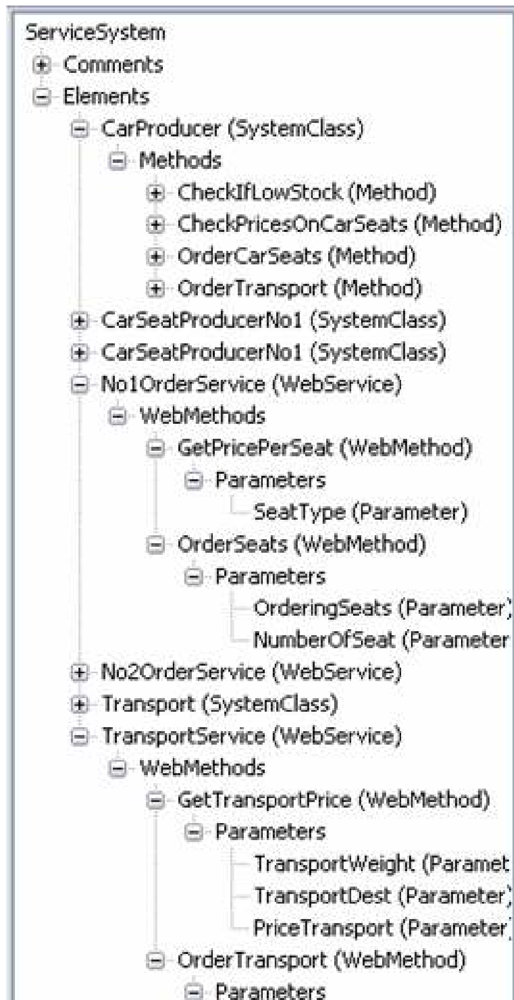
```
<connector>Erik.WebServiceInteraction.Designer.WebServiceInteractionDia
gram/Connectors/OwnsWebServiceConnector</connector>
            </connectorTool>

                              </items>
                        </toolbox>
                  </diagram>
            </diagrams>
      </notation>
      <objectModels>
            <objectModel
name="Erik.WebServiceInteraction.DomainModel.WebServiceInteraction"
fileName="..\DomainModel\DomainModel.dsldm">

      <model>Erik.WebServiceInteraction.DomainModel.WebServiceInteracti
on</model>
            </objectModel>
      </objectModels>
      <propertiesWindow>
            <propertySets>
            </propertySets>
      </propertiesWindow>
      <validation open="false" save="false" menu="false"
custom="false"/>
</designerDefinition>
```

A.2. Designing with domain models

A.2.1. Visual design



A.2.2.2. WebServiceInteractivity



A.3. Custom code

A.3.1. Custom code for asmx files

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTran
sformation"#>
<#@ output extension=".asmx" #>
<#@ servicesystem processor="WebServiceInteractionDirectiveProcessor"
requires="fileName='Sample.wsi'" provides="ServiceSystem=ServiceSystem"
#>
<#

	foreach(Element el in this.ServiceSystem.Elements)
	{
		if(el.GetType().Name == "WebService")
		{
			WebService ws = (WebService)el;
#>
<#			//using(new File(ws.Name, this))
			{
#>
<%@ WebService Language="C#" CodeBehind="codeBehindFile.cs"
Class="<#=ws.SystemClass.Name#>.<#=ws.Name#>" %>
<#
			}
		}
	}
#>
```

A.3.2. Code behind custom code

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTran
sformation"#>
<#@ output extension=".cs" #>
<#@ servicesystem processor="WebServiceInteractionDirectiveProcessor"
requires="fileName='Sample.wsi'" provides="ServiceSystem=ServiceSystem"
#>
using System;
using System.Web.Services;

<#
      foreach(Element el in this.ServiceSystem.Elements)
      {
            if(el.GetType().Name == "WebService")
            {
                  WebService ws = (WebService)el;
#>
namespace <#=ws.SystemClass.Name#>
{
<#
            foreach(Erik.WebServiceInteraction.DomainModel.Comment co
in this.ServiceSystem.Comments)
            {
                  if(co.Subjects == ws)
                  {
#>
      /*
      <#=co.ActualComment#>
      */
<#}}#>
      public class <#=ws.Name#> : WebService
      {
<#
                  foreach(WebMethod wm in ws.WebMethods)
                  {
#>
            [WebMethod]
            public <#=wm.ReturnType == "" ? "void" : wm.ReturnType#>
<#=wm.Name#>(<#
                  int counter = 0;
                  foreach(Parameter pm in wm.Parameters)
                  {
                        counter++;
                        if(counter > 1)
                        {#>, <#=pm.InputType#> <#=pm.InputName#><#}#><#
                        else{
            #><#=pm.InputType#> <#=pm.InputName#><#}}#>)
            {
                  throw new NotImplementedException("Not implemented
yet");
            }
<#                } //end foreach   #>
      }
```

```
}
<#          } //end if              #>
<#    } //end foreach               #>
```

A.3.3. Custom code for the rest of the modeling

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTran
sformation"#>
<#@ output extension=".cs" #>
<#@ servicesystem processor="WebServiceInteractionDirectiveProcessor"
requires="fileName='Sample.wsi'" provides="ServiceSystem=ServiceSystem"
#>
using System;

<#
	foreach(Element el in this.ServiceSystem.Elements)
	{
		if(el.GetType().Name == "SystemClass")
		{
			SystemClass sc = (SystemClass)el;
#>
namespace <#=sc.Name#>
{
<#
		foreach(Erik.WebServiceInteraction.DomainModel.Comment co
in this.ServiceSystem.Comments)
		{
			if(co.Subjects == sc)
			{
#>
	/*
	<#=co.ActualComment#>
	*/
<#}}#>
	public class <#=sc.Name#>
	{
		public <#=sc.Name#>()
		{

		}
<#
			foreach(Method m in sc.Methods)
			{
#>
		public void <#=m.Name#>()
		{
			throw new NotImplementedException("Not implemented
yet");
		}
<#			} //end foreach   #>
```

```
       }
}
<#          } //end if               #>
<#     } //end foreach               #>
```

A.4. Generated files

A.4.1. No1OrderService asmx file

```
<%@ WebService Language="C#" CodeBehind="codeBehindFile.cs"
Class="CarSeatProducerNo1.No1OrderService" %>
```

A.4.2. No2OrderService asmx file

```
<%@ WebService Language="C#" CodeBehind="codeBehindFile.cs"
Class="CarSeatProducerNo2.No2OrderService" %>
```

A.4.3. TransportService asmx file

```
<%@ WebService Language="C#" CodeBehind="codeBehindFile.cs"
Class="Transport.TransportService" %>
```

A.4.4. Code behind classes

```
using System;
using System.Web.Services;

namespace CarSeatProducerNo1
{
	public class No1OrderService : WebService
	{
		[WebMethod]
		public int GetPricePerSeat(string seatType)
		{
			throw new NotImplementedException("Not implemented
yet");
		}
		[WebMethod]
		public string OrderSeats(string seatType, int
numberOfSeats)
		{
			throw new NotImplementedException("Not implemented
yet");
		}
	}
}
namespace CarSeatProducerNo1
{
	public class No2OrderService : WebService
	{
		[WebMethod]
		public int GetPricePerSeat(string typeOfSeat)
		{
			throw new NotImplementedException("Not implemented
yet");
		}
		[WebMethod]
		public string OrderSeats(string seatType, int howManySeats)
		{
			throw new NotImplementedException("Not implemented
yet");
		}
	}
}
namespace Transport
{
	/*
	Web Service for getting prices on transport and ordering
transport
	*/
	public class TransportService : WebService
	{
		[WebMethod]
		public int GetTransportPrice(string source, string
destination, int weight)
		{
```

```
                    throw new NotImplementedException("Not implemented
yet");
              }
              [WebMethod]
              public string OrderTransport(int orderNo)
              {
                    throw new NotImplementedException("Not implemented
yet");
              }
       }
}
```

A.4.5. Other generated code

```
using System;

namespace CarProducer
{
       /*
       A car producer that needs seats for their car Has 2 producers of
seats that they can order from via Web Services A Transport is needed
for transporting the seats to the car producer. A Transport Web Service
is availible
       */
       public class CarProducer
       {
              public CarProducer()
              {

              }
              public void CheckIfLowStock()
              {
                    throw new NotImplementedException("Not implemented
yet");
              }
              public void CheckPricesOnCarSeats()
              {
                    throw new NotImplementedException("Not implemented
yet");
              }
              public void OrderCarSeats()
              {
                    throw new NotImplementedException("Not implemented
yet");
              }
              public void OrderTransport()
              {
                    throw new NotImplementedException("Not implemented
yet");
              }
```

```
        }
}
namespace Transport
{
        public class Transport
        {
                public Transport()
                {

                }
        }
}
namespace CarSeatProducerNo1
{
        public class CarSeatProducerNo1
        {
                public CarSeatProducerNo1()
                {

                }
        }
}
namespace CarSeatProducerNo1
{
        public class CarSeatProducerNo1
        {
                public CarSeatProducerNo1()
                {

                }
        }
}
```

B. Bibliography

[XML03] W3C: *eXtensible Markup Language (XML),* http://www.w3.org/XML 2003

[WS02] W3C: *Web Services,* http://www.w3.org/2002/ws/ 2002

[SFAP05] Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools
http://msdn.microsoft.com/vstudio/teamsystem/workshop/dsltools/default.aspx?pull=/library/en-us/dnbda/html/softfact3.asp 2005

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker: Staged Configuration Using Feature Models, 2004

[UML05] Unified Modeling Language (UML), http://www.uml.org/ 2005

[UML04] Booch, G. I. Jacobsen: *The Unified Modeling Language Reference Manual, Second Edition.* 2004

[DSLDM] DSL Dm->Dd tool: http://www.modelisoft.com/Dmd2Dd.aspx 2005

[ME+] MetaEdit+ Domain specific model designer http://www.metacase.com/

[JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh: *The Unified Software Development Process,* 1999

[GSCK04] Jack Greenfield and Keith Short with Steve Cook and Stuart Kent: *Software Factories, Assembling Applications with Patterns, Models, Frameworks and, tools*, 2004

[MDA03] Bast W. A. Kleppe: *MDA Explained. The Model Driven Architecture: Practice and Promise*. 2003

[MSVS05] Microsoft Visual Studio 2005 http://msdn.microsoft.com/vstudio

[SF05] Software Factories: *industrialized software development*
http://www.softwarefactories.com 2005

[.Net05] Microsoft .Net platform, http://www.microsoft.com/net/default.mspx 2005

[J2EE05] Java 2 Enterprise Edition, J2EE http://java.sun.com/j2ee/index.jsp 2005

[CORBA05] Common Object Request Broker Architecture, CORBA
http://www.corba.org/ 2005

[WSDL05] Chinnici, R., J. –J Moreau: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, 2005 http://www.w3.org/TR/wsdl20

[WSDL01] Cristensen, E. F. Curbera: *Web Services Description Language (WSDL) Version 1.1*, 2001 http://www.w3.org/TR/wsdl

[SOAP03] Simple Object Access Protocol v. 1.1 (SOAP 1.1), 2003 http://www.w3.org/TR/soap/

[SOAP05] Simple Object Access Protocol v. 1.2 (SOAP 1.2), 2005 http://www.w3.org/TR/soap12

[T4M05] Custom code and T4 engine syntax, https://blogs.msdn.com/garethj/archive/2005/06/01/T4Syntax.aspx,

[PAR76] D. Parnas. *On the design and development of program families*. IEEE on Transactions on software engineering. 1976

[ESF87] ESF Control Board. *Eureka Software Factory*. 1987

[OMG05] Object Management Group (OMG). http://www.omg.org/ 2005

[AGMO] Agile Modeling Home Page http://www.agilemodeling.com/

[AJ03] Scott W. Ambler, Ron Jeffries: *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process,* 2003

[DSLFOR] Microsoft Domain-Specific Language Tools discussion forum: http://forums.microsoft.com/msdn/showforum.aspx?forumid=61&siteid=1

[T3C] ColorizedT3, editor colorizer for Visual Studio 2005 custom code http://www.modelisoft.com/ColorizedT3.aspx

[WSE02] Ethan Cerami, *Web Services Essentials* .2002

[CFWS] Contract first Web Services http://msdn.microsoft.com/vstudio/java/interop/webspehereinterop/default.aspx

[SOA03] Service Oriented Architecture http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[TTWS] Think Tecture, Contract first Web Services for Visual Studio http://www.thinktecture.com/Resources/Software/WSContractFirst/default.html

[FMW01] Feature Modeling Workshop, http://www-st.inf.tu-dresden.de/gcse-fm01/ 2001

[MDDD04] Stephen J. Mellor, Kendall Scott, Axel Uhl and Dirk Weise: *MDA Distilled, Principles of Model-driven Architecture*, 2004

[XMI05] MOF 2.0 / XMI Mapping Specification, v2.1, http://www.omg.org/technology/documents/formal/xmi.htm

Video:
http://msdn.microsoft.com/msdntv/episode.aspx?xml=episodes/en/20041118SoftFact/manifest.xml