EasyWeb:
A Software Engineering Environment for Developing Web Applications in .Net

A M I R T A L A E I – K H O E I

Master of Science Thesis
Stockholm, Sweden, 2007

ICT/ECS-2007-134

ROYAL INSTITUTE
OF TECHNOLOGY

Master of Science Thesis

AMIR TALAEI-KHOEI

Supervised by Prof. Claude Petitpierre (EPFL/IC/ISIM/LTI)

Examined by: A/Prof. Vladimir Vlassov (KTH/ICT/ECS)

Stockholm, Sweden, November 2007.

Abstract

Web applications are ubiquitous on the Internet, and almost every type of business now needs to be able to quickly develop their own applications, but as web applications become more complex, developers look for more systematic ways to build quality applications with minimum effort.

This thesis proposes an environment that focuses on generating the first draft of a web application for an architectural-based system model design. Since the description is high level, it's easy to produce and it only needs the basic knowledge of web application development, additionally because the chosen architecture is very close to the way that the most of web applications are developed the generated code is still easy to change in .Net platform.

Acknowledgment

I'm extremely grateful for kind consideration of a person, who guided me during the whole project, from its very beginning to the end; Olivier Buchwalder, a PhD student at LTI. He made me interested in the topic, and helped me a lot with theoretical issues, as well as with problems that arose during the implementation.

Finally, I would be pleased to dedicate this work to my parents because of their loves.

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

Chapter 1

1 Introduction

As Web applications grow in size, the size of development projects grows as well, and it becomes more and more critical to support modular application design and parallel development. The challenge lies in meeting a few unique requirements.

The most important requirement to be met is that the solution supports the appropriate division of the development effort into smaller and different kinds of tasks that can be performed by various kinds of developers, such as architects, screen designers, and business logic programmers. It must then support the efficient integration of the many outputs from these different tasks. There must also be a cost-effective method of unit and integration testing if we are to maximize the parallel progress of the tasks that are inherently dependent upon one other.

Most of the technology is here to implement systems that exploit the web paradigm, but the effective design of Web applications is still a concern. The complexity and requirements on web applications are constantly growing, while the supporting technologies and platforms rapidly evolve.

Web applications are quite common now, though the development and specially requirement analysis phase of them are still a big challenge. Such a situation is attributable to the flood of technologies and lack of standards.

At the time, most of web application developers and companies involved in this industry need some facilities to rapid prototyping and tools for easy design web application not only in case of interface but also for developing a web applications with all links, operations in a speedy architectural design, actually considering amount of strong tools in interface deign for web applications the point of interface is doesn't matter for developers specially when they want to make a prototype and deliver it to customer as soon as possible.

In this work I have tried to present a tool for developing the first draft web application or prototype which can be used by persons who has basic knowledge about web applications and how they can be developed. In this case if we can find a person with this amount of knowledge in the customer side which is not really far from of imagination, the tool can be counted as a user development environment which is important in the recent aspects of software engineering to find the requirements as better as possible.

EasyWeb, the tool that I am going to release can help .Net developers to make their own web application just with architecture design which is possible in graphical user friendly designer or programming editor with the user assist abilities. In this work I have tried and make much more efforts on simplicity to generate web application easily and rapidly with architectural design.

1.1 Problem Statement

When you are using .Net environment to develop web applications, you have a very strong IDE to design your web pages and coding inside with another strong programming language (ASP.Net), but actually which is needed for rapid first draft development and also needed for designing web application is an environment that you can have an architectural design and then generate the application.

.Net Environment with a lot of facilities and components needs an expert programmer to use and develop applications but unfortunately we can see the opposite in reality that most of time designers know the basics but they are not expert in programming. Therefore EasyWeb has been developed for designers to design the system architecture and then they can generate executable end user application, however this one is not deliverable to market but it can show the functionality and structure.

1.2 Goals

EasyWeb as a tool for developing web application has been goaled for developing first draft web application coded in ASP.Net and not goaled for developing ready market application or interface design because as we have described in pervious section we have already strong tools and IDE such as .Net to do these kinds of stuffs.

To sum up the goals for EasyWeb tool is speeding up the following items in the web application development life cycle:

- Developing the first draft of web applications
- Clarifying the architecture
- Justification the operations

This Tool has to be made developer friendly environment for rapid development however it doesn't need to consider about the interface design.

The scenario has been designed for developing web applications using this tool for requirement engineering and especially for prototyping then using .Net IDE for release the market ready version of end application. Figure 1 shows this life cycle or scenario which is proposed for developing web applications, for the first draft, designer uses EasyWeb for simple architectural design which can generate executable code for web application, this version is also useful for clarifying the architecture.

1.3 Brief Description

As far as we said in pervious sections, I'm going to have a tool for simplification of web application development in .Net with architectural design. EasyWeb as my tool and which I will propose in this work tries to provide a graphical designer and also a programming environment with user assistance ability for developing these kinds of application.

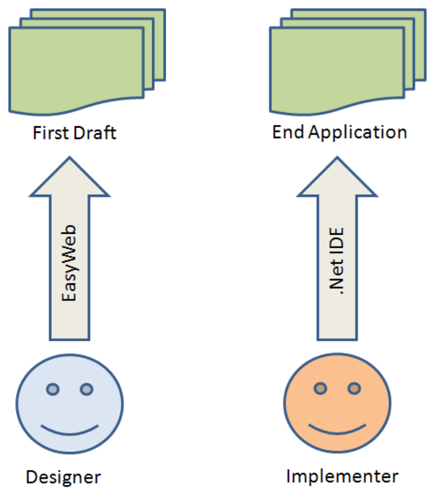


Figure 1. Roles for developing web application using EasyWeb

1.3.1 Roadmap of Work

Figure 2 tries to show the roadmap of this work for developing EasyWeb. In this work we have described a language consists of datatypes and their properties for describing a web application and using this language developer can model their systems in two different environments:

- *A Graphical Designer*: meta developer (in this case I am) using the Microsoft DSL tools defined a language which is used for modeling systems and run the Microsoft DSL Tools, then the result was a Microsoft VS Designer for developing web applications, this environment has some drag and drop facilities for describing entities of mentioned application and also it has some wizards for determining properties of these entities. In designer tool I have tried to make some facilities for simplification of properties determination.

- *A Programming Editor*: EasyWeb has also a programming completion-enabled editor for modeling systems with programming. In this case I have used a tool developed in LTI[1] to define the different language (but with the same concepts of pervious one) in graphical editor and also produced an editor. Designers can model their systems in this editor simply and then after compilation, it will be transformed to language designed in Microsoft DSL Tools.

So both graphical and programmed model produced by designer should be transformed to a machine enabled language and then Code Gen will be used for generating executable codes for end application. EasyWeb Code Gen read the machine enabled program and then generate .aspx and .cs files, which are executable and then it is also possible to change in Microsoft .Net Environment.

[1] Lab. Teleinformatique located at Department of Computer and Communication Sciences, EPFL, Switzerland and where WebLang has been also developed. You can find its web site on ltiwww.epfl.ch

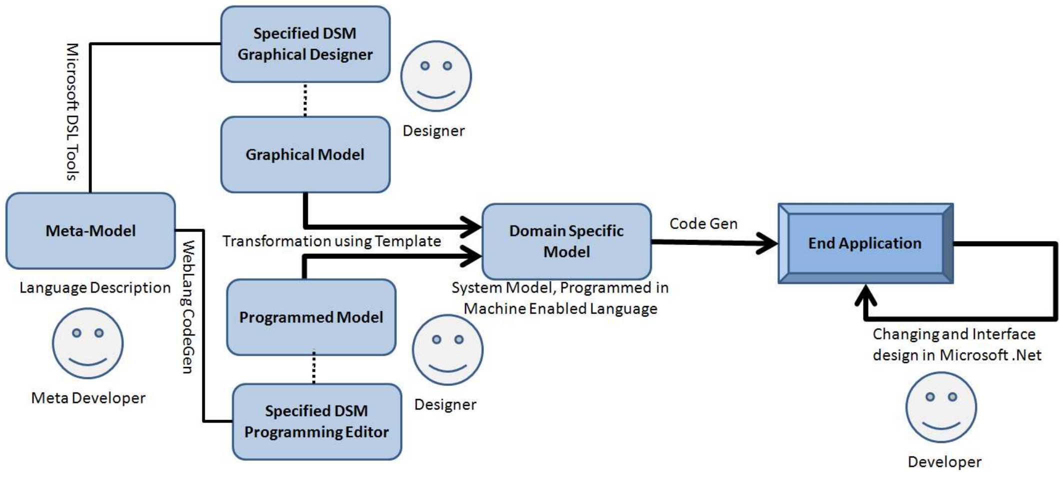

Specified DSM Graphical Designer
Designer
Graphical Model
Microsoft DSL Tools
Meta-Model
Language Description
Meta Developer
WebLang CodeGen
Transformation using Template
Domain Specific Model
Code Gen
End Application
System Model, Programmed in Machine Enabled Language
Programmed Model
Designer
Specified DSM Programming Editor
Changing and Interface design in Microsoft .Net
Developer

Figure 2. Roadmap for Developing in EasyWeb

1.3.2 Limitations

As far as I have to use Microsoft DSL Tools and WebLang CodeGen tool I will face to some limitations that these tools force:

- *Microsoft DSL Tools:* Microsoft DSL Tools are designed to develop product line and not executable end application and actually for developing end application. We also need some properties which should be advised and manage by some wizard to make their specification more easily than just entering some properties. But architecture of these tools however allows defining properties but it's hard to generate some wizard for getting data.

- *WebLang CodeGen*: WebLang doesn't allow us to generate graphical designer so to generate this kind of designer for who think it's easier, we should use Microsoft DSL Tools however I believe programming environments are much easier and more efficient.

1.4 Thesis Outlines

The rest of the report has been organized as follows:

- **Chapter 2, Background:** It gives the reader brief explanation of Model Driven Development and its approaches, and then it explains the much more details in Software Factory as a used concept in EasyWeb. It also has a introduction on Microsoft DSL Tools and WebLang as tools that I have used in this work.

- **Chapter 3, Methods:** This chapter presents the usecases and design of EasyWeb, it also covers the architecture of this tool.

- **Chapter 4, Implementation:** All Design methods need implementation to create the tool, this chapter is specified for explaining my implementation and how I have actually created EasyWeb.

- **Chapter 5, Analysis:** Firstly, it discusses about how our roadmap and activities to achieve a Domain Specific Language is valid, and then it covers the evaluation of EasyWeb and its compression to other tools in this area.
- **Chapter 6, Case Study:** This chapter explains a simple web application has been developed with EasyWeb. Chapter can also be used as a tutorial of EasyWeb.
- **Chapter 7, Conclusion:** This chapter sums up the all I have achieved in this project and also it clarifies directions have been opened in my works.
- **Some Appendices:** During the report we have pointed to some information that it can be read only as a reference so I have put them at the end of the report.

Please consider that references are presented in Chicago Style. Therefore the name of author and year inside of the parentheses mean a reference which can be found in biography part of report.

Chapter 2

2 Background

This chapter covers the theory of Model Driven Development (MDD) that readers need to know for understanding my work. In following sections, we will have an overview on Model Driven Development and its strategies; Model Driven Architecture and Software Factory, then we will focus on Domain Specific Language, its Microsoft DSL Tools and we will also explain WebLang.

2.1 Model Driven Development (MDD)

In recent years many organizations have begun to focus attention on Model Driven Development (MDD) as an approach to application design and implementation. This is a very positive development for several reasons. This approach encourages efficient use of system models in the software development process, and it supports reuse of best practices when creating families of systems.

As Pham said, "Since the introduction of high-level programming languages like C, C++ and Java, software developers have seen a big breakthrough in productivity. Instead of having to construct their applications from primitives like *load*, *store* and *jump*, developers are able to express their logic using higher and more natural programming constructs like *loops*, *conditionals*, etc., and then call upon a compiler to generate (virtual) machine-level code. Also, using these languages, the task of porting software to a new execution platform when it comes along involves only a recompile, as opposed to a rewrite or a redesign, of the programs in the system. These benefits came from the basic idea that, by raising the abstraction level of the language used by developers, and by automating the process of transforming code from the raised abstraction level to the target (lower) abstraction level, productivity in both software development and maintenance can be significantly improved" (H. N. Pham 2007).

MDD is a software engineering methodology with particular focus on models, automation and code generation. The difference to traditional software development is that MDD proposes to leverage models to generate the specified software system. Two currently dominant approaches to MDD are Model-Driven Architecture (MDA) (J. Mukerji 2003) and Software Factories (SF) (J. Greenfield 2004).

The concept of model-driven development (MDD) is really starting to catch on because of its promise to increase the productivity of those charged with the task of developing and maintaining application systems. But what exactly is MDD?

MDD is a development practice where high-level, agile, and iterative software models are created and evolved as software design and implementation takes place. The key defining characteristic of MDD is that the model literally becomes part of the development process. Contrast this with an approach such as the waterfall development process where modeling appears as a separate step in the process and tends to get left behind once the development proceeds to the next phase (Schwaderer 2006).

Models are used to specify software systems, but unfortunately these models mostly serve only for the purpose of documentation and comprehending the system. Changing this fact by using these existent models to generate the application, software development can easily be automated. By automatic code generation, the quality of an application can be increased, due to the fact that code is produced according to a certain structure, scheme or rules. In this way the generated code will precisely match the models. Further on this road the evolution could lead to the fact that modeling languages replace the implementation languages, just like the way third-generation languages replaced the assembly languages through the introduction of compilers (Demir 07).

2.1.1 Modeling Rationale and MDD

Referring to oxford dictionary;" Model is a simplified mathematical description of a system or process, used to assist calculations and predictions" and Fishery Glossary says "Models help to show relationships between processes (physical, economic or social) and may be used to predict the effects of changes in land use" and it also defines Modeling as "The construction of physical, conceptual or mathematical simulations of the real world".

The concept of the model and the modeling in the IT area is focusing. In this field a model represents a part of reality (target) which was specified by a modeling view and described by modeling facilities in order to the purposes for recognition, understanding, and manipulation of the target.

A model is a form of abstraction that allows real-world entities to be represented in a simplified manner, so that they can be dealt with in safer, cheaper and easier ways (Rothenberg 1989).

Finally, by IBM definition; Models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on relevant ones. All forms of engineering rely on models to understand complex, real-world systems. Models are used in many ways: to predict system qualities, reason about specific properties when aspects of the system are changed, and communicate key system characteristics to various stakeholders. The models may be developed as a precursor to implementing the physical system, or they may be derived from an existing system or a system in development as an aid to understanding its behavior (Brown 2004).

In the software engineering world, modeling has a rich tradition of programming. The most recent innovations have focused on notations and tools that allow users to express system perspectives of value to software architects and developers in ways that are readily mapped into the programming language code that can be compiled for a particular operating system platform. The current state of this practice employs the

Unified Modeling Language (UML) as the primary modeling notation. The UML allows development teams to capture a variety of important characteristics of a system in corresponding models. Transformations among these models are primarily manual. UML modeling tools typically support requirements traceability and dependency relationships among modeling elements, with supporting documents and complementary consulting offerings providing best practice guidance on how to maintain synchronized models as part of a large-scale development effort (Brown 2004).

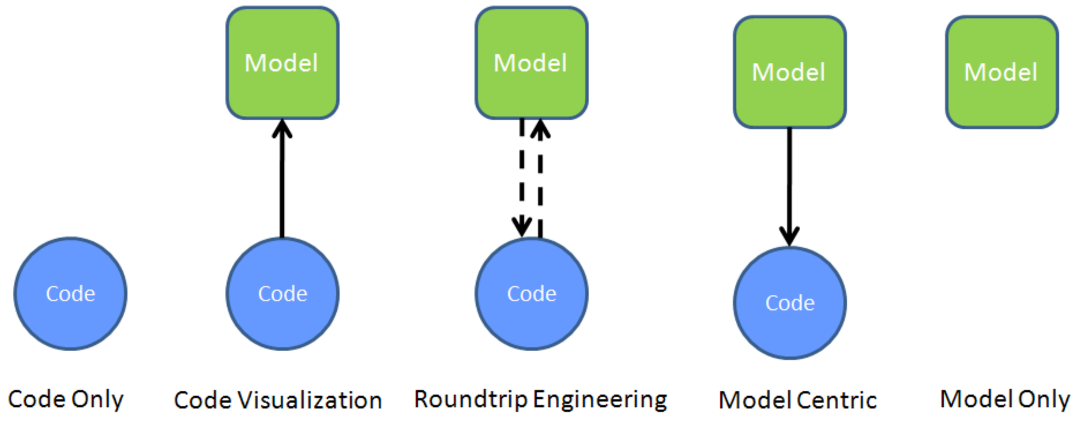


Figure 3. Different ways of synchronization of code and model (Brown 2004)

Code Only: Today, a majority of software developers still take this approach and do not use separately defined models at all. They rely almost entirely on the code they write, and they express their model of the system they are building directly in a third-generation programming language such as Java, C++, or C# within an Integrated Development Environment (IDE) such as Visual Studio .Net. Any "modeling" they do is in the form of programming abstractions embedded in the code (e.g., packages, modules, interfaces, etc.), which are managed through mechanisms such as program libraries and object hierarchies. Any separate modeling of architectural designs is informal and intuitive, and lives on whiteboards, in PowerPoint slides, or in the developers' heads. While this approach may be adequate for individuals and very small teams, it makes it difficult to understand key characteristics of the system among the details of the implementation of the business logic. Furthermore, it becomes much more difficult to manage the evolution of these solutions as their scale and complexity increases, as the system evolves over time, or when the original members of the design team are not directly accessible to the team maintaining the system (Brown 2004).

Code Visualization: As developers create or analyze an application, they often want to visualize the code through some graphical notation that aids their understanding of the code's structure or behavior. It may also be possible to manipulate the graphical notation as an alternative to editing the text-based code, so that the visual rendering becomes a direct representation of the code. Such rendering is sometimes called a code model, or an implementation model (although many feel it is appropriate to call these artifacts "diagrams" and reserve the use of "model" for higher levels of abstraction). In tools that allow such diagrams (e.g., IBM WebSphere Studio and Borland Together/J), the code view and the model view can be displayed simultaneously; as the developer manipulates either view, the other is immediately synchronized with it. In this approach, the diagrams are tightly coupled representations of the code and provide an alternative way to view and possibly edit at the code level (Brown 2004).

Roundtrip Engineering (RTE): This approach is a functionality of software development tools that provides generation of models from source code and generation of source code from models; this way, existing source code can be converted into a model, be subjected to software engineering methods and then be converted back. Round-trip engineering encompasses two engineering practices, forward engineering and reverse engineering. And what are these? In short, they are terms that represent a relationship between diagrams and code. The idea of Roundtrip Engineering is closely related to reverse engineering. Reverse engineering can be defined as the process of reconstructing the design of a product from the product itself. Assume that there is a reverse engineering procedure that is always able to give the design of a given product. Now assume that there is a procedure that will always generate the product from a given design. If a design is reverse engineered from a product, used to generate a product and the generated product is identical to the original product then this is a roundtrip engineering system (A. Henriksson 2003).

Model Centric: In this approach, the system models have sufficient detail to enable the generation of a full system implementation from the models themselves. To achieve this, the models may include, for example, representations of the persistent and non-persistent data, business logic, and presentation elements. If there is any integration with legacy data and services, the interfaces to those elements may also need to be modeled. The code generation process may then apply a series of patterns to transform the models to code, frequently allowing the developer some choice in the patterns that are applied (e.g., among various deployment topologies). This approach frequently makes use of standard or proprietary application frameworks and runtime services that ease the code generation task by constraining the styles of applications that can be generated. Hence, tools using this approach typically specialize in the generation of particular styles of applications (e.g., IBM Rational Rose Technical Developer for real-time embedded systems and IBM Rational Rapid developer for enterprise IT systems). However, in all cases the models are the primary artifact created and manipulated by developers (Brown 2004).

A *model-only:* In this approach developers use models purely as aids to understanding the business or solution domain, or for analyzing the architecture of a proposed solution. Models are frequently used as the basis for discussion, communication, and analysis among teams within a single organization, or across multi-organizational projects. These models frequently appear in proposals for new work, or adorn the walls of offices and cubes in software labs as a way to promote understanding of some complex domain of interest, and to establish a shared vocabulary and set of concepts among disparate teams. In practice, the implementation of a system, whether from scratch or as an update to an existing solution, may be disconnected from the models. An interesting example of this is the growing number of organizations that outsource implementation and maintenance of their systems while maintaining control of the overall enterprise architecture (Brown 2004).

There is a significant question, which explores that which of these approaches is adapted by MDD; unfortunately none there isn't an exact answer, it could be some things same as Model Centric approach but it should generate code directly and automatically from a visualized model.

What exactly do you mean by "visualize" a model? It's basically to graphically represent code syntax and code concepts or domain concepts and structures. A picture is worth a thousand words. It's much better to see a UML diagram with a bunch of boxes with lines connecting them than to try and read through source code. Domain visualization is basically going beyond the code to model more abstract concepts, such as 'what is a customer,' 'what is a purchase order' and 'what is an address.' The idea is to come up with a domain-specific language. This is a specialized notation that enables you to have the business analyst communicate in a very intuitive fashion. For instance, you wouldn't use UML to prove a calculus theorem; you would use the domain-specific language of calculus.

The model should be platform independent, which basically decouples you from underlying technology and buffers you from changes in the technology. And it allows you to redeploy on different technology, like the ability to deploy onto Java or .Net. Then you bind that platform independent representation to a specific technology and you create a platform-specific model such as .Net.

From a MDA perspective, it all is about creating usable models that allow you to know about your code assets because it's very easy to inventory and visualize them. And then you can extend those assets through creating inheritance and derived models.

Model-Driven Development is a software engineering approach that aims to push this idea one step further. It proposes a software development methodology in which software is developed not by writing code directly in implementation languages, but by constructing high level models that can be transformed into code by automated transformation engines and code generators, as illustrated in Figure 3 below (H. N. Pham 2007).

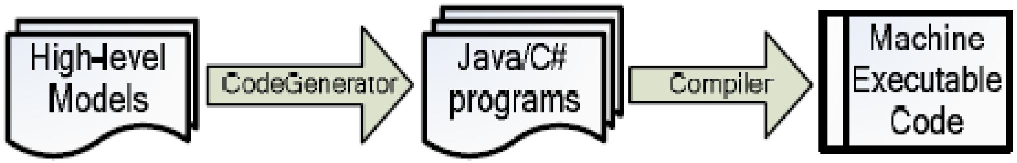


Figure 4. Sequence of MDD (H. N. Pham 2007)

As you can see in the diagram, firstly, we have to figure out a model from our system which should be visualized by business analysis and then generate the program which will be compiled and produce the machine executable code.

2.1.1.1 Benefits of MDD

Following the concept of high level programming languages; we can point on three benefits of MDD (H. N. Pham 2007):

- *Easier software specification, understanding, and development:* high-level modeling concepts, as compared to those found in implementation languages, are much closer to the real concepts in the problem domain, so we could have much easier development.

- *The ability to generate any where:* since the concepts used in the models are less bound to the underlying implementation technology, software is less susceptible to technological change. This makes software maintenance easier and more economical.
- *Reusable:* since expert implementation knowledge is encoded into the transformer, it can easily be reused and shared between different projects and teams, increasing both the productivity and quality of software development.

2.1.1.2 MDD's Approaches

Two MDD's approaches are the Model Driven Architecture (MDA) by the Object Management Group (OMG), and the Software Factories framework (SF) by Microsoft Corporation. Both of these methodologies call for the treatment of models as the primary artifacts – as opposed to an overhead that consumes development resources – in the software development process. They differ however, on how general (or specific) those artifacts should be, and how they are to be transformed to produce the actual implementation of the system (H. N. Pham 2007).

MDA proposes to apply the Unified-Modeling Language (UML). Due to aspects like platform independence and reusability, the software system is supposed to be modeled in three major steps, described further down. SF, as proposed by Microsoft, is an entire software development paradigm, which makes use of Domain Specific Modeling (DSM) (DSM Publication 2007).

2.2 Model Driven Architecture (MDA)

Referring the idea that models are vital and necessary to handle complexity in software development, Model-Driven Architecture (MDA) specifies a process for creating models.

As Brown Said "There are many views and opinions about what MDA is and is not. However, the most authoritative view is provided by the Object Management Group (OMG). Why does the OMG's view of MDA matter so greatly? As an emerging architectural standard, MDA falls into a long tradition of OMG support and codification of numerous computing standards over the past two decades. The OMG has been responsible for the development of some of the industry's best-known and most influential standards for system specification and interoperation, including the Common Object Request Broker Architecture (CORBA), OMG Interface Definition Language (IDL), Internet Inter-ORB Protocol (IIOP), Unified Modeling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI), Common Warehouse Model (CWM), and Object Management Architecture (OMA). In addition, OMG has enhanced these specifications to support specific industries such as healthcare, manufacturing, telecommunications, and others (Brown 2004)."

As a glance on Processes of MDA in Figure 5 we can understand that three principles underlie the OMG's view of MDA (Brown 2004):

- *Computation Independent Model (CIM):* Models expressed in a well-defined notation are a cornerstone to understanding systems for enterprise-scale solutions.

- *Platform Independent Model (PIM)*: The building of systems can be organized around a set of models by imposing a series of transformations between models, organized into an architectural framework of layers and transformations.

- *Platform Specific Model (PSM):* A formal underpinning for describing models in a set of meta-models facilitates meaningful integration and transformation among models, and is the basis for automation through tools.

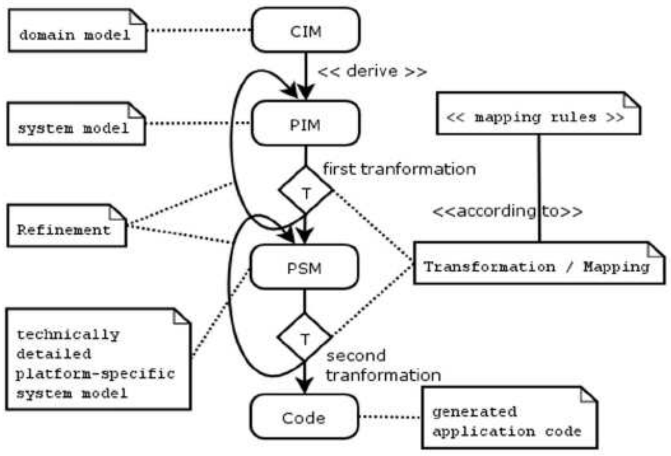


Figure 5. Process Model of MDA (Brown 2004)

Three ideas are important here with regard to the abstract nature of a model and the detailed implementation it represents (Brown 2004):

- *Model classification*: We can classify software and system models in terms of how explicitly they represent aspects of the platforms being targeted. In all software and system development there are important constraints implied by the choice of languages, hardware, network topology, communications protocols and infrastructure, and so on. Each of these can be considered elements of a solution "platform." An MDA approach helps us to focus on what is essential to the business aspects of a solution being designed, separate from the details of that "platform."

- *Platform independence:* The notion of a "platform" is rather complex and highly context dependent. For example, in some situations the platform may be the operating system and associated utilities; in some situations it may be a technology infrastructure represented by a well-defined programming model such as J2EE or .Net; in other situations it is a particular instance of a hardware topology. In any case, it is more important to think in terms of what models at different levels of abstraction are used for what different purposes, rather than to be distracted with defining the "platform."

- *Model transformation and refinement*: By thinking of software and system development as a set of model refinements, the transformations between models become first class elements of the development process. This is important because a great deal of work takes places in defining these transformations, often requiring specialized knowledge of the business domain, the technologies being used for implementation, or both. We can improve the efficiency and quality of systems by capturing these transformations explicitly and reusing them consistently across solutions. If the different abstract models are well-defined, we can use standard transformations. For example, between design models expressed in UML and implementations in J2EE, we can, in many cases, use well-understood UML-to-J2EE transformation patterns that can be consistently applied, validated, and automated.

2.3 Software Factory

Increasingly complex and rapidly changing requirements and technologies are making development increasingly difficult. Promising advances have been made, however, in component based and model driven architecture, software architecture, aspect oriented programming, and requirements, process and software product line engineering. We will present Software Factories, a paradigm for automating software development that integrates these advances to increase agility, productivity, and predictability across the software life cycle. We will show a worked example of a software factory and perform small group exercises that help participants explore this approach. Participants will learn about the software factory schema, a graph of viewpoints used to separate concerns, relating work done at one level of abstraction, in one part of a system, or in one phase of the life cycle, to work done at other levels, or in other parts and phases, and about how the schema can be used to deliver guidance and to support its enactment through model transformation, constraint checking and other techniques. We will also describe the software factory life cycle and show how software factories can be specialized and composed. Finally, we will discuss software supply chains and show how Software Factories compose across organizational boundaries.

Microsoft introduces Software Factories (SF) as a new software development paradigm. SF primarily focuses on product Line Development, which copes with developing a set of similar but distinct products. In this context SF relies heavily on models and automation, which are basic concerns of MDD. This paper will focus on the MDD concerned aspects of SF (Demir 07).

A Software Factory is a Software Product Line that configures extensible tools, processes, and content using a software factory template based in a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework based components [3].

A Software Factory has two central elements, a Software Factory Schema and a Software Factory Template. A SF Schema defines, categorizes and summarizes the artifacts and assets required to build a software product line. It can be seen as a recipe listing ingredients, tools and the application process. A SF Template is based on the SF Schema and represents the implementation of the SF Schema that means that all defined assets and artifacts have to be built and made available. The implementation comprises among others developing DSLs. The SF Template can be seen as a bag of groceries containing the ingredients listed in the recipe (SF Schema) (Demir 07).

2.3.1.1 Software Factory Schema (Greenfield 2007)

A software factory schema is a document that categorizes and summarizes the artifacts used to build and maintain a system, such as XML documents, models, configuration files, build scripts, source code files, SQL files, localization files, deployment manifests and test case definitions, in an orderly way, and that defines relationships between them, so that we can maintain consistency among them.

A software factory schema is represented as a directed graph whose nodes are viewpoints and whose edges are computable relationships between viewpoints called mappings. This allows nodes that would not be adjacent in a grid representation to be related. Also, it relaxes the artificial constraint imposed by a grid that the viewpoints must fit into neat classification schemes, creating rows and columns. Finally, and most importantly, it allows the schema to reflect the software architecture. So, for example, a schema for a family of business applications might contain several clusters of viewpoints, one for each subsystem like customer management, catalog management, or order fulfillment. The viewpoints in each cluster might then be further grouped into subsets reflecting the layered architecture of each subsystem, as illustrated in Figure 6.

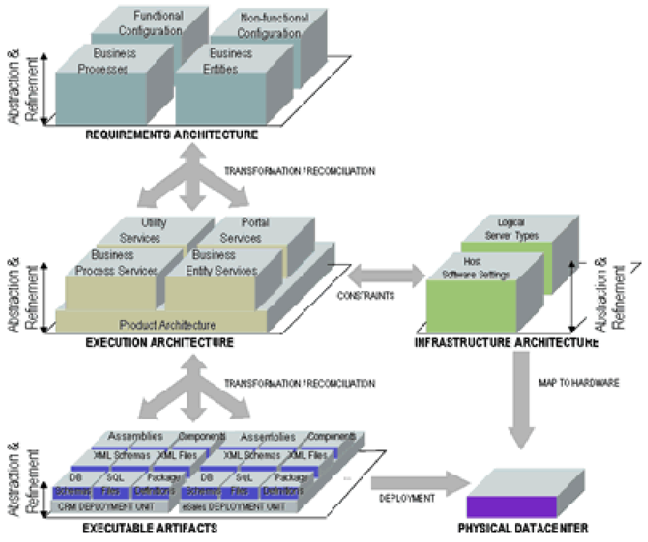


Figure 6. Software Factory (Greenfield 2007)

A software factory schema describes the artifacts that comprise a software product, just as an XML schema describes the elements and attributes that comprise a document, and a database schema describes the rows and columns that comprise a database. Like an Architectural Description Standard (ADS), a software factory schema is a template for describing the members of a software product family. Despite this similarity, however, there are several major differences between a software factory schema and ADS:

While an ADS deals only with architecture, a software factory schema deals with many other aspects of a software product family, such as requirements, executables, source code, test harnesses and deployment artifacts. While an ADS organizes design documentation, a software factory schema organizes development artifacts.

While an AD implies a software product family, it does not explicitly identify one, or incorporate mechanisms to support family based development, such as a way to express how the members of the family differ from a family archetype. A software factory schema, on the other hand, targets a specific software product family and can be instantiated and customized to describe a specific family member in terms of its differences from the family archetype.

While an ADS does not necessarily support automation, a software factory schema can be implemented by a software factory template to automate software development tasks, as we shall see shortly.

Of course, the essential property of a software factory schema is that it provides a multi dimensional separation of concerns based on various aspects of the artifacts being organized, such as their level of abstraction, position within architecture, functionality or operational qualities. According to Coplien (Coplien 1999):

We can analyze the application domain using principles of commonality and variation to divide it into sub domains, each of which may be suitable for design under a specific paradigm.

We can now see the grid as a two dimensional projection of the graph that plots one or more aspects on the horizontal axis and different levels of abstraction on the vertical axis. Another two-dimensional projection is an aspect plane, which projects related viewpoints onto part of the product architecture, providing a consolidated view from that viewpoint. Examples of aspects planes include logical data, security policy and transaction planes.

A software factory schema essentially defines a recipe for building members of a software product family. Clearly, the viewpoints describe the ingredients and the tools used to prepare them, but where is the process of preparing them described? Recall that a process framework is constructed by attaching a micro process to each viewpoint, describing the development of conforming views, and by defining constraints like preconditions that must be satisfied before a view is produced, post conditions that must be satisfied after it is produced, and invariants that must hold when the views have stabilized. This framework defines the space of possible processes that could emerge, depending on the needs and circumstances of a given project. Clearly, there is a resemblance between a process framework and a software factory schema. The viewpoints of a software factory schema already define micro processes for producing the artifacts they describe. Adding constraints to a software factory schema to govern the order of execution makes it a process framework, as well. We now have a recipe for the members of a product family. It defines the ingredients, the tools used to prepare them and the process of preparing them.

2.3.2 Software Factory Templates (Greenfield 2007)

As Greenfield said “If all we have is the software factory schema, then we can describe the assets used to build family members, but we do not actually have the assets. Before we can build any family members, we must implement the software factory schema, defining the DSLs, patterns, frameworks and tools it describes, packaging them, and making them available to product developers. Collectively, these assets form a *software factory template*.

A software factory template includes code and metadata that can be loaded into extensible tools, like an Interactive Development Environment (IDE), or an enterprise life cycle tool suite, to automate the development and maintenance of family members.

We call it a software factory template because it configures the tools to produce a specific type of software, just as a document template loaded into a tool like Microsoft Word or Excel configures it to produce of a specific type of document."

2.3.3 Systematic Reuse (Greenfield 2007)

One of the most important innovations in software development is defining a family of software products, whose members vary, while sharing many common features. A family provides a context in which problems common to the family members can be solved collectively. This enables a more systematic approach to reuse, by letting us identify and differentiate between features that remain more or less constant over multiple products and those that vary. A software product family may consist of either components or whole products.

Software product lines exploit product families, identifying common the features and recurring forms of variation in specific domains to make the production of family members faster, cheaper, and less risky. Products developed as family members reuse requirements, architectures, frameworks, components, tests and many other assets.

Figure 7 describes the key tasks performed and artifacts produced and consumed in a product line. Product line developers build production assets applied by product developers to produce family members in much the same way that platform developers build device drivers and operating systems used by application developers. A key step in developing the production assets is to produce one or more domain models that describe the common features of problems in domains addressed by the product line, and the recurring forms of variation. These models collectively define the scope of the product line, are used to qualify prospective family members. Requirements for family members are derived from them, providing a way to map variations in requirements to variations in architecture, implementation, executables, development process, project environment, and many other parts of the software life cycle.

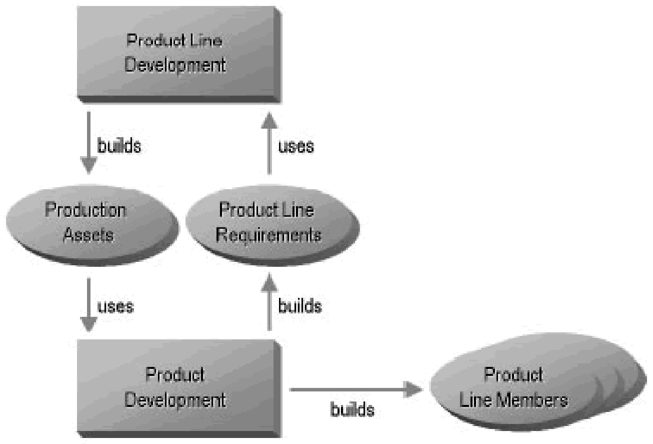


Figure 7. Reusability of Software Factory (Greenfield 2007)

2.4 Domain Specific Modeling and Language (DSM & DSL)

We are not interested in using the general purposed languages (GPL) (Garwick 1968) because, GPLs, compared with DSLs, use a vocabulary that is simple and basic enough to describe any domain without the specifics. The same level of expression and understanding of a domain is possible using GPLs, but the expected level of knowledge regarding the domain and the general language is considerably higher compared to a DSL approach. A major advantage of a DSL is that it requires significantly less time to understand and to communicate details of a domain. It also requires less time to learn to use the pertinent tooling (Kovari 2004).

For example, business applications are often implemented using complex software solutions, but most of the solutions use the same building blocks (patterns) to deliver business functions. A DSL enables you to abstract the software solution and hide the implementation details. A DSL can also use the vocabulary from the business domain and provide the translation for the IT domain.

We are also not interested in models rendered by hand on white boards, or on note pads. We are interested in models that can be processed by tools, and we propose to use them in the same way that we currently use source code. Models used in this way cannot be written in languages designed for documentation. They must be precise and unambiguous. In order to raise the level of abstraction, a modeling language must therefore target a narrower domain than a general purpose programming language. We find that the Unified Modeling Language (UML), in particular, is suitable for sketching, but not for the capture of high fidelity metadata used to generate models, code and other software development artifacts. A detailed discussion of the issues around using UML as language for MDD is beyond the scope of this article, but Cook provides a cogent analysis, and Fowler adds several insights on his blog page.

A language that meets these criteria is called a Domain Specific Language (DSL), because it models concepts found in a specific domain. A DSL is defined with much greater rigor than a general purpose modeling language. Like a programming language, it may have either textual or graphical notation.

A core principle of the SF approach is to enable a high degree of reuse of existing assets and development of new reusable assets. The development of a specific member of a product family comprises reusing existing assets and developing variable assets for that specific member. The SF approach uses the concept of Domain-Specific Modeling (DSM), which utilizes Domain-Specific Languages (DSLs) for modeling (Demir 07).

2.4.1 Basic Concepts

In the scope of engineering we have two approaches:

Generic approach: providing the general solution for many problems in a certain area (A. V. Deursen 2000).

Specific approach: searching for specific solutions in smaller area (A. V. Deursen 2000).

It's clear that when we want to be more general we should cover much more points, instances, cases and even exceptions so absolutely our solution is hard to be optimal but when we are in specific concept we can be more optimal than the pervious approach.

Currently, we can have three approaches in solving problems in a well-defined application domain:

Subroutine libraries contain subroutines that perform related tasks in well-defined domains like, for instance, differential equations, graphics, user-interfaces and databases. The subroutine library is the classical method for packaging reusable domain-knowledge (A. V. Deursen 2000).

Object-oriented frameworks and component frameworks continue the idea of subroutine libraries. Classical libraries have a flat structure, and the application invokes the library. In object-oriented frameworks it is often the case that the framework is in control, and invokes methods provided by the application-specific code (A. V. Deursen 2000), (R. E. Johnson 1988), (M. E. Fayad 1997) .

Domain Specific Languages are small, usually declarative, languages that offer expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library (A. V. Deursen 2000).

2.4.2 Migration from Abstract to Real and from Meta to Instance

As you can see in Figure 8, first of all we should define a meta and abstract language to develop a model that the language is called Domain Specific Language (DSL) and the model is Domain Specific Model (DSM).

DSL: A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain (A. V. Deursen 2000).

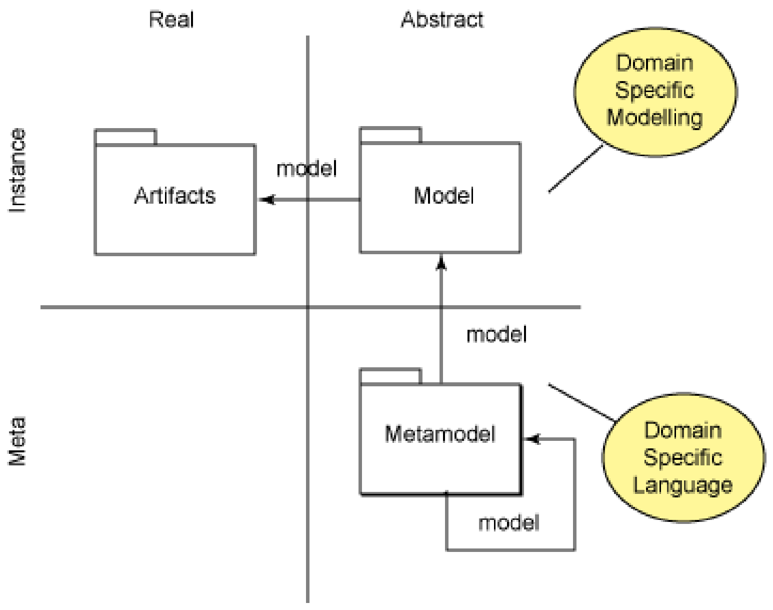


Figure 8. Model Migration (Kovari 2004)

For advantages of DSL we can name (M. E. Fayad 1997):

- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs (M. E. Fayad 1997).

- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes (D. A. Ladd 1994).
- DSLs enhance productivity, reliability, maintainability (M. E. Fayad 1997), (E. K. A. V. Deursen 1998), (R. B. Kieburtz 1996) and portability (R. M. Herndon 1988).
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge (M. E. Fayad 1997).
- DSLs allow validation and optimization at the domain level (A. Basu 1997), (Bruce 1998), (V. Menon 1999).
- DSLs improve testability following approaches such as (E. G. Sirer 1999).

Its disadvantages are (M. E. Fayad 1997):

- The costs of designing, implementing and maintaining a DSL.
- The costs of education for DSL users.
- The limited availability of DSLs (Krueger 1992).
- The difficulty of finding the proper scope for a DSL.
- The potential loss of efficiency when compared with hand-coded software engineering

Modeling with DSL and UML are at the opposite ends of the spectrum, in some respects. UML is a *unified* (or general) modeling language; it can support literally any model. DSM is a domain specific modeling language. It can only support specific types of models (M. E. Fayad 1997). Considering the concept of DSL and UML we can mark the DSL as more practical approach than UML and absolutely when you use UML you can reach the Model Centric Approach or maybe also roundtrip but as far as you see DSL tries to visualize the concept with code generation including some aspects of business logics.

DSM (Kovari 2004)*:* In engineering sciences like IT, experts use models, diagrams, and sketches to describe specific details of a problem or a solution. The need for visual representation arises from the high degree of complexity surrounding the industry. Abstraction and automation justify the need for visual modeling.

Engineers in IT work with various inputs, outputs, work products, and deliverables. One output may become the input for another work product. Models and diagrams are often part of or are the actual work products. The flow of information, the reuse of previous results, and the automation of the workflow justify the use of models over simple diagrams.

Figure 2 shows the most basic approach for building an application for modeling, modeling itself, and producing various artifacts from the models in domain specific areas.

According Figure 2, starting from the meta-model, the developer has to establish the meta-model or language for the specific domain. Constraints can enrich the meta-model to ensure the semantic correctness and validation of the model instances. Most of the graphical editor can be generated from the meta-model, but other parts have to be manually defined. A new set of constraints can be specified for the graphical editor because the graphical representation may use different constructs for modeling than the original meta-model. Model instances can be created and edited using the graphical editor. These models are the results of model-driven development. Models become the input to transformations to generate the final artifacts (code, for example).

2.5 Comparison of MDA and SF (Demir 07)

As Demir Said "The MDA solution shows that productivity can be increased by applying the MDA approach. Certainly, the utilized tool plays an important role, but a better productivity can be achieved particularly due to the omission of the implementation phase. A disadvantage in this case is that the learning phase for the MDA tool is due to its complexity and individuality very time-consuming. Nevertheless, the modeling process can be started immediately, since the modeling language (UML) is already provided, apart from the fact that specific UML Profiles are needed and not granted by the MDA tool.

The SF solution shows that the SF methodology can increase productivity as well, basically for the same reason as in the case of MDA, the implementation part is omitted. But, before reaching this point, the expense for the SF approach is much higher, because the DSL has to be developed first, which is time-consuming and sophisticated, and requires expert knowledge about the problem and the solution domain. This fact delays the start of the modeling process.

Once a DSL is created, a better efficiency can be achieved, because a DSL comprises domain concepts and thus it is closer to the problem domain. This fact facilitates involving business stake holders into the specification process to avoid misinterpretation and confusion. This is an advantage for DSLs, since the comprehension of UML models require UML experts.

Quality and reliability of a software system can be improved as well in both approaches, particularly due to the reason that the generated code is less error prone, because of its generation according to a scheme, rules or code-templates. Provided that the model interpreter works properly, the generated code is more reliable than handcrafted code that usually contains bugs, because a developer tends to make mistakes. Furthermore, the generated applications exactly meet their specification in form of models, since they are generated according to them.

Certainly, MDA and SF apply similar methods and techniques for modeling and mapping, but they have distinct objectives. Due to the high expense of developing a DSL and a code generator, the SF approach is only recommendable for developing Product Lines, because this expense has to be compensated somehow. However, once a DSL and the code generator are developed, the costs for generating the product line members are very low due to their similarity. The overall costs of a product line development can then be distributed on the amount of all members, which makes the SF approach productive. Otherwise, for One-Off development, the SF approach would be expensive in terms of time, budget and resources. MDA on the contrary can be used for One-Off Development and Product Line development, since there is no additional expense to compensate. Product Line development with the MDA approach would even increase the regular expected degree of productivity, because the first model could be reused for the other product line members.

Fact is that MDA is a pure MDD approach and focuses on platform independence, while SF is an entire software development methodology and focuses on product line development. UML as the standard modeling language for MDA is a general purpose language, which has to be specialized and constrained with Profiles to be appropriate for MDD. A DSL in contrary is supposed to be developed for a specific domain from beginning, without specializing and constraining afterwards, in this manner DSLs can be very efficient within that domain, but also very useless in other domains.

On the whole, both approaches have their strengths and weaknesses; none of them is clearly in advance. Depending on the purpose they are applied for, they demonstrate different strengths and weaknesses. An appropriate problem domain, professional developers, a suitable tool and a precise idea of the intended products or product family, can guarantee each approach's benefits."

2.6 Microsoft Visual Studio DSL Tools

Domain-Specific Language Tools allow Visual Studio 2005 developers to create their own graphical designers and code generation tools like the ones you find in Visual Studio today, such as the Class Designer.

You can use Domain-Specific Language Tools to generate visual designers that are customized for your problem domain. For example, you can create a tool to describe concepts that are specific to how your organization models business processes. If you are building a state chart tool, you can describe what a state is, what properties a state has, what kinds of states exist, how transitions between states are defined, and so on. A state chart that describes the status of contracts in an insurance company is superficially similar to a state chart that describes user interaction among pages on a Web site. However, their underlying concepts differ significantly. By creating your own domain-specific language and custom generated designer, you can specify exactly what state chart concepts you need in your tool (Domain-Specific Language Tools 2007).

DSL Tools takes a graphical approach to DSL construction. When you start a new DSL project what you are provided with is a DSL Designer that allows you to create a "diagram editor" in the form of a generated VS Designer. This might seem confusing because what you have is a Designer of Designer which you use to create a custom Designer which the end user then uses to create a diagram and generate code. It begins to make more sense when you start to make use of it, but it helps to realize that you first use the DSL Designer to specify the type of diagram the user can create – a flow chart, class diagram, workflow diagram – essentially any collection of boxes and arrows subject to the rules that you specify for how they can be arranged to create a diagram (James 2007).

Once you have used the DSL Designer to specify your custom designer you use the standard templates to generate code which you then run. The generated code creates your custom designer which you can run in VS. It has a toolbox full of the entities you specified and you can use it to create an instance of the type of diagram you specified (James 2007).

2.6.1 Building a Designer using Microsoft DSL Tools

The goal of this activity is developing an environment such as one shows in Figure 9, to enable developer for modeling his/her system and then generated directly the code.

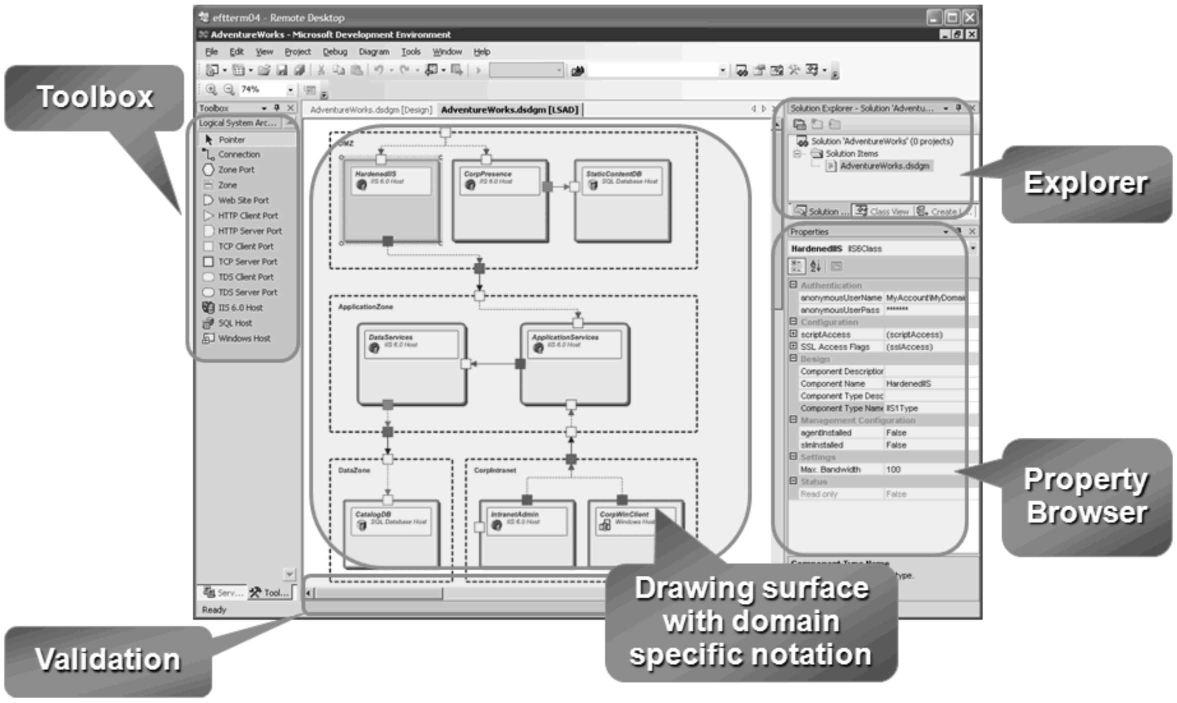


Figure 9. A Microsoft Visual Studio Graphical Designer, built by DSL Tools

As you can see, we have following parts in these kinds of modeling environments:

Toolbox: as far as you may familiar with visual environments specially .Net's ones for doing stuffs of modeling we may have some toolboxes which help us to develop and represent our intention and what are in our mind.

Drawing Surface: this part takes a place for your model and then the environment supposes that your model is entirely summarized in this place.

Property Browser: in this part the environment shows the defined properties of selected component in the drawing surface.

Explorer: explorer part of the environment can show and brow all components that you have defined in the drawing surface using toolbox.

Validation: to be sure about the validity of your model this is a tool helps you to check your model before generating codes.

After illustrating the goal, we can focus on the structure of Microsoft DSL Tools and where they are located. Fist of all, at the base of structure we can see modeling platform of VS which can generate all modeling platforms in Visual Studio, so to achieve our desired environment we should develop our libraries on this platform. What the DSL Tools can do for us is presenting an environment that we should use it for more high level work than using the modeling platform for itself. Figure 10, tries to clarify this concept.

As far as you can see in Figure 10, DSL tools are developed to produce new designer on the .Net framework using modeling platform libraries. In this case for defining your language with tools have been proposed in DSL and running the program we could have such an environment.

to four libraries and one engine as follows:

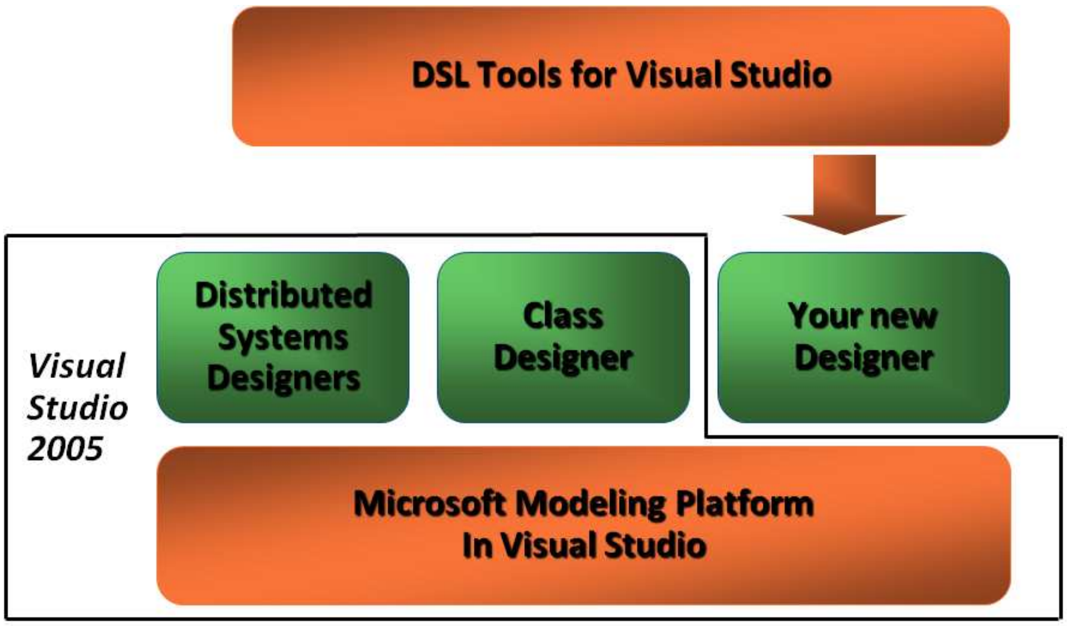

DSL Tools for Visual Studio
Visual Studio 2005
Distributed Systems Designers
Class Designer
Your new Designer
Microsoft Modeling Platform In Visual Studio

Figure 10. Where Microsoft DSL tools seat.

Figure 11 presents modeling platform structure and which parts it has. In this case we can divide this platform

Shell Framework: The Premier Partner Edition (PPE) provides a version of Visual Studio that includes the Visual Studio IDE, the debugger, and source code control integration. No programming languages are included. Although PPE does not include programming languages, PPE does provide a framework that lets you add programming languages (Microsoft 2005). This framework provides a core IDE for users to develop their own custom programming language or development tools, so it doesn't provide languages or compilers, or a lot of the content of Visual Studio and it will be available in two modes, integrated and isolated (Frye 2007).

Validation Framework: These libraries can be used for applying reusable and customizable rules to properties and methods to provide validation for your strongly typed business objects.

Domain Model Framework: These libraries consist of some non-accessible classed used for developing design surface libraries and template engine.

Design Surface Framework: These libraries use the domain model framework and build the graphical designer specified for your language.

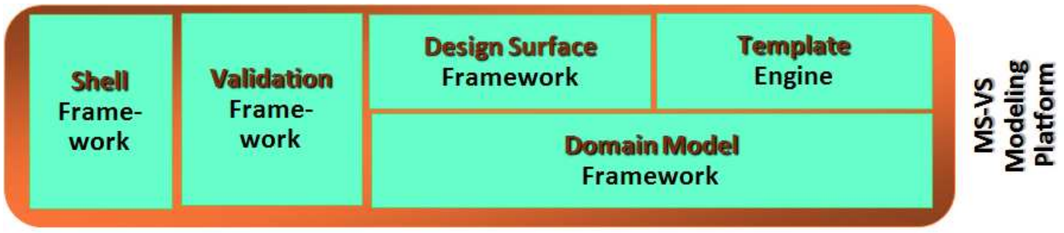
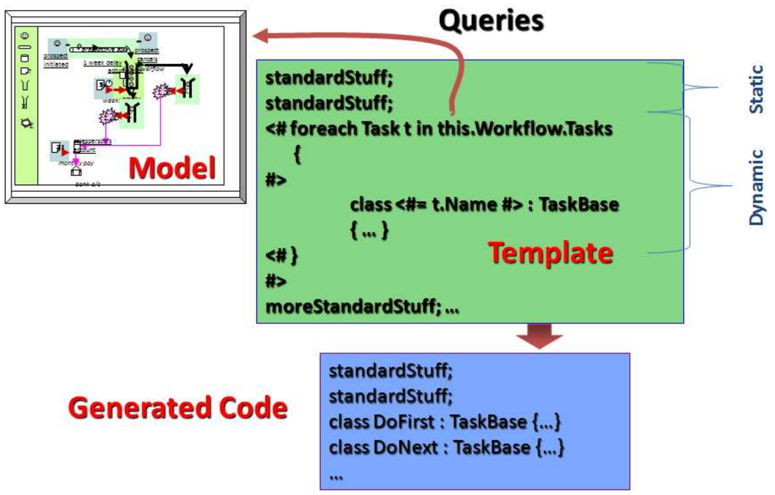


Figure 11. Microsoft Visual Studio Modeling Platform

Template Engine: First of all, let us to explain what a template is? The templates can access the meta objects directly; properties of the meta objects can be used to provide data for template evaluation as shown in the pervious illustration, and then generate some codes. To generate the end application we should run a code like one is showed in Figure 12, which is divided in two types of code; the first one is more standard stuffs and the second one is model dependent stuffs. For more clear words we must call them the constant part and the dynamic part.



```
standardStuff;
standardStuff;
<# foreach Task t in this.Workflow.Tasks
    {
#>
        class <#= t.Name #> : TaskBase
        { ... }
<# }
#>
moreStandardStuff; ...
```

```
standardStuff;
standardStuff;
class DoFirst : TaskBase {...}
class DoNext : TaskBase {...}
...
```

Figure 12. Template Based Code Generation

To sum up the Microsoft DSL Tools, you may consider the illustration diagram in Figure 13 which provides a high-level overview of how you can design, customize, test, and deploy a domain-specific language. As you can see in this figure, first of all we should define our language and with pressing the F5, you can see a graphical environment that designer can use it to model his/her system. Then with using template files (.tt files) you can read the model and then based on what you have read generate codes.

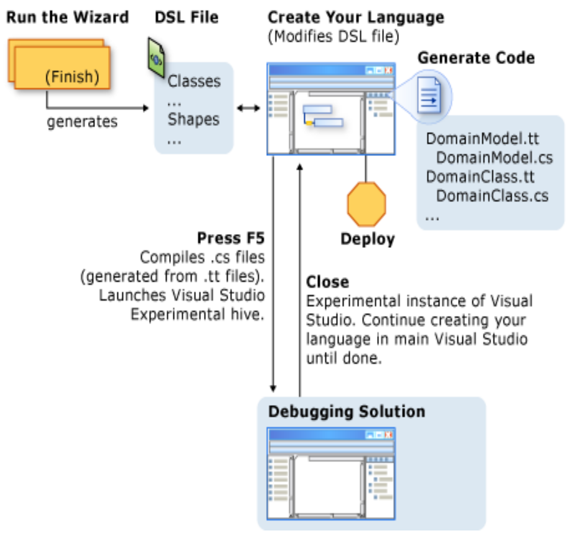


Figure 13. Illustration Diagram for Roadmap in Microsoft DSL tools (Microsoft, Microsoft DSL Tools 2005)

For more technical things about the Microsoft DSL tools please go to Technical Tutorial Appendix in this report.

2.7 Related Works

In this section we are going to explain some of works with the same goal as ours, actually we want to present the tools and frameworks that they have already existed in generating web applications.

2.7.1 AndroMDA (Bhatia 2006)

AndroMDA (pronounced "Andromeda") is an extensible generator framework that adheres to the Model Driven Architecture (MDA) paradigm. It transforms UML models into deployable components for your favorite platform. While AndroMDA ships with cartridges that can generate code for several platforms and technologies, this tutorial will focus on generating a Java application.

During development of large applications, most architects and developers already create class diagrams and data diagrams. These diagrams are usually made in tools like Visio, and the resulting artifacts are static pictures. When code changes, the diagrams must be updated. With AndroMDA, these diagrams become a living part of your application -- they are used to generate large portions of your application, and hence always reflect the current state of the system. When you need to modify your application, you change the model first, regenerate the code, and then add or update custom code as necessary. Thus, you get a production quality application out of assets that you had to create anyway.

According Figure 14 AndroMDA provides several cartridges out-of-the-box. For example, the Hibernate and Spring cartridges generate robust service and data layers for your application. In addition, database schema can be exported to script files to allow the creation of your application's database. There is also an easy way to map your model to an existing schema if your database has already been defined. If you wish to generate custom artifacts from your model, you can write a custom cartridge to accomplish this.

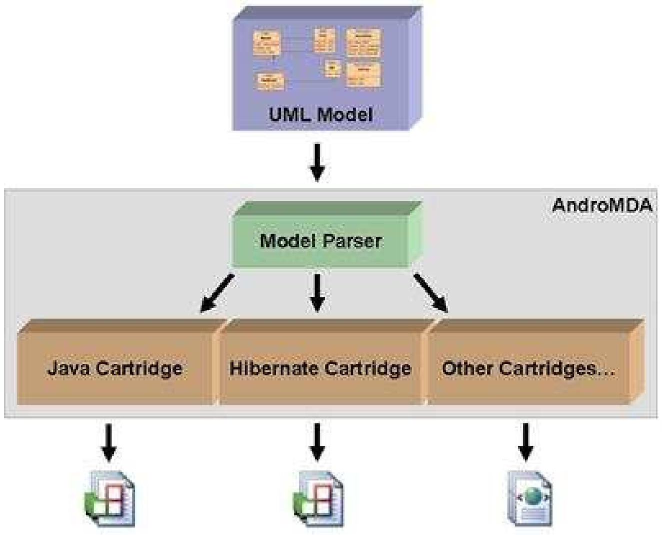


Figure 14. AndroMDA Architecture (Bhatia 2006)

2.7.1.1 Evaluation of AndroMDA

Remarkably, the AndroMDA (Bhatia 2006), (Kozikowski 2005) is an open source MDA generator also proposes a so-called cartridge that is able to generate very sophisticated Java Struts web applications from UML Activity Diagrams. It is called Bpm4Struts. However, Bpm4Struts expects input models with a much higher detailedness. I believe that business users – who are targeted as application engineers would be overextended when asked to use AndroMDA. Furthermore, AndroMDA does not address the concept of software product lines at all.

Understanding new tools and technologies can be a daunting task, and AndroMDA is no exception, but when you will take a look at this environment you can find how it's complicated and how much efforts need to understand and be professional, so we can't imagine this environment with a lot of abilities can use as much as very simple tools for developing web applications.

2.7.2 Web Relational Blocks (WebRB) (A. Leff 2007)

Web Relational Blocks (WebRB) is a browser-based visual editor and run-time environment that enables developers to visually assemble Web applications without adding any imperative code. WebRB is made for developers of "enterprise" Web applications: multi-page applications, containing non-trivial GUI (graphical user interface) and business logic, whose data reside in relational databases.

WebRB increases Web developer productivity in the following ways:

- The application's GUI is developed visually (no imperative code) by dragging HTML widgets off a palette. The entire application is assembled in the visual editor.

- All the application's components ("blocks") use the same API and have the same visual representation. Because blocks have a common interface, they are easily combined to produce the desired effect. In contrast with languages such as HTML and PHP, developers use a single API for all parts of the application and assemble the application at a higher level of abstraction.

- The "code, test, and debug" development cycle is improved because applications are directly executed from the visual editor.

Incremental construction is encouraged because only a small set of blocks is required for starting a working application. Blocks can be added, removed, or rewired at any time, and the application can be immediately validated and re-executed.

Every element in a WebRB application is represented as a block with optional input and output pins. Wires are used to connect blocks, and represent data-flow between the connected blocks. Figure 15 shows how wires are used to specify relational data-flow between two blocks. Wires connect input pins to output pins: in this Figure, the two text-input blocks on the left – each with an output pin named “outputValue” – transmit data to the JOIN block’s two input pins (“input0” and “input1”). The result of the JOIN block is available from its “output” pin. An input pin implies that a block can receive relation-valued input from other blocks. An output pin implies that the block can transmit relation-valued output to other blocks. Blocks often have an “enable” pin: these are boolean-valued relations, and imply that the block operates only if some other block enables this block. An input pin can be connected to at most one output pin but need not be directly connected to that pin. For example, input pin A may be connected to input pin B which, in turn, is connected to output pin C. Data will flow from pin C to pin A, and from pin C to pin B. The only requirements are that an input pin must be connected directly or indirectly to a single output pin, and output pins may not be directly or indirectly connected to other output pins. No other restrictions are imposed on the circuit topology. A block’s semantics specify a well-defined functional transformation over their inputs to their outputs. Blocks use *relational algebra* to perform operations on relational data. For example, the JOIN block computes the relational AND of two input relations and places the result on its output pin. The WebRB visual editor contains a set of pre-supplied block prototypes, which can be in one of three flavors: model, widget, and algebra. These correspond, respectively, to the well-known Model/View/Controller paradigm. An application is simply a set of pages – graphs built from the WebRB pre-supplied block instances. Pages are blocks: i.e., they have exactly the same relational API and visual representation as pre-supplied blocks. Pages can therefore be hierarchically embedded in other pages in a recursive process (J. Rayfield 2007).

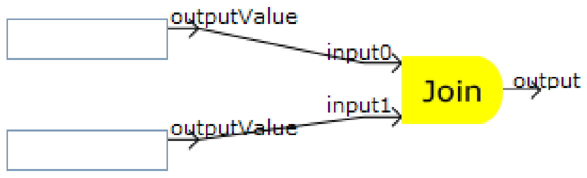


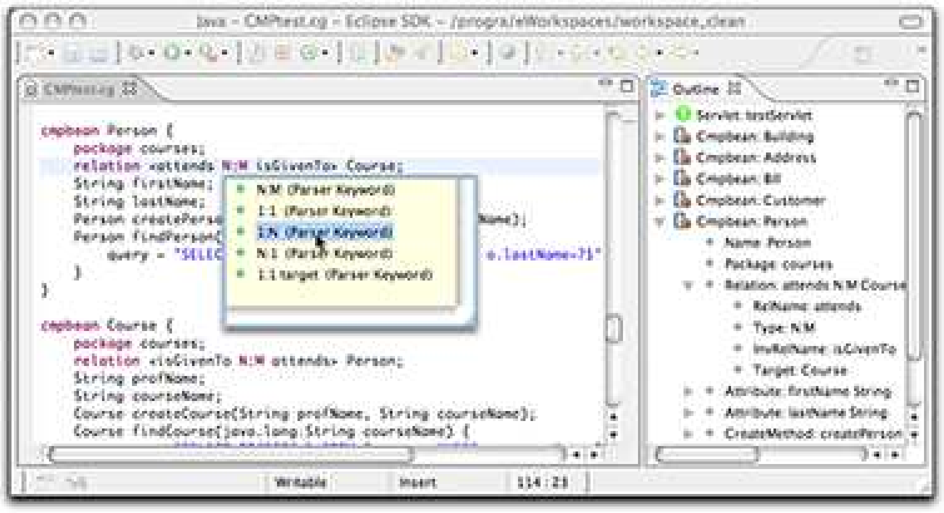
Figure 15. WebRB: Blocks and Wires (J. Rayfield 2007)

2.7.2.1 Evaluation of WebRB

As far as we have described this framework and tool, we can see that using this tool is not in that much easy for designers and if you want to use it as prototype development or as a user development environment users should be introduced with logic basics and absolutely relational theory. So I believe that WebRB is not adaptable with our goals, however it has a lot of facilities and powerful tools.

2.7.3 WebLang

WebLang is a Domain-Specific Language, developed in LTI that makes possible to define web applications with high level design. The main motivation behind WebLang is to abstract the application components with a useful model, but to remain sufficiently close to the technology to reduce the enormous gap that exists between a PIM model and implementation. The WebLang approach expects to provide a simple and realistic method for designing the architecture of a web application and a usable tool for generating a testable prototype. The WebLang development process follows the MDD approach and brings a language model as key element of the development. The language syntax is oriented towards being naturally editable for a human in comparison with XML, which is more adapted to the machine. A WebLang application is defined by the assembling of several components that can specify each structural properties, business logic, and interconnections with other components. The WebLang tool checks and compiles the model, and then generates the application in one atomic action. This approach is easier to implement than the incremental generation of application fragments. Furthermore, it provides the developers with a well-defined environment, where the whole application is defined with a unique and centralized model. All the generated files are standard and can be freely modified by the developer. The tool is integrated in the IBM Eclipse IDE (Figure 16), and is currently available for the J2EE JBoss platform, but the approach is extendable to other servers or technologies, by extending the templates or implementing new adapted modules (O. Buchwalder 2006).



```
cmpbean Person {
    package courses;
    relation <attends N:M isGivenTo> Course;
    String firstName;
    String lastName;
    Person createPers                              Name);
    Person findPerson(
        query = "SELEC                              o.lastName=?1"
    }
}

cmpbean Course {
    package courses;
    relation <isGivenTo N:M attends> Person;
    String profName;
    String courseName;
    Course createCourse(String profName, String courseName);
    Course findCourse(java.lang.String courseName) {
```

Figure 16. WebLang: Webapplication Development IDE (C. Petitpierre 2006)

All J2EE components more or less are in the form of a simple object containing attributes and methods. LTI has proposed a language, WebLang, which emphasizes this aspect. Some of the J2EE components are accessed directly by the user through a browser or a GUI, Regarding Figure 17 WebLang compiler directly creates the corresponding human machine interface. Some components (channels) depend on other components, and in that case WebLang offers statements to integrate them in the components they are linked. Finally, some components have an existence of their own. Then, WebLang offers some supports to access them (C. Petitpierre 2006).

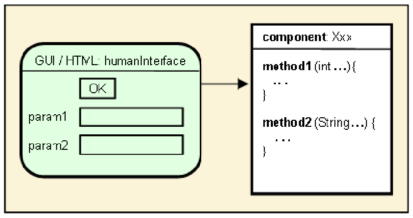


Figure 17. How WebLang mapped J2EE components (C. Petitpierre 2006)

WebLang compiler can generate files in a Java project as well as in a setting made and managed by WTP, which allows a developer to start his or her prototype with WebLang and to handle the details and maintain the application with the help of WTP later on. In most cases, both the *Run XDoclet* and *Run As > Run On Server* commands of WTP can be used instead of *CGxdoclet.xml* and *CGpackaging.xml* scripts (C. Petitpierre 2006).

The WebLang compiler uses Jet templates to produce the files composing an application. It is very easy to get these templates in the developer's environment, to modify and to use them to produce files better adapted to one's own requirements (C. Petitpierre 2006).

LTI has developed a code gen Eclipse plugin, which has been used in Defining WebLang. This plugin provides structured and partial automatic methods to develop and use *Parser, Generator and Editor* for Domain-Specific Languages (DSL), then we can develop a real Eclipse editor for our language with following features (C. Petitpierre 2006): Outline view, Completion, Syntax highlighting and Syntax error warning

As you can finf in Figure 18 shows for defining new language and then having a real Eclipse editor you should define your DSL very easily in this environment and just compile it, then you will have an editor with mentioned features.

The tutorial of WebLang can be found on (C. Petitpierre 2006).

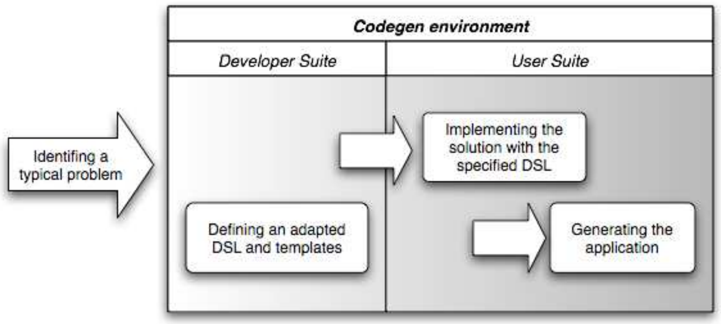

Codegen environment
Developer Suite
User Suite
Identifing a typical problem
Implementing the solution with the specified DSL
Defining an adapted DSL and templates
Generating the application

Figure 18. WebLang Codegen (O. Buchwalder 2006)

2.7.3.1 Evaluation of WebLang

As far as LTI's feedback in some experiences for developing web applications, it could meet the mentioned requirements for being as a very easy and rapid environment to develop prototypes. LTI believes to use such these kinds of environments we should make it easy to learn for designers.

2.8 State of Art

As far as we have described in this chapter, for developing we application we have possibility to use some tools developed before such as WebRB or AndroMDA but the point is most of these tools are designed for developing market ready version application and supporting every thing thus they are so complicated and complex to use so to make the development easy as much as possible we have an idea to make the development in two phase one for designers and one for developers. WebLang has followed this idea but the point is WebLang can generate java codes and not cs. and .jspx ones and also it's so close to technology model. I believe we should be some more far from technology model. In EasyWeb we have graphical designer which has been developed by Microsoft DSL tools but I have changed the customary architecture of this tool because generating end applications needs some meta data which should be recommended or generated by some helping wizards. It has also a programming editor to develop the model in high level specified language, which has been developed by WebLang Code Gen.

2.9 Summarizing Terms

In this section we are going to summarize terms that we will use in the report as follows:

- *Meta-Meta Language*: a language that enables us to define domain specific language and in this case we will call the person who do it the *Mate-Meta Developer*.
- *Meta Language:* domain specific language which can model the domain with entities, properties, validations and etc. The person who does it is called *Meta Developer*
- *Domain Specific Model (DSM):* A system model or instantiation of DSL for modeling your system, some times we call DSM to *Domain Specific Modeling*. The person in charge of this modeling is *Designer* who will model the system and then generate the executable code.
- *Developer*: A person who has generated codes and he should develop the market ready version of application.
- *Meta-Data:* Data about entities of end application to describe how we can develop them. For example name, size, file growth and etc. can be meta data for a database.

Chapter 3

3 Methods

Regarding related works described in section 2.7, LTI motivated to have tools for developing web application supporting Software Factory to make it easy as much as possible. Our goal was laid on developing an environment to program web applications in high level approach however this environment had to be easy to learn, too.

3.1 EasyWeb

In EasyWeb the designer should be able to model the structure of his/her web application and then add some meta-data which describe the features of mentioned application. Finally EasyWeb will generate the codes of the end web application in .Net. The goal of EasyWeb isn't supporting the interface design because as we know Microsoft has very good and helpful IDE for developing the presentation layer of web applications. It focuses on code generating for operations and functions. EasyWeb is designed and developed for the first draft or as a tool for generating the prototypes in customer side. Considering the easy scenario of the web application development in this environment, if customer has a person who is familiar with basic concepts of web applications, we can count EasyWeb as a *User Development Environment.*

3.1.1 Requirements

Regarding to roadmap diagram presented in Figure 2, we designed a life cycle for developing web applications including the following phases:

- Developing tools and language for programmed web applications with high level approach.
- These tools should support architectural design to present the structure of web application and generate executable code which can be used as the first draft or prototype.
- Market Ready Development to deliver the product to market should be done by developer who is a professional in .Net and knows details about the developing a web application in this IDE.

Figure 19 tries to clarify the scope of EasyWeb as an architectural designer with the goal of code generation so it may be so different with normal procedure to design the architecture. In the case of architectural design in EasyWeb we should cover three phases:

Entity Design: Designer is asked to introduce the entities and their properties. EasyWeb should help the designer and simplify the property definition.

Business Design: To develop an executable prototype, designer should determine the business logic, also in this phase EasyWeb should help designer to program the business logic of application.

Executable Code Generation: Finally, EasyWeb should deliver the executable code after pervious activities.

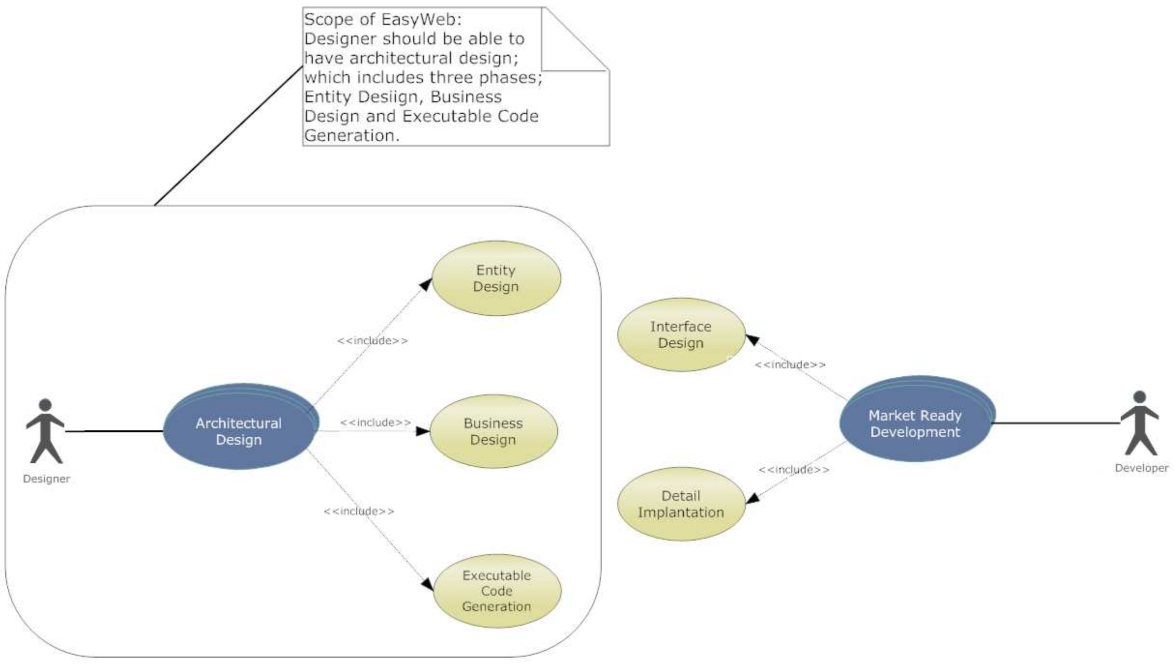


Figure 19. Scope of EasyWeb

To meet the mentioned goals and simplify the procedure for designers as much as possible we should design two possibilities to produce their system model which we call it DSM as we have defined in section 2.9:

Graphical Modeling: In this case EasyWeb will provide an environment with drag and drop tools defining entities and some wizards for determining properties of these entities and designing the business logic. Goal of these wizards is user assistance development with some offers in designing procedure.

Programming Modeling: EasyWeb also provides an editor for human-usable language which should be able to complement, keyword highlighting and other usual facilities for programming editors.

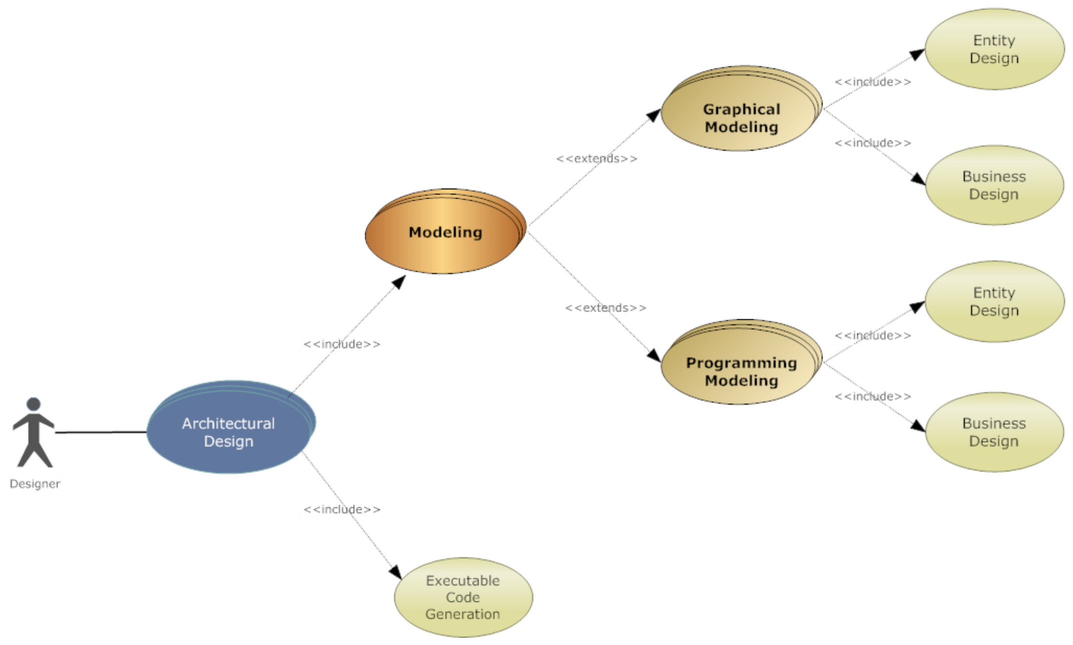


Figure 20. EasyWeb Detail Requirements

3.2 DSL Development Methodology

In developing a DSL we should answer the following questions as well as respecting to requirement engineering (E. K. A. V. Deursen 1998):

- Who will be writing the Domain Specific Descriptions (DSD)? What is the expected domain-specific background, and how much programming knowledge is required?
- How many DSDs will there be needed, and how long will they be? It may be possible to validate the correctness of three pages of DSL code, but who is going to predict the impact of a change in one out of 100 DSDs, each 25 pages long?
- Which (decidable) forms of static analysis and which integrity checks on DSDs are anticipated?
- What should happen if it turns out that the language requires new data types or new functionality?
- Does the DSL support user-definable syntax for, e.g., naming procedures? This may increase the readability, an important issue in DSLs, but it seriously complicates the construction of DSPs, including analysis tools that are needed during later maintenance phases.
- Is the main library written in the DSL or written in the target language? Who will be responsible for maintaining the library?
- Is the interface (data representation) to other systems easily adaptable or is it hidden inside the implementation of the DSL compiler?

- Who will be responsible for maintaining the DSPs? Is the knowledge about the domain sufficiently stable such that changes in the design of the DSL or the DSP are not to be expected?

So for answering these questions we should develop a DSL in three phases which have also several steps (P. K. A. V. Deursen 2000), (Cleaveland 1988), (E. K. A. V. Deursen 1998):

3.2.1 Analysis

This phase raise the important question how to recognize a domain, and how to determine the scope of a domain. Two definitions of *domain* could be used. The first is generally used in the artificial intelligence and object-oriented communities. It lets a domain correspond to the "real world", without direct relation to software systems it might be encoded in. The second definition comes from the systematic software reuse research community. It defines a domain as "a set of systems including common functionality in a specified area" (Simos 1995), but according to (E. K. A. V. Deursen 1998), most benefits in terms of reducing maintenance costs, however, are to be expected from the "domain as a set of systems" approach. Candidate domains should be

- Mature, i.e., a set of legacy systems exists
- Reasonably stable, i.e., certain aspects of these systems are satisfactory and worth studying
- Economically viable, i.e., new systems are anticipated in the domain.

The pieces of functionality in the legacy systems will help to identify the domain. Furthermore, modification requests from the past, or differences between the various systems, will help to identify the variability in the domain. The DSL should then be designed such that it is expressive over this variability (Simos 1995).

Hence, we consider following steps for this phase (P. K. A. V. Deursen 2000):

- Identify the problem domain.
- Gather all relevant knowledge in this domain.
- Cluster this knowledge in a handful of semantic notions and operations on them.
- Design a DSL that concisely describes applications in the domain.

In addition to clarifying the concept of Domain Analysis we should say it means examining needs and requirements of a collection of systems which seem "similar" (P. (R. N. Taylor 1995), (K. A. V. Deursen 2000).

3.2.2 Implementation

After domain analysis we should implement the obtain knowledge about the mentioned and chosen environment, in this phase we have these steps to achieving the goal which is a Domain Specific Language:

- Construct a library that implements the semantic notions (P. K. A. V. Deursen 2000).

- Design and implement a compiler that translates DSL programs to a sequence of library calls (P. K. A. V. Deursen 2000).

According (P. K. A. V. Deursen 2000), implementation phase can be done in four ways:

- *Interpretation or compilation*: This is the classical approach to implementing a new language. Standard compiler tools (P. K. A. V. Deursen 2000), (A.V. Aho 1986), (Bentley 1986)
- *Embedded languages / domain specific libraries*: In this approach, existing mechanisms such as definitions for functions or operators with user-defined syntax are used to build a library of domain-specific operations. The syntactic mechanisms of the base language are used to express the idiom of the domain .
- *Preprocessing or macro processing*: In this approach the new constructs are translated to statements in the base language by a preprocessor.
- *Extensible compiler or interpreter*: This approach is similar to the previous one, but the preprocessing phase is now integrated in the compiler.

To compare these approaches, we should consider that the main advantage of building a compiler or interpreter is that the implementation is completely tailored towards the DSL and no concessions are necessary regarding notation, primitives and the like. Also, error detection, static analysis, and optimizations can be done at the domain level. Clearly, an important problem is the cost of building such a compiler or interpreter from scratch, and the lack of reuse from other (DSL) implementations, although some DSL tool sets (for example Microsoft DSL Tools) are particularly designed to overcome such problems. An advantage of the second approach is that the compiler or interpreter of the base language is reused *as is* for the DSL. The main limitation is in the expressiveness of the syntactic mechanisms in the base language. In many cases, the optimal domain-specific notation has to be compromised to fit the limitations of the base language. The main advantage of third one is simplicity. Its main disadvantage is that static checking and optimization are not done at the domain level. Consequently, generated code is error prone, and the user is provided with feedback on these errors at the level of the base language, or only at run-time. The advantage of the last approach is that more type checking and better optimization is possible.

3.2.3 Test

In this phase, we want to be sure about the performance of our design and implemented language, such as other softwares we should write at least one example for all desired domain and recognized application types.

3.3 Customary Architecture for Modeling Environments in Microsoft Visual Studio

Modeling environments such as EasyWeb Graphical Designer have a special architecture supported with *Microsoft Visual Studio Modeling Platform* and *Microsoft DSL Tools* which we have described in section 2.6.1. In other hand to develop a graphical designer in Visual Studio you should follow the architecture presented in Figure 21.

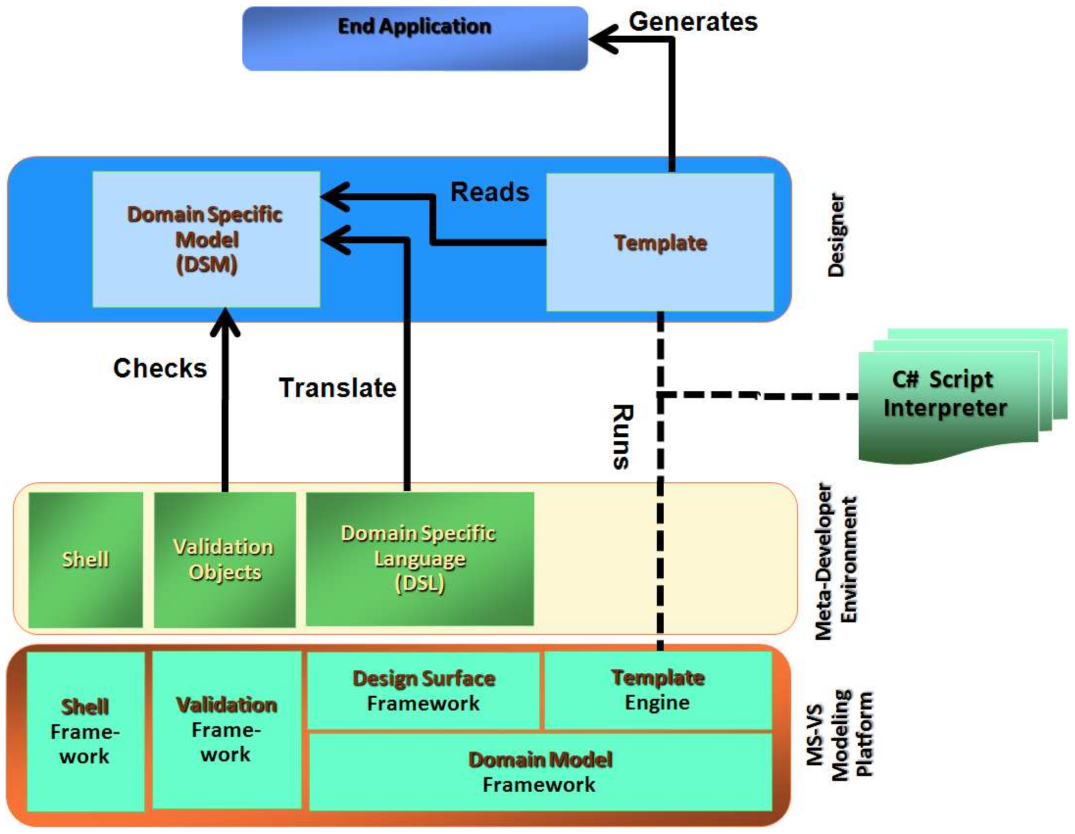


Figure 21. Customary Architecture of Microsoft DSL Tools

As you can see in Figure 21, A Graphical Designer when it was developed in MS-VS has three separated layers as follows:

- *MS-VS Modeling Platform:* Libraries that Microsoft DSL Tools uses for generating these kinds of designers. This layer consists of four frameworks and one engine described in section 2.6.1.
- *Meta Developer Environment(Microsoft DSL Tools):* These tools that lie on the Modeling Platform is made up from three components:
  - *Shell:* Instance of Shell framework and it's used for integrity of components.
  - *Validation Objects:* Instances of Validation Framework, and they are used to validate your model and its entities, you should consider that here you will write your business logic for validating objects. In Microsoft DSL Tools if you need some business logic to add your graphical designer to make it more dynamic this is a trick that you can use for adding codes inside of MS-VS DSL Tools.
  - *Domain Specific Language (DSL):* Microsoft DSL tools provide a meta-meta language for defining a DSL. But this language has a graphical interface and you can use the drag and drop toolbox to develop your language.

- *Graphical Designer Used by Developer:* This is exactly the product that we expected. Designer can use the toolbox produced by DSL tools and dependent on defined language. In other hand in this tool box you will have customized tools for entities defined in DSL. Designer should define following items:

  - *Domain Specific Model (DSM):* Here, using graphical designer modeling the system is possible, in fact designer uses the drag and drop toolbox, and then sets the properties of each entity.
  - *Template:* As we have described in section 2.6.1, as template is used for generating codes, so here in produced graphical designer we also need template to generate end application from our system model. Therefore designer should write his/her expected codes to generate in this tool. Template works as an interpreter of your system model and for each entity type it should generate different kinds of code.

3.3.1 Microsoft DSL Tools: Worthy to Catch on?

After using DSL tools for a while, I'm not completely convinced that they are really right tools and worthy to use. For a code generation environment to catch on it has to be significantly easier than the task it replaces. At the moment DSL Tools are difficult to use for designers who are not in that much professional that I expect them to write their codes in a not user assistance environment and without some of data types and in other side visual programming IDEs such as .Net are fairly easy to use. Therefore a rational decision for software companies between using Microsoft DSL tools and writing its codes in template or using visual programming IDEs would be expected to be the second choice. However this comment can't make doubt or hesitation on Software Factory Theory truth, I should consider that Microsoft DSL tools are not the right ones for this goal. They have been designed for developing product line and not end applications, so if we want to use them we should change their customary architecture.

3.4 EasyWeb Graphical Designer

To develop EasyWeb Graphical Designer, we should use advices explained in section 3.3.1, so I have planned to change customary architecture of MS-VS Modeling Environments.

Figure 22 shows how EasyWeb changed this architecture. Using EasyWeb Graphical Designer web application generation will be done in three phases:

- *Step 1: System Modeling:* In this phase designer models his/her system, and as we said it can be done with drag and drop and also following wizards and windows forms.

  In fact, generating code for an end application needs more than entities name and we should have some data about their properties, called metadata. EasyWeb gets them in Wizards and then stores in database.

Getting data in Wizards is accomplished by model checking and validation, this ability is possible because validation objects have access to definition of language (DSL) and also have access to metadata.

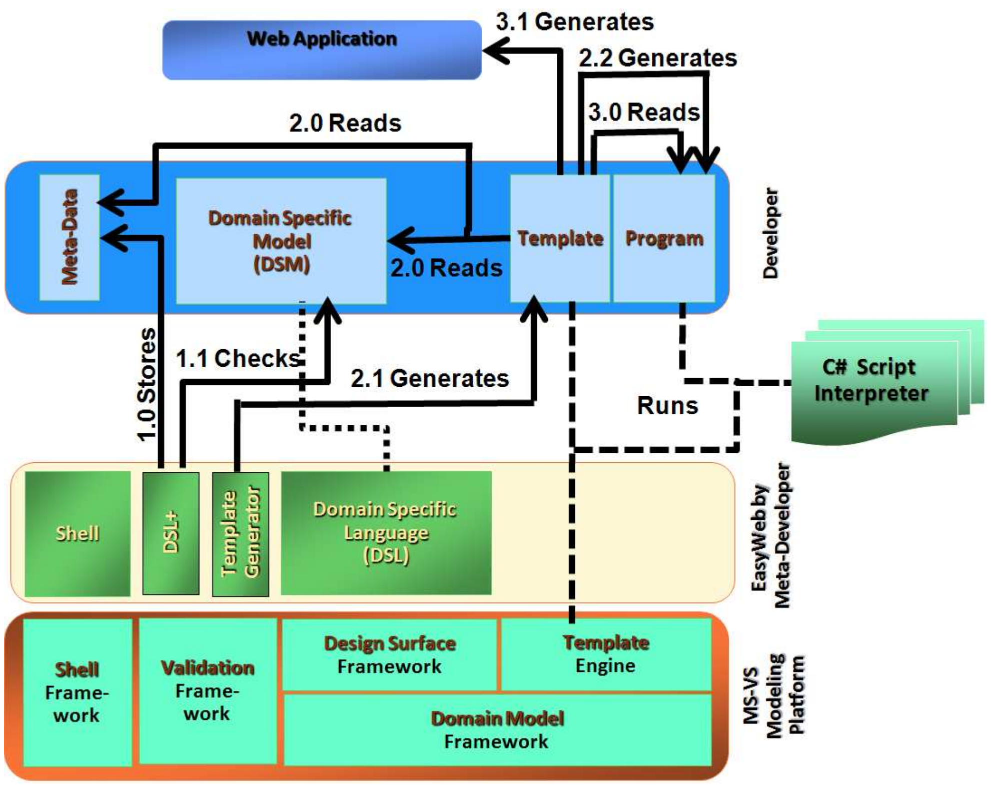
These two activities will be done with a new component that I have added to Microsoft DSL tools and we call it *DSL+*. As far as we have explain we should add all of our code in validation framework so we have developed this component which is made of some windows forms as a validation object.

- *Step 2: Intermediate Code Generation:* we have an intermediate code that we read the model from DSM and Meta-Data then generate *Program* which includes all data to generate end web application.

  Program is written in a machine-enable language that we don't need to compile and just read as a text file because it was generated automatically and we are sure that it's correct, in addition it is a little bit same as xml and uses the same idea (entities and properties).

  First of all we will read DSM and Meta-Data by *Template Generator* (It's also a validation object) and then generate template. First part of this template is related to generating Program and the second part is related to reading the Program and generating end web application. There is clear question about the reason for generating Program and then read it, let us to explain that this file can be used for generating end application even the entry is not graphical designer. We mean you may change this file in Programming Editor and then generate code.

  Step 3: Web Application Generation: In this step we will read Program and then generate the executable code. As we said, we don't need to compile it because we are sure that it's correct and we have always entities and properties same as xml.



Step 1. System Modeling
Step 2. Intermediate Code Generation
Step 3. Web Application Generation

Figure 22. EasyWeb Graphical Designer Architecture

3.5 EasyWeb Programming Editor

I have designed another interface to model systems, for who likes programming instead of graphical modeling. In this editor, designers model their systems with a human-usable language and completion, keyword highlighting and user-assist facilities.

Developing this editor can be done by WebLang Codegen in two steps:

- *Step1: Defining the Language:* we should introduce the domain for which you want to define specific language (in this case, it is web application). The domain definition is important and will directly influence the scope of our language. In this step we will introduce our keywords, language structure and etc.
- *Step 2: Generating Program:* To generate Program we should use *WebLang Codegen Template*. In this tool, templates can be defined for each module, submodule and global singleton. The BNF syntax for template definition is available and we can use it.

When we have Program, as I have develop its engine in graphical designer we can read it and generate end application.

Chapter 4

4 Implementation

This chapter covers important details of the performed implementation, including descriptions of languages and how they have been developed and also brief details of added objects in mentioned customary architecture, explained in section 3.3. We will try to clarify implementation limitations that we have faced during the doing this work, which can be useful for persons who want to continue this idea.

4.1 Languages

EasyWeb as we have described uses two languages; Intermediate language and human-usable one. These two languages, actually use the same concept and definition but in different presentation. In other hand they use the same entities and relations but because of two goal that they follow, we have different presentation; one should be used for machine must be readable line by line for getting entities and properties (the idea is the same as xml) and one should be used by designers must be much easier and understandable. An important point that we should consider is that we have to develop a user assist enable editor for the second one.

4.1.1 Domain Model for Web Applications

A model which wants to describe a domain of web application is a model consists of all entity and relationships that any application in this domain conducts their business. This model for all of them is the same. But actually different people have different ideas about a domain (such as web applications), but it could be raised from different concepts when they look at the web application topic in detail or for more clear words these drive from how the model is used in the context of particular families of methodologies.

Here, we have defined a domain model for web application in the context of producing system model in high level approach without taking time on details.

Figure 23 tries to clarify our domain model of web applications. Actually a web application is consists of *Databases* and *Web Pages* and a database can include *Tables* which have *Columns* for itself. Then a Web application may be built of *Links*, also an *Initialization*, *Operations and an Output*. An initialization is a component that initializes web page with some data from database. For example when we want to present editable personal information of a specified person we can use initialization. Initializations are will be coded in the load event of page and it can be consists of query statement of *SQL*. Operations present submit buttons in web pages so each operation needs its variables which can be submitted by pervious page or presented in current page as input it also can described *Business Logic* for non-query tasks on database and some codes. Operation's business logics will be coded in click event of submit button. Output is a component to present a report of chosen fields in database with delete, edit and select facilities.

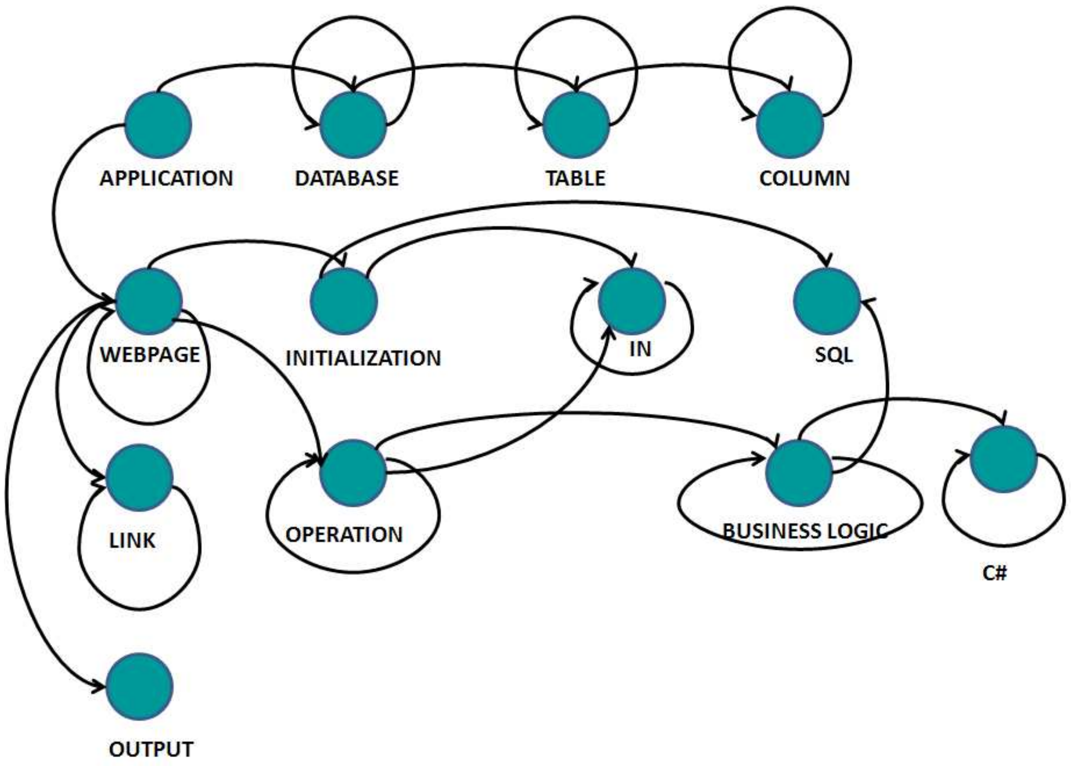


Figure 23. Conceptual Diagram for Languages used in EasyWeb

4.1.2 Intermediate Language

We will transfer our system model in graphical designer and even in programming editor to a program with in the Intermediate Language as we can see it's totally for machine to read entities and properties all together. This language should be easy to read for machine so it could be nice if we can present data line by line. We don't need any editor or syntax error checking because it will be generated by machine automatically.

Table 1 shows keywords used in this language and their concepts, as you can see we have twenty five keywords and because of easy readability for machines we always close an statement with different end-clause.

Table 1. Keywords in Intermediate Language

Keyword	**Concept**	**Keyword**	**Concept**
APP	Application	SQL	sql statement
END_APP	end of application	END_SQL	end of sql statement

DB	Database	BL	business logic
END_DB	end of database	END_BL	end of business logic
TABLE	Table	OP	operation
END_TABLE	end of table	END_OP	end of operation
COLUMN	Column	OUT	output
WEBPAGE	Webpage	END_OUT	end of output
END_WEBPAGE	end of webpage	WHERE	condition for output
PRE	Initialization	DEL	delete button for output
END_PRE	end of initialization	EDIT	edit button for output
IN	Input	SEL	select button for output
COND	condition for doing business logic		

In addition we have two operators; ^ for values passed to page from pervious page and @ for values can be accessed from inputs in the page. Keywords plus to these two operators are reserved words in this language.

4.1.3 Modeling Language

For who believe that coding is much easier than graphics, we have designed a human-usable language, which can be translated to intermediate one. Keywords of this language is presented in Table 2.

Table 2. Keywords of Modeling Language

Keyword	Concept	Keyword	Concept
Application	Application	title	title of webpage
Database	Database	dir	ltr or rtl for direction of page
Table	Table	bgColor	Background color
webpage	Webpage	edit	edit button of output
link	Link	del	delete button of output
initialization	Initialization	sel	sel button of output
input	Input	where	condition of output
output	Output	server	server of database
operation	Operation	typeOfFileGrowth	type of file growth in database
sql	Sql	fileGrowth	file growth in database

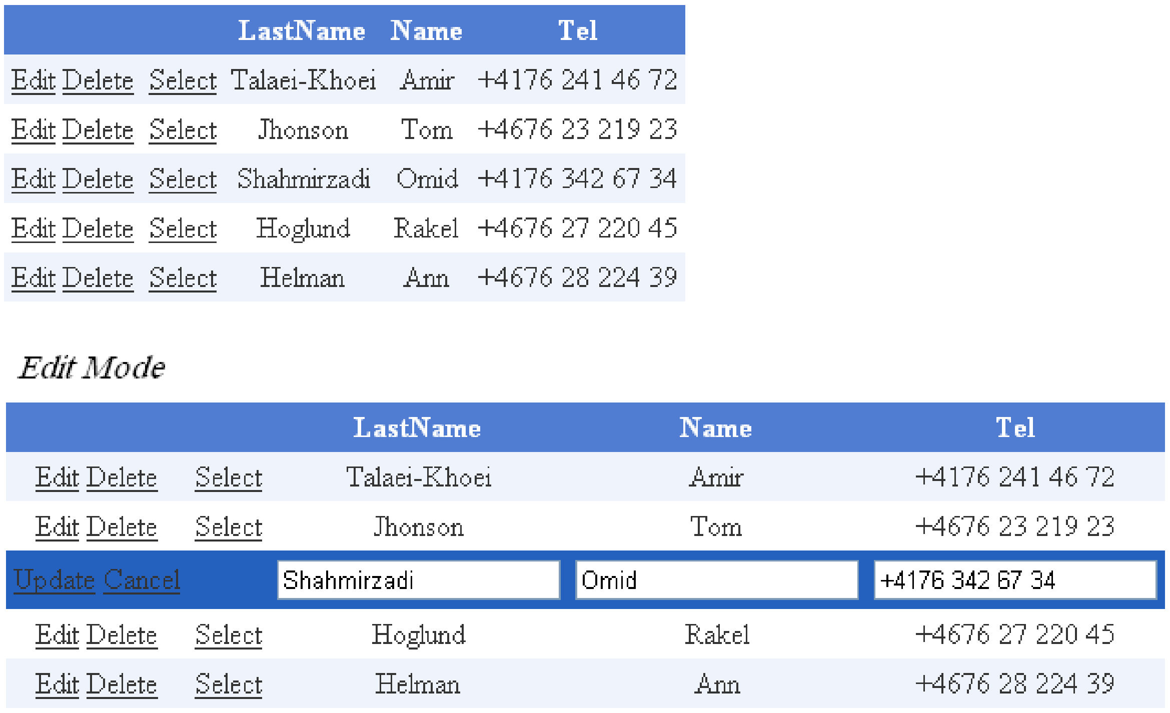
c#	c# code	Size	size of database
logic	business logic	source	source of data in output: server_name.database_name
sqlString	sql code	condition	condition of doing business logic
destionationPage	page that server should be transfer		

The same as intermediate language we have @ and ^ operation to access values. they plus keywords make the reserved words set.

The use of our human-usable language provides freely property determination of technologies specified as entity. We present entities in a format same as class which can be inline or not and we also use functions to getting some properties. I have tried to change intermediate language for normal GPLs such as C++ or Java.

4.1.4 Simple Example

Figure 1 present a simple web application which needs a report from three fields in database with edit, delete and select facilities.



	LastName	Name	Tel
Edit Delete Select	Talaei-Khoei	Amir	+4176 241 46 72
Edit Delete Select	Jhonson	Tom	+4676 23 219 23
Edit Delete Select	Shahmirzadi	Omid	+4176 342 67 34
Edit Delete Select	Hoglund	Rakel	+4676 27 220 45
Edit Delete Select	Helman	Ann	+4676 28 224 39

Edit Mode

	LastName	Name	Tel
Edit Delete Select	Talaei-Khoei	Amir	+4176 241 46 72
Edit Delete Select	Jhonson	Tom	+4676 23 219 23
Update Cancel	Shahmirzadi	Omid	+4176 342 67 34
Edit Delete Select	Hoglund	Rakel	+4676 27 220 45
Edit Delete Select	Helman	Ann	+4676 28 224 39

Figure 24. A Simple Web Application

This application is built by three object; Persons, Report and ListOfPersons which are entity, page and output respectively. Figure 25 show the abstract model of this example.

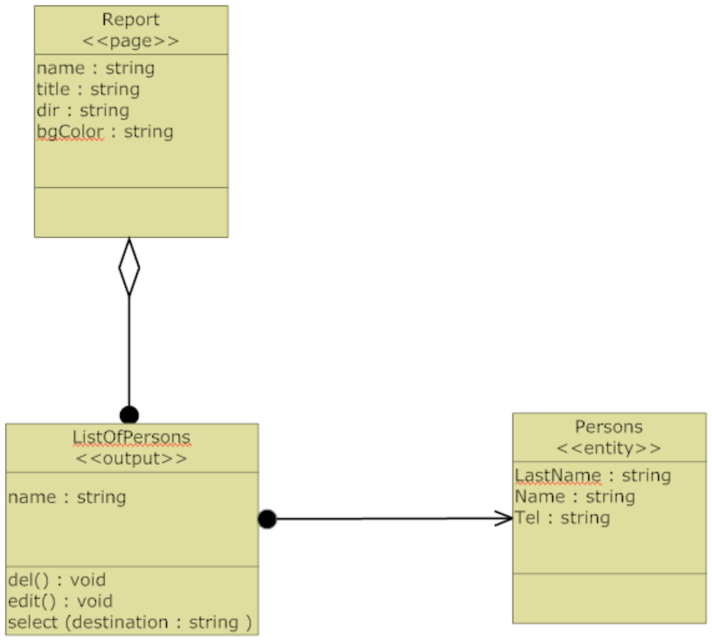
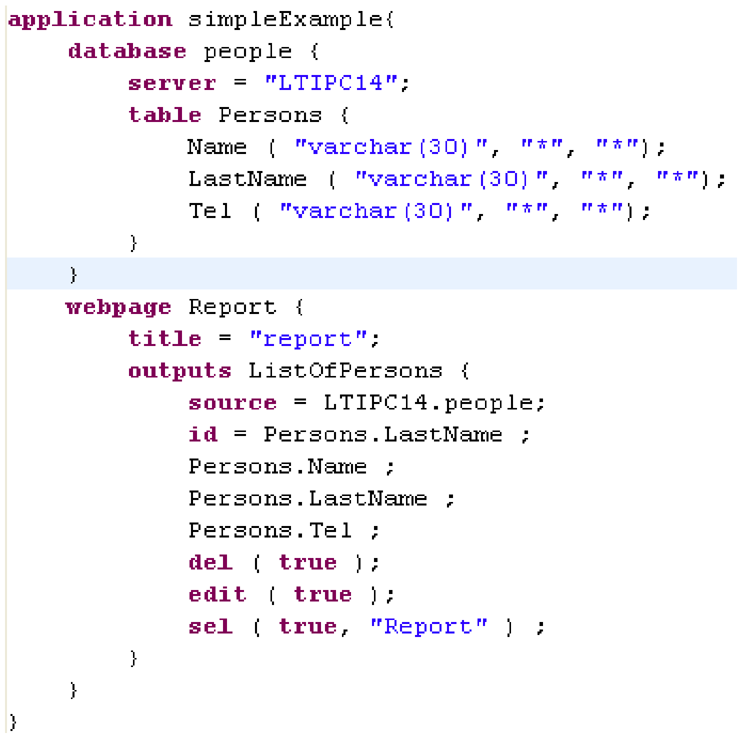


Figure 25. Abstract Model of Simple Example

Figure 26 tries to show how we can model the simple example presented in pervious figures in our human-usable language. Please consider we can skip some information such as bgColor, dir and etc.



```
application simpleExample{
    database people {
        server = "LTIPC14";
        table Persons {
            Name ( "varchar(30)", "*", "*");
            LastName ( "varchar(30)", "*", "*");
            Tel ( "varchar(30)", "*", "*");
        }
    }
    webpage Report {
        title = "report";
        outputs ListOfPersons {
            source = LTIPC14.people;
            id = Persons.LastName ;
            Persons.Name ;
            Persons.LastName ;
            Persons.Tel ;
            del ( true );
            edit ( true );
            sel ( true, "Report" ) ;
        }
    }
}
```

Figure 26. System Model in Programming Editor for a simple example

Then after compilation of this code we can have *Program.txt* as our intermediate code. Code generated by our editor in intermediate language is presented in.

```
APP EASYWEB_simpleExample
DB people LTIPC14 % 10 UNRestricted
TABLE Persons
COlUMN Name varchar(30) * *
COlUMN LastName varchar(30) * *
COlUMN Tel varchar(30) * *
END_TABLE
END_DB
WEBPAGE Report report ltr #FFFFFF
OUT ListOfPersons people.LTIPC14
Persons.LastName *
Persons.Name
Persons.LastName
Persons.Tel
DEL
EDIT
SEL Report
END_OUT
END_WEBPAGE
END_APP
```

Figure 27. Intermediate Code for Simple Example

4.2 Implementation of Graphical Designer

Graphical designer consists of several components that some of them will be also used for Programming Editor. First of all we should define our domain and then add *DSL+* and *Template Generator* to Microsoft DSL tools.

4.2.1 Domain modeling in Microsoft DSL Tools

As we have explained before, first of all we should introduce and define our domain with a graphical tool in Microsoft DSL to model the domain of web applications (Figure 28). As you can see we have two kinds of objects; one for domain entities (Classes and Relationships in the figure) and one for tool (Diagram Elements in the figure) in graphical interface we should connect our tools to related entities in a our domain model.

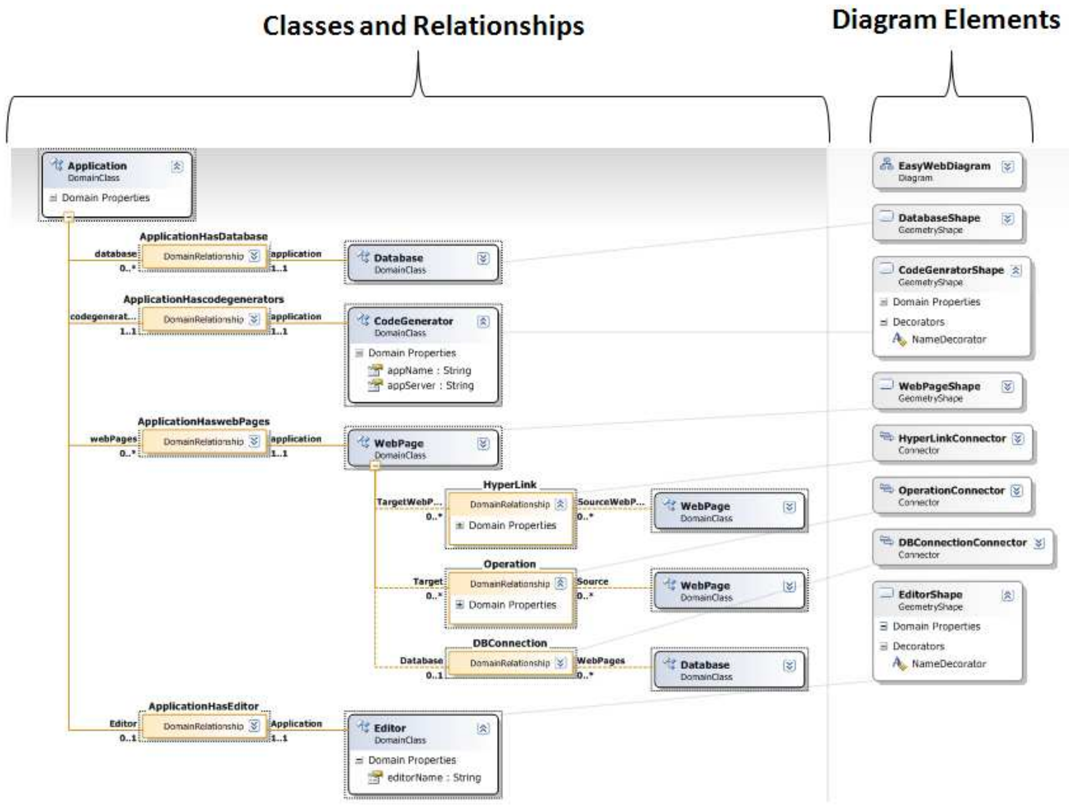


Figure 28. Domain Modeling in Microsoft DSL Tools for EasyWeb

4.2.2 Implementation of DSL+

For developing DSL+ (some Wizards to get metadata) I should develop validation objects. These objects that have been defined for database and operation open a windows form and with an easy to use procedure helps designer to define business logic and properties of entities.

We also have two entities for editor and CodeGenerator. With editor tool we can open our programming editor to program our model in designed editor, which is not related to graphical designer and with CodeGenerator we should define a *Template Generator*.

4.2.3 Implementation of Template Generator

Template Generator made up by three components:

- *Interface*: this component first asks from designer to select his/her .tt file that he/she wants to generate template inside (Figure 29) by a .net open dialog.

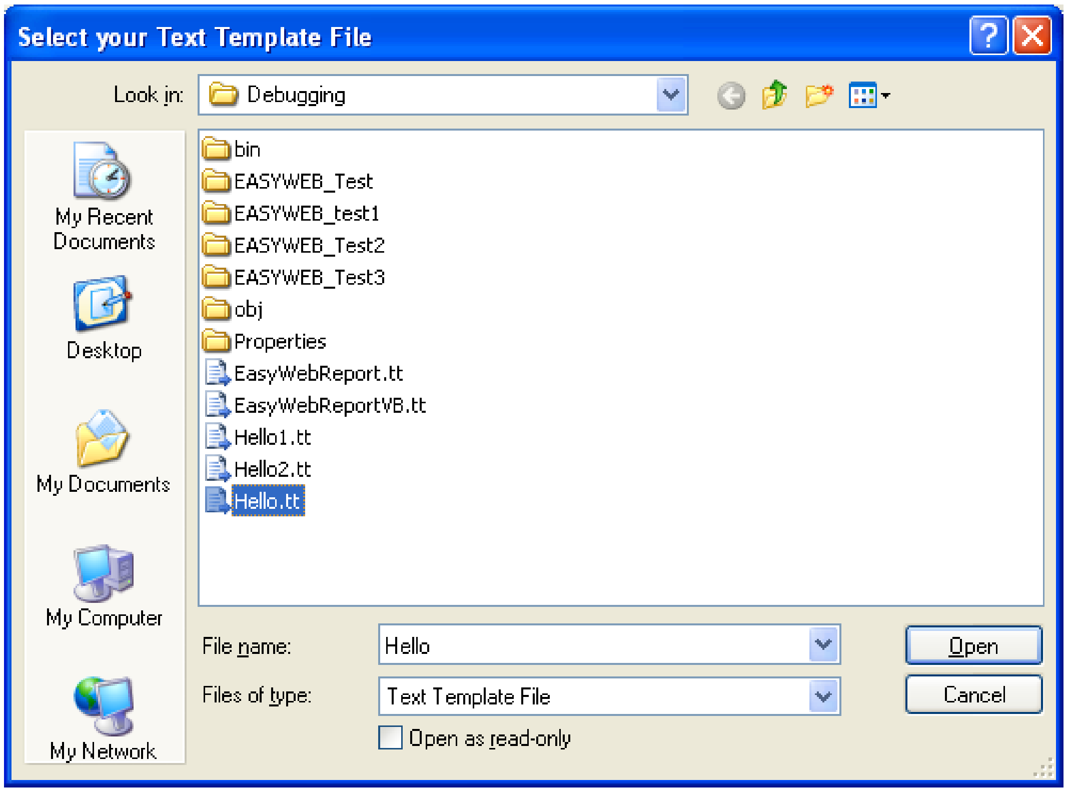


Figure 29. A Dialog for Selecting a .tt file to generate Template inside.

After selecting a .tt file, Interface component asks for mode of generation; the first item is related to programming mode and the second one is for graphical designing (Figure 30).

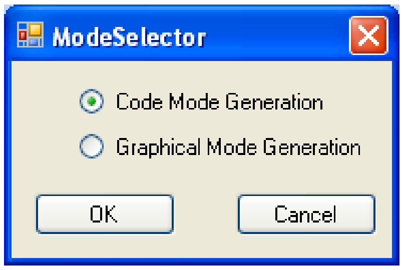


Figure 30. Choosing Mode for Generating Template

- *Program Generator*: template generator after getting mode call *Generator Class* and pass a parameter to call *programGenerate()* method or not. This method is used when you select Graphical mode and generate a code in mentioned .tt file to read system model and metadata for generating *Program.txt*. Structure that we use in this component is figured out in the following pattern:

```
//Reading System model and metadata
templateGeneratorObject.WriteLine(ProgramWriterObject.WriteLine(programCode));
```

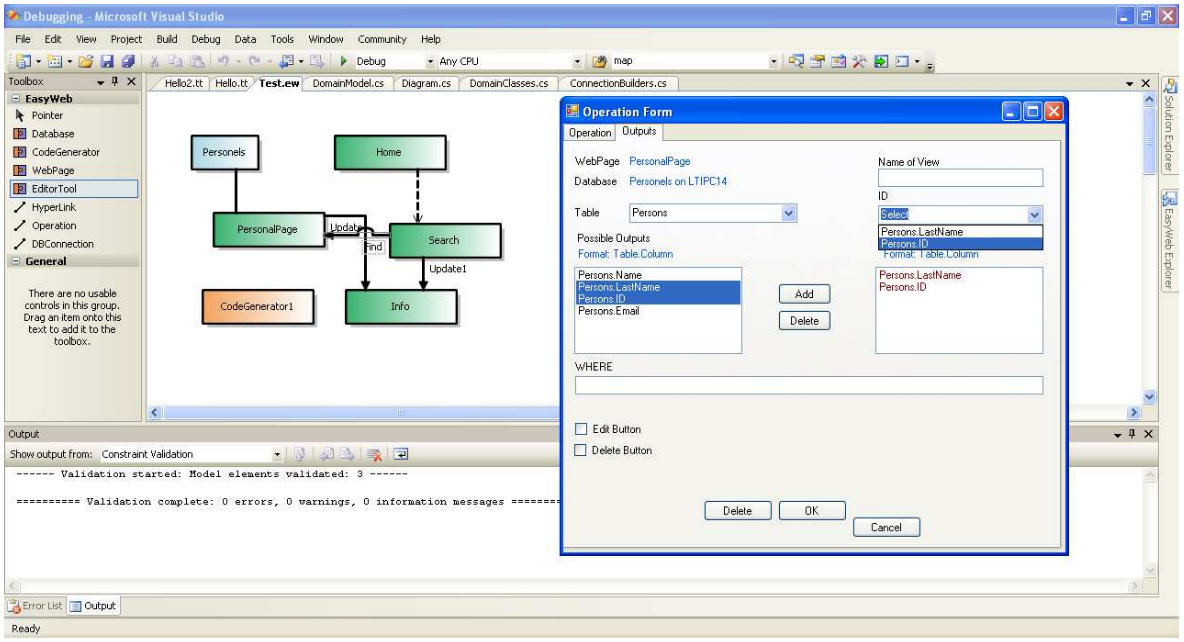
- *Interpreter Generator*: Interpreter is not only to read the Program but it also used for generation codes. Code generation means generating a string inside of *Write* or *WriteLine* functions in the text template which can write in a .cs or .aspx file as follows:

  WriteLine (writerObject.WriteLine(C# or ASP code));

  So we should generate in .tt file a switch-case inside of each item; there is an ability to generate some string related to what has been read. Thus we should interpret the Program and then generate some codes inside of WriteLine(). We will explain interpreter in section 4.4. But here we generate the second part of .tt file that includes the interpreter. So we should use the following structure:

  templateGeneratorObject.WriteLine(interpreterCode);

Finally, with running Microsoft DSL tools we have a graphical environment which can also help designers using wizards ().



WebPage PersonalPage
Database Personels on LTIPC14
Table Persons
Name of View
ID Select
Persons.LastName
Persons.ID
Possible Outputs
Format: Table.Column
Persons.Name
Persons.LastName
Persons.ID
Persons.Email
Persons.LastName
Persons.ID
WHERE

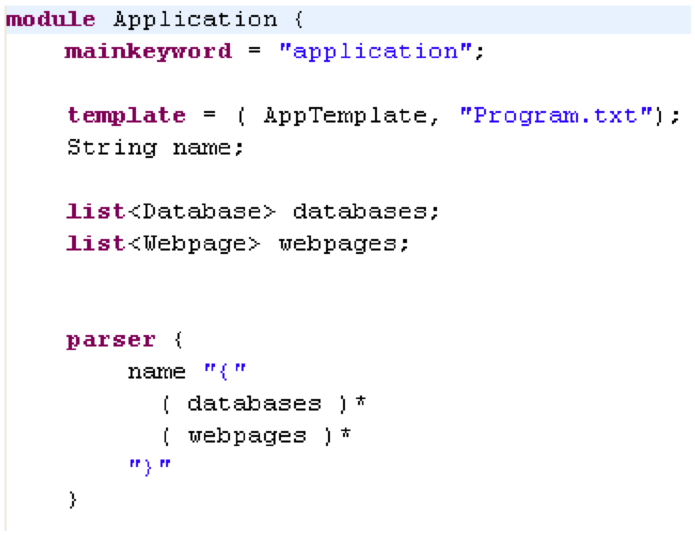
```
------ Validation started: Model elements validated: 3 ------

========== Validation complete: 0 errors, 0 warnings, 0 information messages =======
```

Figure 31. EasyWeb Graphical Designer

4.3 Implementation of Programming Editor

Programming Editor has been written on the WebLang Codegen, and with a code base definition of language. Then with compiling this definition we had an editor with complementing and keyword highlighting features. But we should get objects from the compiled programmed in editor and write an intermediate code in Program.txt.

Figure 32 shows the definition of application that it consists of databases and webpages as you can see we can define 0-n database and 0-n webpages and syntax is highly human preferable following such Java or C++ style. In this definition you have only one module and inside of our module we have our submodules. And for each module or submodule we have to define a parser to introduce our style. Please consider that ( )*, ( )+ and [ ] mean repeatability 0-n, 1-n and 0-1 respectively.

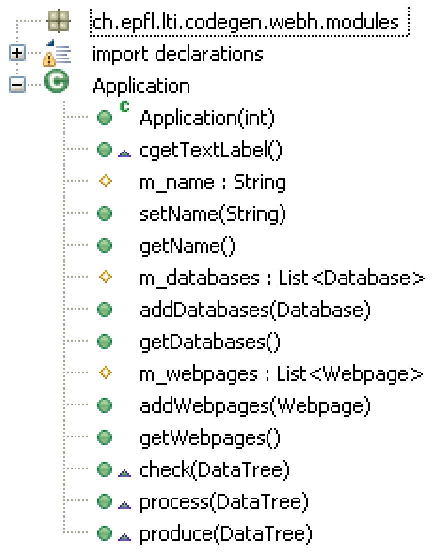


```
module Application {
    mainkeyword = "application";

    template = ( AppTemplate, "Program.txt");
    String name;

    list<Database> databases;
    list<Webpage> webpages;


    parser {
        name "{"
           ( databases )*
           ( webpages )*
        "}"
    }
```

Figure 32. Definition of application that it consists of databases and webpages

Then compiling this file generate all of our submodules in a java file, we can define *check ()* in each one for checking the entry of designer and some concentrates. in this java file you can find set and get methods for properties that you have defined in your module or sub-module.

For example Figure 33 shows all methods that have been generated from Application module. You can find *check()* or *get* and *set* for properties that you have defined. so you can have access to values defined by designers in editor.



```
ch.epfl.lti.codegen.webh.modules
import declarations
Application
    Application(int)
    cgetTextLabel()
    m_name : String
    setName(String)
    getName()
    m_databases : List<Database>
    addDatabases(Database)
    getDatabases()
    m_webpages : List<Webpage>
    addWebpages(Webpage)
    getWebpages()
    check(DataTree)
    process(DataTree)
    produce(DataTree)
```

Figure 33. Methods generated for module Application in Codegen

The next step is getting data from editor as we have described and writing them in Program.txt for generating codes WebLang Codegen provides a template and you can generate your code much easier than Microsoft DSL without Write or WriteLine and just with <%= %> for dynamic entries and for other one you can just write them. To add some dynamic processes on entries in editor you should use <% %> these dynamic processes can be a for-loop or some things such as that. For example for generating APP for application entry I have used following code:

```
APP EASYWEB_<%= application.getName() %>
```

and for tracing all databases I have also used a for loop as follows:

```
<% for (int i = 0 ; i <application.getDatabases().size() ; i++){ %>
```

Finally, to have an editor you should just set a file in a defined java project to run considering codegen.

4.4 Implementation of Interpreter

Interpreter as we have discussed is not only an interpreter but it also is a codegen to read Program line by line and sensitively to keywords generate codes to write in end application .cs and .aspx files.

4.4.1 Structure

Functionality of interpreter can be done a loop to read the Program line by line and check the keywords to how many properties it should be expected in this line, to achieve this ability I use a variable which shows the status of reader and current keyword, then regarding this keyword and read properties, we should generate different codes. Generally, its structure is as follows:

```
while ( ( line = readline()) != null ){
        switch (line){
                case [KEYWORD]:
                        //reading related properties
                        //generation of WriteLine (writeObject.WriteLine(C# or ASP code))
                        break;
                …
        }
}
```

4.4.2 Overview of Datatypes and Generated Codes

Components used in web applications are more or less can be modeled by inner classes. In other hand a web application in EasyWeb is included some inner objects. Here we will describe datatypes used in EasyWeb and what we generate in interpreter.

4.4.2.1 Application

This datatype introduces a web application including databases and webpages. You can have just one instant of this type and in fact this object makes some initialization of our variables.

4.4.2.2 Database

Each web application can have one or more database. Databases include tables and generate a webpage that if you run it on server and click the button you can create database. In the clicking event of this button you will have “CREATE DATABASE” statement for mentioned database.

4.4.2.3 Table

Tables are located in databases and they includes with columns. Each columns has name, type, nullability (NULL or *) and id (* or name of coulmn). We will generate “CREATE TABLE” statement in the clicking event.

4.4.2.4 Webpage

Each application can also have one or more webpage and it can be included by outputs, initialization and operation. This datatype makes creating .cs and .aspx files and writing primary includes and headers. Please note that when we use webpage term we mean a webform.

4.4.2.5 Output

A powerful datatype to present a report from mentioned fields of database that and ability to *delete*, *edit* and *select*. An object of this datatype makes <asp:SqlDataSource> and <asp:GridView> .

4.4.2.6 Initialization

This datatype may be used for filling inputs in the webpage. Initialization is a query based sql that it will be generated in the load event of webpage. So it could have a sql object.

4.4.2.7 Operation

Each webpage may have one or more Operations and they will be presented in end application by submit button. In fact, an operation is a non-query sql and c# code run in the clicking event of the button. Each operation includes sql and c# objects.

4.4.2.8 Sql

Both initialization and operation can have sql objects which present a server, database and sqlstring. We should generate a code that opens a connection and do the sqlstring then closes it.

4.4.2.9 C#

This is freely c# code that can be added before and after sql object in an operation to run in the clicking event of related submit button.

Chapter 5

5 Analysis

This chapter enumerates the validation what we have done to develop EasyWeb and also it covers the evaluation of EasyWeb in the aspect of a MDD Environment and quality of web applications can be developed with this tool.

5.1 Validation

Here we are going to explain our roadmap mathematically, and why we need to define an interpreter. Our method to explain is browed from (A. Moss 2005).

According to Equation 1, if we have a language *D* and its interpreter to transform it to tool language (*T*) then you can run the interpreter on a program which has been written in *D* (P_D) and obtain a program written in T (P_T).

$$\forall i: [[spec](int_T^D, P_D)] = P_T \qquad \textbf{Equation 1}$$

But composition of two interpreters means running an interpreter on definition of another language, for example according Equation 2, if we have an end application language (*M*) and run interpreter of programs written in tool language to *M* (int_M^T) on the definition of int_T^D then in opposite of usual mistake we will not get int_M^D because we have run it on an interpreter definition and not on a program written in *T* so we will obtain$int_{M'}^D$.

$$[spec_M]\ (\ int_M^T, int_T^D\ ) = \ int_{M'}^D \qquad \textbf{Equation 2}$$

Then when we run $int_{M'}^D$ on a program written in language *D* (P_D), we will get a language written in M' (Equation 3)

$$\forall i: \left[[spec_M]\left(int_{M'}^D, P_D\right)\right](\ i\ ) = P_{M'} \qquad \textbf{Equation 3}$$

The next step is designing an interpreter with ability to transfer programs written in M' to *M* one and run it on $P_{M'}$.

$$\forall i: \left[[spec_M]\left(int_M^{M'}, P_{M'}\right)\right](\ i\ ) = P_M \qquad \textbf{Equation 4}$$

So if we let $int_M^{M'}$ as switch-case interpreter our cycle is exactly follows to what we have showed mathematically.

5.2 Evaluation Method

In this section we will describe, some criteria that we can evaluate this system in compression of the mentioned environments in pervious sections. But actually, because of lack of batch data I didn’t do it and here, I will just explain some reasonable points that we can expect.

5.2.1 Using as a Requirement Engineering tool

Regarding Figure 34, current code generators are highly effective on implementation, test, deploy and maintenance but they don’t seem they work well for analysis/Design and specially for *Requirement Engineering* as we have explained in section 1.2 we have designed EasyWeb as prototyping tool, to fill this gap, and I believe with skipping interface design and forget to generate a market ready application we can simplify producing system model and use it as a tool for requirement engineering. EasyWeb is totally usable for a person who has only knows basic concept of web applications and finding these kind of professions in customer side is not always far from reality, so if you can find this amount of knowledge you can count EasyWeb as tool for user development environment, which can be effectively decreases the effort in requirement engineering phase. Both WebRB and AndroMDA because of idea to cover every things are so complex to use in compression of EasyWeb and it takes much more time to design a system model.

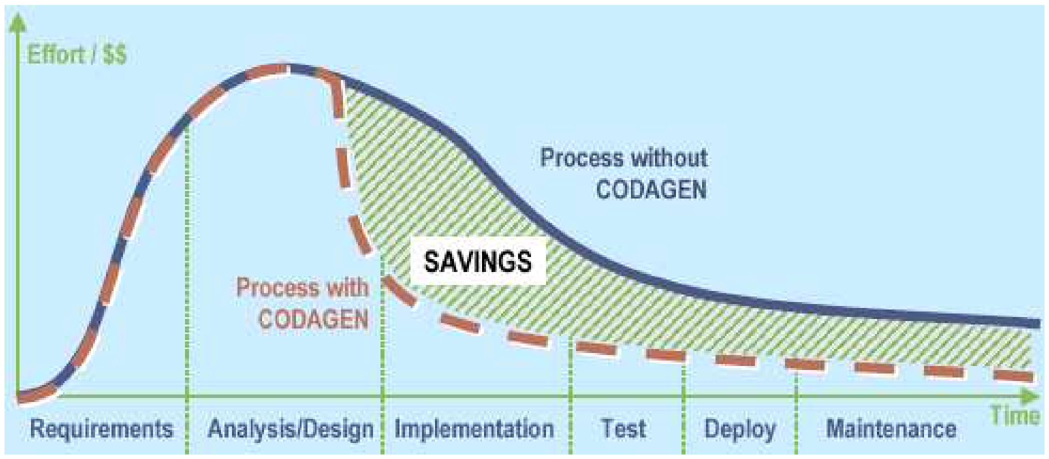


Figure 34. MDD Saving and Benefits (Siegel 2005)

5.2.2 Computational Model

EasyWeb tries to use of a very close to reality computational model, and easy to map from which in designers’ mind and what they want to develop, for example when a designer wants to add a functionality to a webpage, it’s easy to map to an operation, but in this case WebRB model of computation closely following the relational concept of web application which is not in that much easy to understand.

5.2.3 Technological Model Independency

EasyWeb tries to be independent from the technology as much as possible but in this deal we face to a trade off between complexity of modeling and technological dependency, for example when you want to use a sql statement in some occasions, the easiest way is getting as a sql code, or in the case of business logic if we wanted to be independent from the technology and target language we had to use some ideas same as WebRB, and increase the complexity of system modeling. Technology dependency in EasyWeb for sure is more than WebRB or AndroMDA, but WebLang uses the same idea as EasyWeb.

5.2.4 Graphical Designer

However EasyWeb has a graphical designer but I believe that wizard modeling is much more difficult than using a good and user assistance enable programming editor. To use graphical designer we should consider to following items:

- Using wizards is too limited.
- Wizards usually are not good enough because designing habits are so different.
- Maintenance is not so confusable, but in programming editor it may be possible just with a copy and paste, or deleting a one line of code.
- Using wizards decreases the design flexibility.

In the term of Graphical Designer WebLang is totally on a human-usable language but WebRB and AndroMDA try to use graphical interface for modeling systems, and you can feel the complexity in their design process in compersion of EasyWeb.

5.2.5 Quality of Generated Codes

Unfortunately, rapid development of EasyWeb makes the generated codes not very qualified in compression of WebLang, WebRB and AndroMDA, which can be solved in next versions of EasyWeb. But I have tried to solve some common problems in developing web application without any extra stuff for designers.

5.2.5.1 Double Clicking and Back Button Problem

One of the most common problems of web applications specially ones in shop payment systems is double clicking on submit buttons and also back button on the internet explorer, solving this problem in AndroMDA is really a big job with a lot of design stuffs but in EasyWeb we never make designer any awareness to manage it, because I just use two session variables in each page; one with the name of page and one with name of "click". In operators I check that the value of "click" is the same as webpage and then firstly before do any thing in clicking event I make it to the name of destination page. So when we click more than one time, or when we come back to page with back button business logic will not be run. In addition we transfer server with value of this session so if you come back with back button after the click the submit button you will be transfer to pervious page but an interesting point is business logics that have any non-query effect will not be run but you can change parameters used in initialization and queries and then see the different selection in the next page, this strategy can satisfy user as much as possible and in other side makes the application secure in the danger of double click or back button problem. Another session is used for initializing this one in load event.

5.2.5.2 Security

All operations and value passing are implemented in “server transfer” and “Context” respectively so you can’t see any changing in url that can be used for some security attacks.

Chapter 6

6 Case Study: Course Management System

In this chapter we will do a simple and typical web application using EasyWeb and show how it works. We cover an experiment for both graphical designer and programming editor to develop this application. The chapter also can be used as a tutorial.

6.1 Brief Description of System Requirements

A Web based Course Management System will be consider as a web application which has a capability to get the list of students in each course with their grades and also change the detail information of a course such a presentation room, department or etc. these activities can be accessed by person who has password of the mentioned course.

Figure 35 shows the scenario for our web application. This figure clarifies fistly user should login and then he has two options to update or getting list of students. For the list option he can edit each student's information such as his/her grade, use also have capability in List page to delete a student.

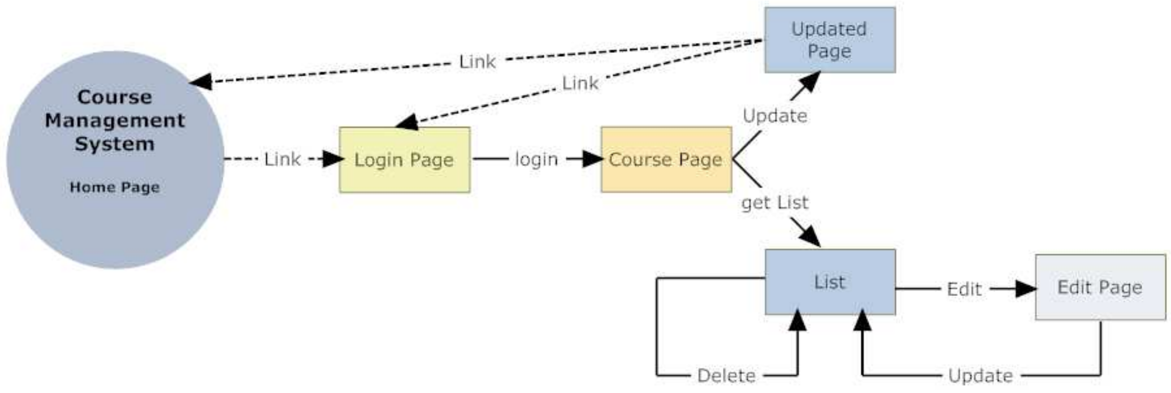


Figure 35. Course Management Scenario

6.2 Designing

Design process using EasyWeb has three steps; *Database Design*, *Webpage Design* and *Business Logic Design*. Generated codes with EasyWeb can have one or more databases with their own tables, and then you can design webpages and connect them to desired database. Finally you should design your business logic which is possible by links, operations, outputs and initializations.

In following firstly we describe the designing by programming editor and then we will switch to graphical designer and show how you can design graphically.

6.2.1 Designing with Programming Editor

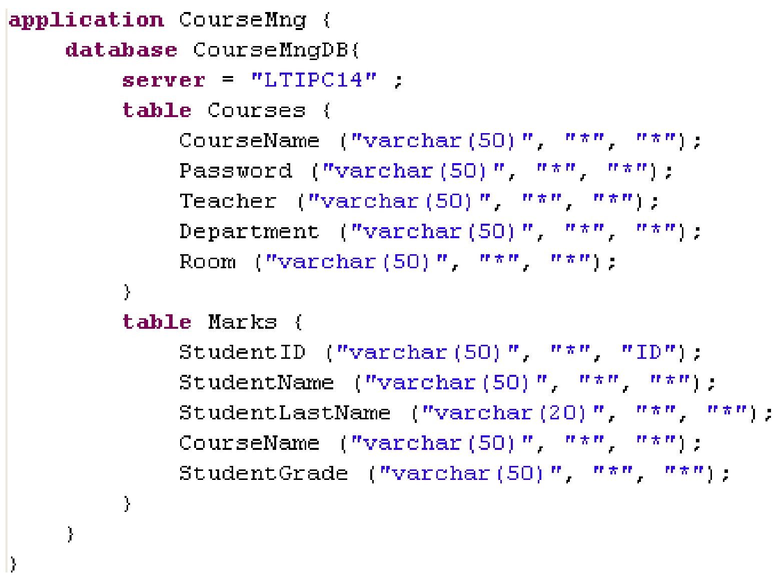
Firstly, you should create a java project in Eclispe and configure it to use our language definition, so please follow the instruction:

1. Create a new Java Project with the Codegen nature.
2. Go into the project's properties, and select as specific engine the location of the newly generated the Project.
3. Create a new cg file called, open it and press CTRL + SPACE, the only available module is a page. Therefore enter a set of Page instances.
4. Control that all default Codegen functionalities are available, syntax highlighting of keywords, completion, structure visualization-selection and syntax error displaying.

Now, you are ready to start to design your system model. As we have already explained, it can be done in three steps:

6.2.1.1 Database Design

We can find two entities in our project and we don't need more than one database (for such a small application). Figure 36 shows the code should be used for modeling.

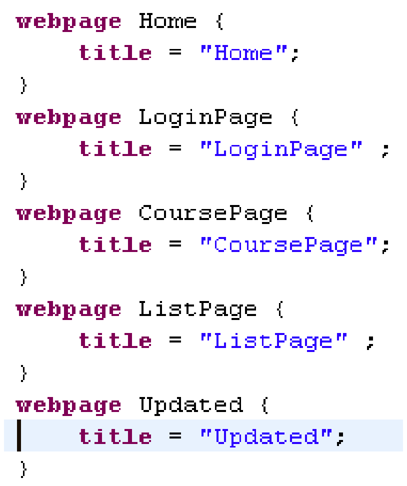


```
application CourseMng {
    database CourseMngDB{
        server = "LTIPC14" ;
        table Courses {
            CourseName ("varchar(50)", "*", "*");
            Password ("varchar(50)", "*", "*");
            Teacher ("varchar(50)", "*", "*");
            Department ("varchar(50)", "*", "*");
            Room ("varchar(50)", "*", "*");
        }
        table Marks {
            StudentID ("varchar(50)", "*", "ID");
            StudentName ("varchar(50)", "*", "*");
            StudentLastName ("varchar(20)", "*", "*");
            CourseName ("varchar(50)", "*", "*");
            StudentGrade ("varchar(50)", "*", "*");
        }
    }
}
```

Figure 36. Database Design in Programming Editor

6.2.1.2 Webpage Design

We have Home, LoginPage, CoursePage, ListPage and Updated pages, so the best way is firstly creating all of them.



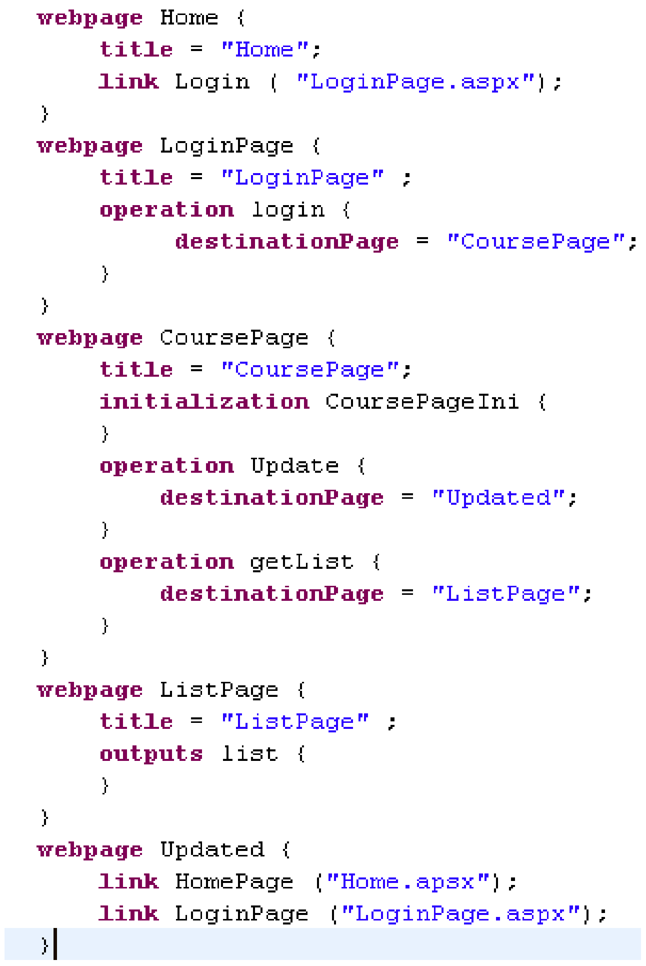
```
webpage Home {
    title = "Home";
}
webpage LoginPage {
    title = "LoginPage" ;
}
webpage CoursePage {
    title = "CoursePage";
}
webpage ListPage {
    title = "ListPage" ;
}
webpage Updated {
    title = "Updated";
}
```

Figure 37. Webpage Design in Programming Editor

6.2.1.3 Business Logic Design

You should consider the relationship between web pages, which can be link or operation. During our design we need to specified data from input in page by @ and data from previous page by ^.

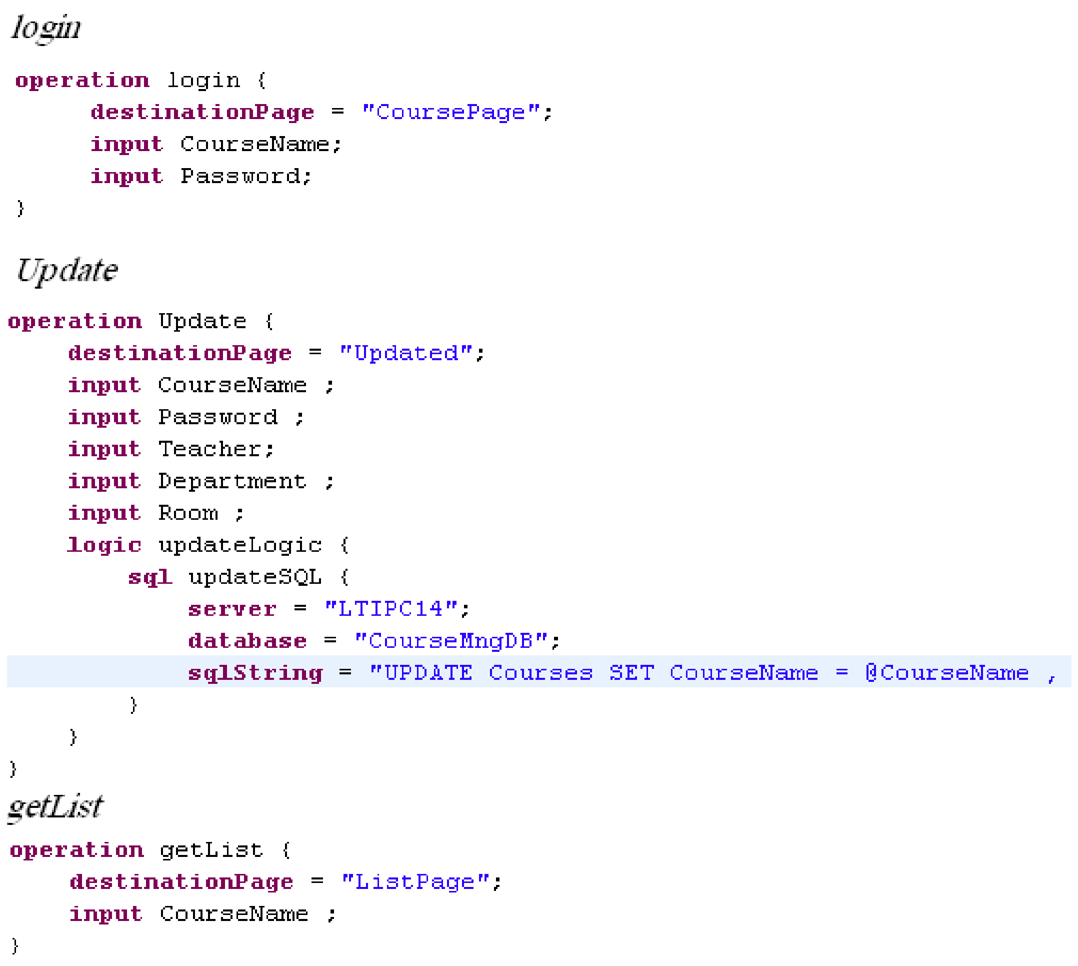
First let us to the skeleton of business logic, our system has links, operations, outputs and initialization as Figure 38:



```
webpage Home {
    title = "Home";
    link Login ( "LoginPage.aspx");
}
webpage LoginPage {
    title = "LoginPage" ;
    operation login {
         destinationPage = "CoursePage";
    }
}
webpage CoursePage {
    title = "CoursePage";
    initialization CoursePageIni {
    }
    operation Update {
        destinationPage = "Updated";
    }
    operation getList {
        destinationPage = "ListPage";
    }
}
webpage ListPage {
    title = "ListPage" ;
    outputs list {
    }
}
webpage Updated {
    link HomePage ("Home.apsx");
    link LoginPage ("LoginPage.aspx");
}
```

Figure 38. Skeleton of Business Logic in Programming Editor

Finally, you should add details of business logic. In this case, we need to design operations; login, Update and getList. Please, consider operations Edit, Update and Delete can be done with outputs(Figure 39).



login

```
operation login {
      destinationPage = "CoursePage";
      input CourseName;
      input Password;
}
```

Update

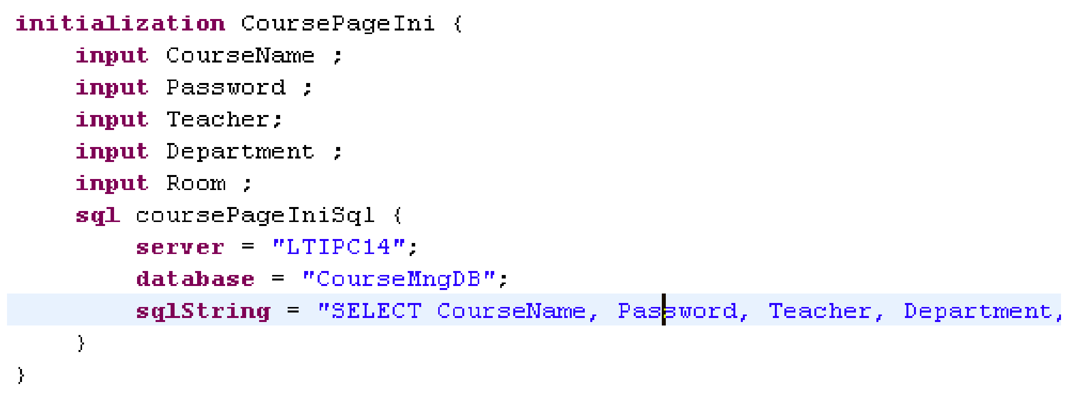
```
operation Update {
     destinationPage = "Updated";
     input CourseName ;
     input Password ;
     input Teacher;
     input Department ;
     input Room ;
     logic updateLogic {
         sql updateSQL {
             server = "LTIPC14";
             database = "CourseMngDB";
             sqlString = "UPDATE Courses SET CourseName = @CourseName ,
         }
     }
}
```

getList

```
operation getList {
     destinationPage = "ListPage";
     input CourseName ;
}
```

Figure 39. Defining Operations in Programming Editor

We have also an initialization in CoursePage, in our initialization we have tried to fill the inputs by course information (Figure 40).

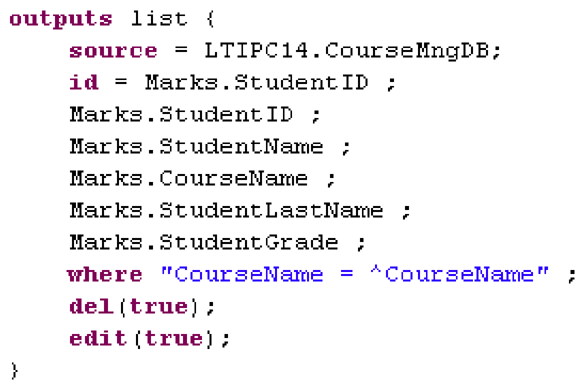


```
initialization CoursePageIni {
     input CourseName ;
     input Password ;
     input Teacher;
     input Department ;
     input Room ;
     sql coursePageIniSql {
         server = "LTIPC14";
         database = "CourseMngDB";
         sqlString = "SELECT CourseName, Password, Teacher, Department,
     }
}
```

Figure 40. Defining Initialization in Programming Editor

The last part of model is output in ListPage, it should have delete and edit facilities, too and it presents data from StudentID, StudentName, StudentLastName, CourseName and

StudentGrade fields of table Marks. As you can see in Figure 41 criteria to select from table Marks was borrowed from CoursePage (^*CourseName*).



```
outputs list {
    source = LTIPC14.CourseMngDB;
    id = Marks.StudentID ;
    Marks.StudentID ;
    Marks.StudentName ;
    Marks.CourseName ;
    Marks.StudentLastName ;
    Marks.StudentGrade ;
    where "CourseName = ^CourseName" ;
    del(true);
    edit(true);
}
```

Figure 41. Designing output in Programming Editor

Whole of model can be found in Appendix B.

After Finishing your Model you should Transform it to Program and then generate codes, so go to EasyWeb and drag CodeGenerator Tools then validate it and you will see an open dialog same as Figure 29, so choose your .tt file and then in the next form (Figure 30) choose your mode as Programming Mode. Then with *saving* mentioned .tt file or *Run it as Custom Tool*, you can generate Course Management System's executable code.

6.2.2 Designing with Graphical Designer

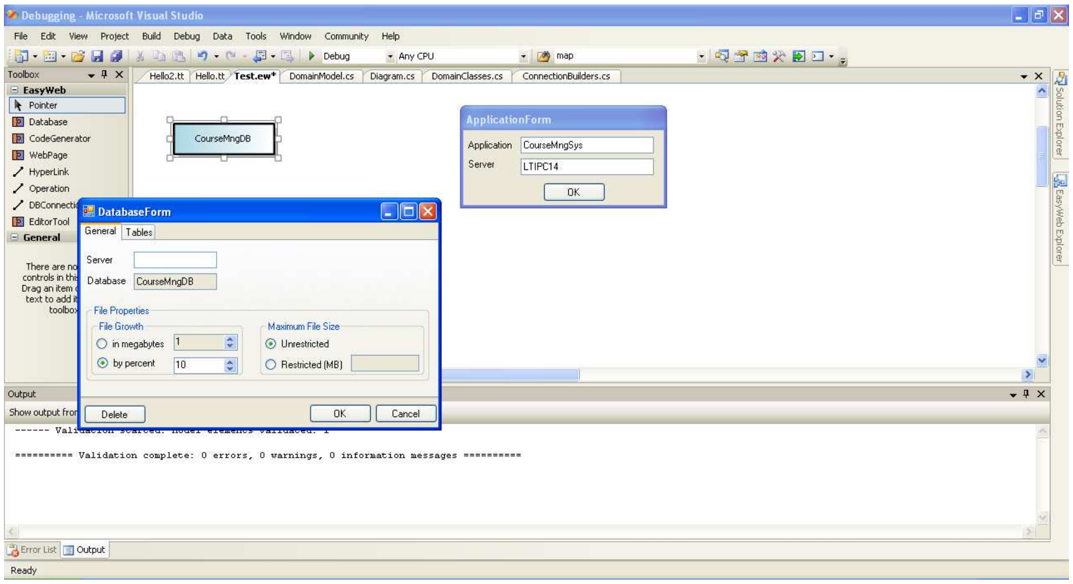
Graphical designer follows steps same as Programming Editor. Here we will show how you can design a web application such as Course Management System by graphical designer but actually, you can feel even for very simple applications graphical designer has a lot of limitations which has been raised from wizard nature and its restrictions.

6.2.2.1 Database Design

We want to design our database and two mentioned entities in pervious section, so please follow the steps:

1. Drag a *Database* entity from toolbox and then by right clicking on it and choosing properties, you can change its name.

2. Right click on your database and choose validation you will see a windows form ask you application name and server of your application. The server here is not server of your database but it's a server that metadata will be stored there, so it will be used by EasyWeb through the designing (Figure 42).



ApplicationForm

Application	CourseMngSys
Server	LTIPC14

DatabaseForm — General | Tables

Server	
Database	CourseMngDB

File Properties
File Growth: ( ) in megabytes 1; (●) by percent 10
Maximum File Size: (●) Unrestricted; ( ) Restricted (MB)

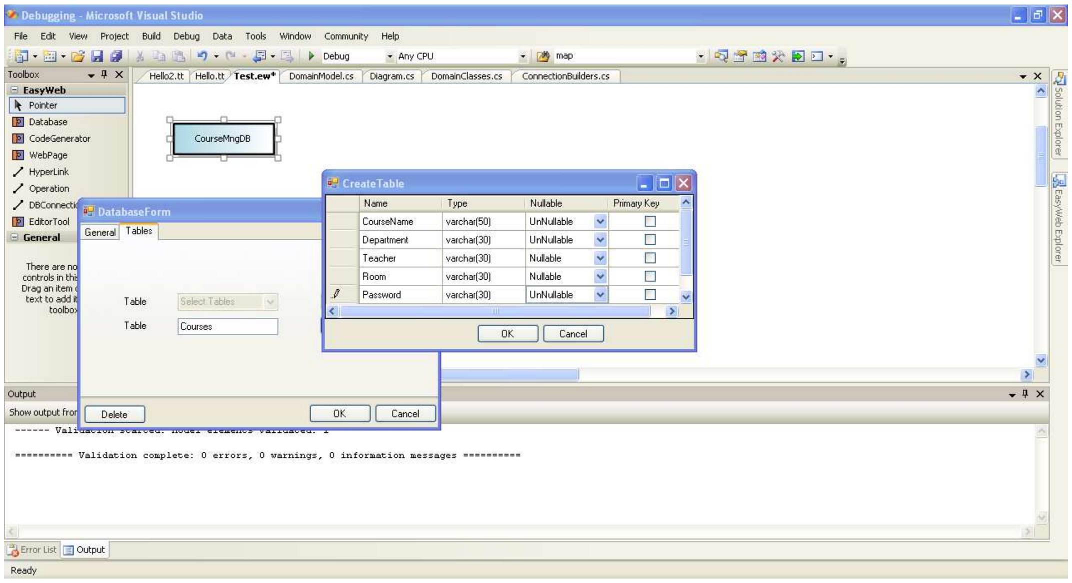
```
------ Validation started: Model elements validated: 1

========== Validation complete: 0 errors, 0 warnings, 0 information messages ==========
```

Figure 42. Specifying the Application Name and the Server that EasyWeb should use in Graphical Designer

3. Create your database by specifying the properties asked in the next form and click on Create.

4. Go to *Tables Tab* and create your tables or edit the created ones. Please note creating database here means storing metadata in EasyWeb.

5. Specify all columns or delete a table in the form which will be appeared after clicking *Create Button* (Figure 43).



DatabaseForm — General | Tables

Table	Select Tables
Table	Courses

CreateTable

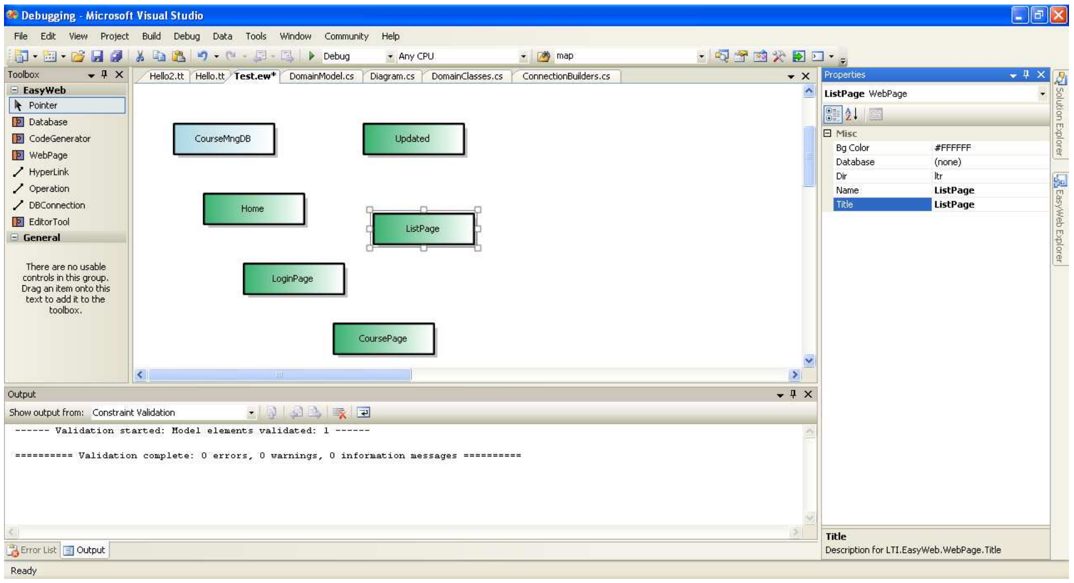
Name	Type	Nullable	Primary Key
CourseName	varchar(50)	UnNullable	☐
Department	varchar(30)	UnNullable	☐
Teacher	varchar(30)	Nullable	☐
Room	varchar(30)	Nullable	☐
Password	varchar(30)	UnNullable	☐

```
------ Validation started: Model elements validated: 1

========== Validation complete: 0 errors, 0 warnings, 0 information messages ==========
```

Figure 43. Specifying Tables in Graphical Editor

6.2.2.2 *Webpage Design*

The same as Programming Editor, it’s a good tip that we first define all our webpages, so for the next step we have defined all of our webpages Figure 44.



ListPage WebPage

Misc

Bg Color	#FFFFFF
Database	(none)
Dir	ltr
Name	ListPage
Title	ListPage

Title
Description for LTI.EasyWeb.WebPage.Title

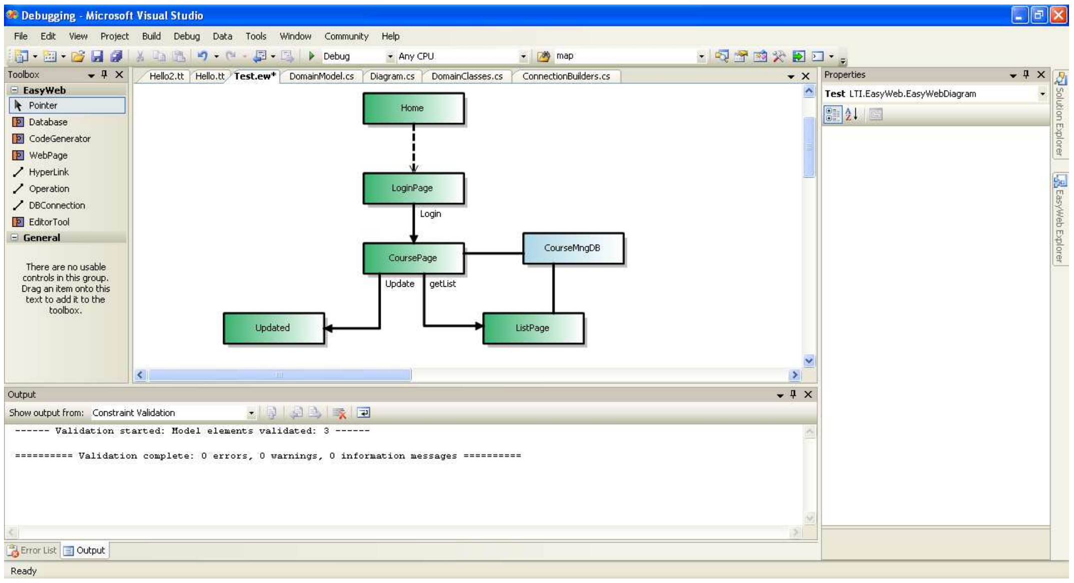
```
------ Validation started: Model elements validated: 1 ------

========== Validation complete: 0 errors, 0 warnings, 0 information messages ==========
```

Figure 44. Defining Webpages in Graphical Designer

Now, we should define our business logics in our system, here business logic design is the same as what we have described in programming editor.

6.2.2.3 *Business Logic Design*

Firstly, we should design the relations of webpages, which can be link or operation (Figure 45).



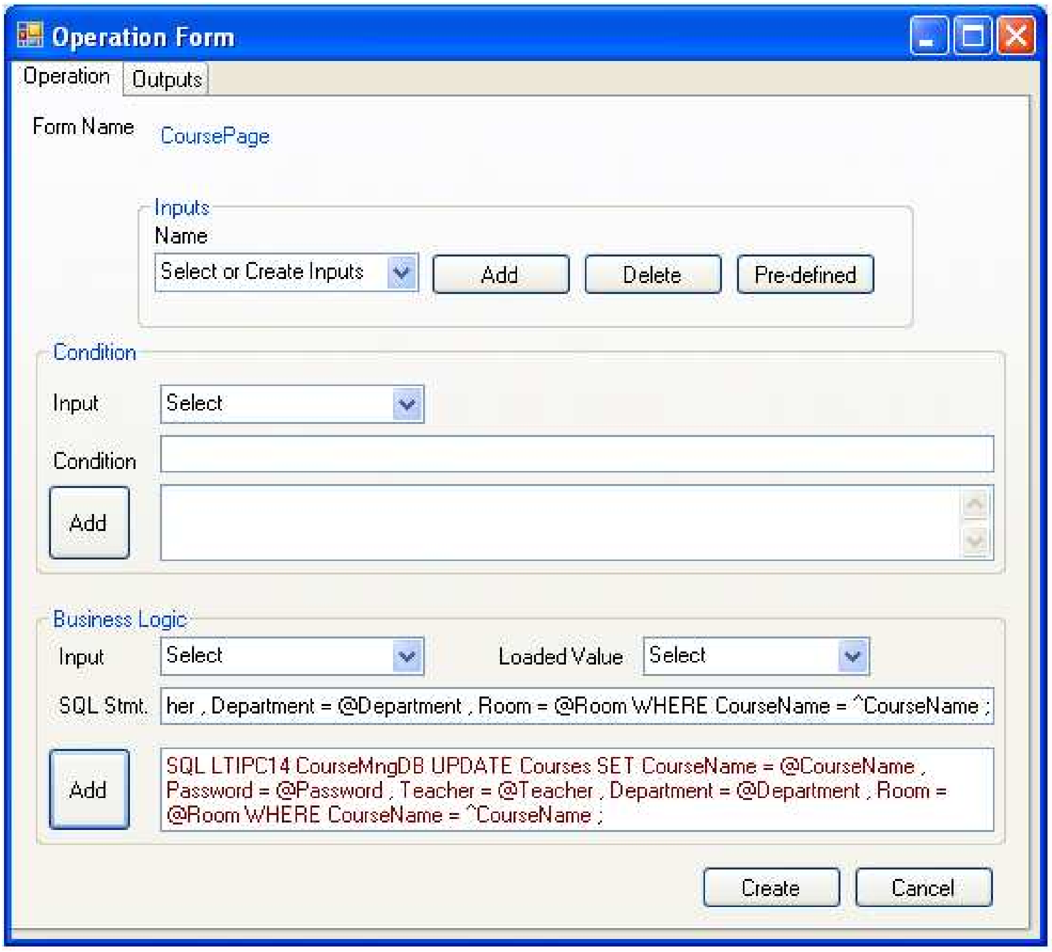
Test LTI.EasyWeb.EasyWebDiagram

```
------ Validation started: Model elements validated: 3 ------

========== Validation complete: 0 errors, 0 warnings, 0 information messages ==========
```

Figure 45. Skeleton of Business Logic Design in Graphical Designer

The next step is defining operations and what exactly we expect from an operation. Firstly, we should design login with adding *CourseName* and *Password* inputs. Then we can do exactly the same for *getList* to define *CourseName* input because in these two operations we just need to pass parameters and there isn't any business logic inside. Finally we have to define Update operation, so right click on this operation and choose validate then add *CourseName*, *Password*, *Department*, *Teacher* and *Room* as inputs. To add logic of update you should write your SQL in SQL Stmt and add it, helping available inputs and loaded values Comboboxes for @ and ^ values can be useful (Figure 46). Update SQL string that you should write is :

UPDATE Courses SET CourseName = @CourseName , Password = @Password , Teacher = @Teacher , Department = @Department , Room = @Room WHERE CourseName = ^CourseName ;



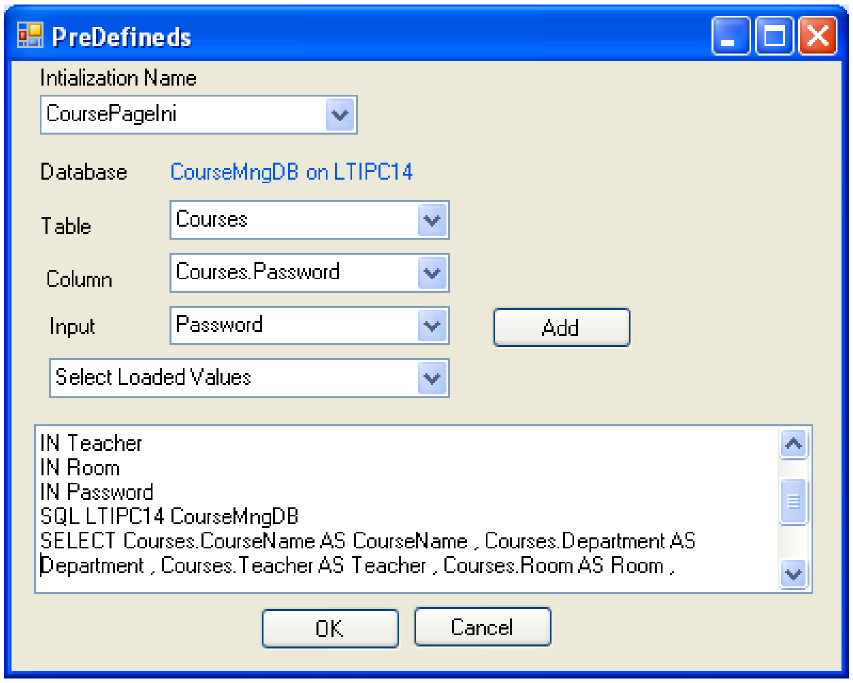
her , Department = @Department , Room = @Room WHERE CourseName = ^CourseName ;

SQL LTIPC14 CourseMngDB UPDATE Courses SET CourseName = @CourseName , Password = @Password , Teacher = @Teacher , Department = @Department , Room = @Room WHERE CourseName = ^CourseName ;

Figure 46. Design getList Operation

Initializations which are called Predefined variables now are ready to define, and we have one case of this kind of entities; Course*PageIni* .To define *CoursePageIni,* go to login operation and choose the name, and the table. Then you should assign fields of table to your variables, in the case of this initialization we should add *CourseName*, *Password*, *Department*, *Room* and Teacher (Figure 47). Finally you have to define your SQL automatically and you should add it the following condition:

WHERE CourseName = ^CourseName AND Password = ^Password



Intialization Name: CoursePageIni
Database: CourseMngDB on LTIPC14
Table: Courses
Column: Courses.Password
Input: Password
Select Loaded Values

```
IN Teacher
IN Room
IN Password
SQL LTIPC14 CourseMngDB
SELECT Courses.CourseName AS CourseName , Courses.Department AS
Department , Courses.Teacher AS Teacher , Courses.Room AS Room ,
```

Figure 47. Initialization of CoursePage: CoursePageIni

The last step is defining the Output, so you should go to *getList* operation and press the Outputs Tab. Then choose the *Marks* Table and add all fields (in the case of this example), you also have to choose an id and you may want to define a where clause (in this case: CourseName = ^CourseName). After choosing *Edit* and *Delete* checkboxes, press OK to create output.

Now you have already modeled your system and only things that have been remained is instantiating a *Code Generator* tool, choosing .tt file and *Graphical Generation* (as your mode) and finally, *save* .tt file or *Run as a Custom tool* to generate codes.

You can find the Intermediate code for Course Management System in Appendix B, please consider that this code is the same for both graphical and programming environment.

Chapter 7

7 Conclusion

This chapter presents the most important points and contributions that I have obtained and also propose next steps to continue this project.

7.1 Summary of Contributions

To sum up all we can count as my contributions, we should say I have developed a prototyping tool for first draft web applications in .Net environment, which can be used for faster and much easier requirement engineering and architectural design. The most important point of this tool is the goal for being easy to use, and not covering all stuffs with complexity of design.

I have tried to decreases the design and analysis phases time and make them much closer to implementation of application with generating codes from architectural design. In this case developers don't need to be busy by some configuration and boring stuffs.

During this Project, I have studied the *Model Driven Development*, its approaches and *Domain Specific Languages* then chosen *Software Factory* for my tool. I have tried to modify the customary architecture of *Microsoft DSL tools*. I have also developed a Programming Editor with WebLang Codegen.

Generated codes however are not comparable with strang codes in WebLang, WebRB or AndroMDA but it can solve common problems such as *double clicking of submit buttons* or *url security* without any extra stuff for designer.

7.2 Future Works

There are few aspects of EasyWeb that I have not studied deeply and they open new directions to continue for next versions of this tool:

- *Ability to generate webservices:* nowadays, web service are going to be most important architecture of web applications, so supporting it is necessary for each tool wants to be used by designers and developers.

- *Generating human-usable code from Program.txt:* If this ability was possible in this version I could use graphical interface for very fast design and then insert all details with programming editor, so developing this possibility can be helpful.

- *Improving generated codes*: As I have mentioned before, generated codes are not strong as much as other tools in this area, so it seems useful if I will improve the quality of generated codes.

- *Ability to generate databases from logical design:* currently, EasyWeb supports physical design and it's really needed to use logical design for generating databases because the goal of EasyWeb is making the system design easy as much as possible, but now there is a person who works on this ability in LTI, I hope I can add this feature soon.

In summary the final goal of my project is developing an easy to use tool which should cover most technologies in this area. To achieve this goal we should define different tools but not include them in a one environment because it can be resulted in complexity and much more parameters for design a simple application.

Bibliography

A. Basu, M. Hayden, G. Morrisett, T. von Eicken. "A language-based approach to protocol construction." *The First ACM SIGPLAN Workshop on Domain-Specific Languages.* ACM, 1997. 1-15.

A. Henriksson, H. Larsson. *A Definition of Round-trip Engineering.* Technical Report, Sweden: Linkopings Universitet, 2003.

A. Leff, J. Rayfield. "What is Web Relational Blocks?" *IBM.* 2007. http://services.alphaworks.ibm.com/webrb/ (accessed November 1, 2007).

A. Moss, H. Muller. " Efficient Code Generation for a Domain Specific Language." *Generative Programming and Component Engineering Conference.* Springer Verlag, 2005. 47-62.

A. V. Deursen, E. Klint. "Little languages: Little maintenance? ." *Journal of Software Maintenance*, 1998: 75-92.

A. V. Deursen, P. Klint, J. Visser. "Domain-specific languages: an annotated bibliography." *ACM SIGPLAN Noti, Volume 35 , Issue 6, ISSN:0362-1340* , 2000: 26-36.

A.V. Aho, R. Sethi, J.D. Ullman. *Compiler: Principles, Techniques and Tools.* Addison-Wesley, 1986.

Bentley, J. L. "Programming pearls: Little languages." *ACM Communications*, 1986: 711-721.

Bhatia, Naresh. "AndroMDA Introduction." *AndroMDA.* 2006. http://galaxy.andromda.org/index.php?option=com_content&task=view&id=104&Itemid=89 (accessed November 1, 2007).

Brown, A. "An introduction to Model Driven Architecture." *IBM.* 2004. http://www.ibm.com/developerworks/rational/library/3100.html (accessed November 1, 2007).

Bruce, D. "What makes a good domain-specific language? ." *Symposium of APOSTLE, and its approach to parallel discrete event simulation.* 1998.

C. Petitpierre, O. Buchwalder, P-L Meylan. "WebLang: Web-Applications Development IDE for Eclipse and JBoss ." *LTI.* 2006. http://ltiwww.epfl.ch/WebLang/ (accessed november 2, 2007).

Cleaveland, J. C. "Building application generators." *IEEE Software*, 1988: 25-33.

Coplien, J. "Multi-paradigm Design." *Generative and Component-Based Software Engineering (GCSE99).* Germany, 1999.

D. A. Ladd, J. C. Ramming. "Two application languages in software production." *USENIX Very High Level Languages Symposium.* 1994. 169-178.

Demir, A. "Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development." *The Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD/MOMPES'06).* IEEE 0-7695-2538-5 /06, 2006.

"Domain-Specific Language Tools." *MSDN.* 2007. http://msdn2.microsoft.com/en-us/library/bb126235(vs.80).aspx (accessed November 1, 2007).

DSM Publication. 2007. http://www.dsmforum.org/publications.html (accessed November 1, 2007).

E. G. Sirer, B. N. Bershad. "Using production grammars in software testing." *The second USENIX Conference on Domain-Specific Languages.* USENIX Association, 1999.

Frye, Colleen. "A look at Visual Studio Shell." *TechTarget.* 2007. http://searchvb.techtarget.com/originalContent/0,289142,sid8_gci1273412,00.html (accessed November 1, 2007).

Garwick, J. V. "Programming Languages: GPL, a truly general purpose language." *ACM Communications, Volume 11 Issue 9* , 1968.

Greenfield, J. "Software Factories." *MSDN.* 2007. http://msdn2.microsoft.com/en-us/library/ms954811.aspx (accessed November 1, 2007).

H. N. Pham, Q. H. Mahmoud, A. Ferworn, A. Sadeghia. "Applying Model-Driven Development to Pervasive System Engineering." *First International Workshop on Software Engineering for Pervasive Computing, Applications, Systems, and Environments (SEPCASE'07).* IEEE 0-7695-2970-4/07, 2007.

J. Greenfield, K. Short. *Software Factories.* Wiley Publishing, ISBN 0-471-20284-3, 2004.

J. Mukerji, J. Miller. *OMG: MDA Guide V1.0.1.* 2003. http://www.omg.org (accessed October 31, 2007).

J. Rayfield, A. Leff. *IBM Web Relational Blocks Developer's Manual.* Developer's Manual, IBM Corp, 2007.

James, M. "Visual Studio Domain Specific Language tools." *IT Architecture.* 2007. http://www.itarchitect.co.uk/articles/display.asp?id=334 (accessed October 31, 2007).

Kovari, P. "Explore model-driven development (MDD) and related approaches: Applying domain-specific modeling to Model-Driven Architecture." *IBM.* 2004. http://www.ibm.com/developerworks/architecture/library/ar-mdd4/?S_TACT=105AGX78&S_CMP=HP (accessed November 1, 2007).

Kozikowski, J.l. "A Bird's Eye View of AndroMDA." *AndroMDA.* 2005. http://galaxy.andromda.org/docs- 3.1/contrib/birds-eye-view.html (accessed November 1, 2007).

Krueger, C. W. "Software reuse." *ACM Computing Surveys*, 1992: 131-183.

M. E. Fayad, D. C. Schmidt. "Object-oriented application frameworks." *ACM Communications*, 1997: 22-35.

Microsoft. "Features in the Visual Studio Premier Partner Edition." *MSDN.* 2005. http://msdn2.microsoft.com/en-us/library/bb129445(VS.80).aspx (accessed November 1, 2007).

"Microsoft DSL Tools." *MSDN.* 2005. http://msdn2.microsoft.com/en-us/library/bb126327(VS.80).aspx (accessed October 31, 2007).

O. Buchwalder, C. Petitpierre. "WebLang: A Language for Modeling and Implementing Web Applications." *18th International Conference on Software Engineering and Knowledge Engineering.* 2006.

R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, L. Walton. "A software engineering experiment in software component generation." *The 18th International Conference on Software Engineering(ICSE-18).* IEEE, 1996. 542-553.

R. E. Johnson, B. Foote. "Designing reusable classes." *Journal of Object-Oriented Programming, 1(2):22-35*, 1988.

R. M. Herndon, V. A. Berzins. " The realizable benefits of a language prototyping language." *Transactions on Software Engineering*, 1988: 803-809.

R. N. Taylor, W. Tracz, L. Coglianese. "Software Development Using Domain-Specific Software Architectures." *ACM SIGSOFT Software Engineering Notes*, 1995: 27- 37.

Rothenberg, J. *The Nature of Modeling.* New York, USA: John Wiley and Sons, Inc. , 1989.

Schwaderer, C. "Pioneering model driven development." 2006. http://www.compactpci-systems.com/columns/software_corner/pdfs/2006,10.pdf (accessed November 1, 2007).

Siegel, J. *Introduction to OMG's Model Driven Architecture* . Presentation, OMG, 2005.

Simos, M. "Organization domain modeling (ODM): Formalizing the core domain modeling life cycle." *The Symposium on Software Reusability (SSR95).* ACM Software Engineering Notes, 1995. 196–205.

V. Menon, K. Pingali. "A case for source-level transformations in MATLAB." *The second USENIX Conference on Domain-Specific Languages.* USENIX Association, 1999.

Appendix A

Technical Tutorial of Microsoft DSL Tools

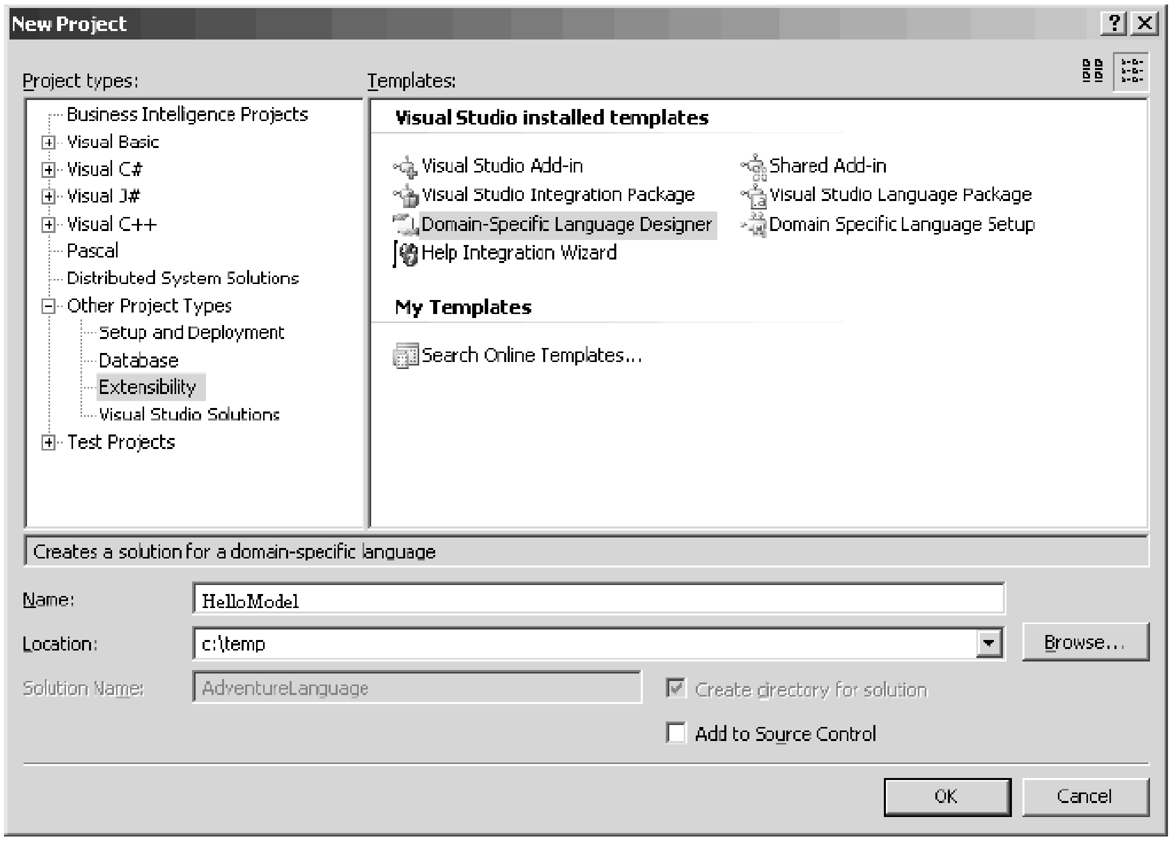
Considering the lack of good technical tutorials for MS- DSL Tools, in this appendix we will describe how we can technically develop a designer environment. Firstly, we will consider the environmental setup and then describe the language definition, capable to write hello world.

Environmental Setup

1. Install Visual Studio .NET 2005.
2. Install Visual Studio SDK (Software Development Kit), which contains the DSL Tools plug-in.

Language Definition

Open Visual Studio 2005, select New Project and then navigate to Other Project Types, Extensibility and select Domain-Specific Language Designer. Assign the new project the name “HelloModel” and a suitable storage location (Figure 48).


New Project

Project types:

Business Intelligence Projects
Visual Basic
Visual C#
Visual J#
Visual C++
Pascal
Distributed System Solutions
Other Project Types
Setup and Deployment
Database
Extensibility
Visual Studio Solutions
Test Projects

Templates:

Visual Studio installed templates

Visual Studio Add-in
Shared Add-in
Visual Studio Integration Package
Visual Studio Language Package
Domain-Specific Language Designer
Domain Specific Language Setup
Help Integration Wizard

My Templates

Search Online Templates...

Creates a solution for a domain-specific language

Name: HelloModel

Location: c:\temp

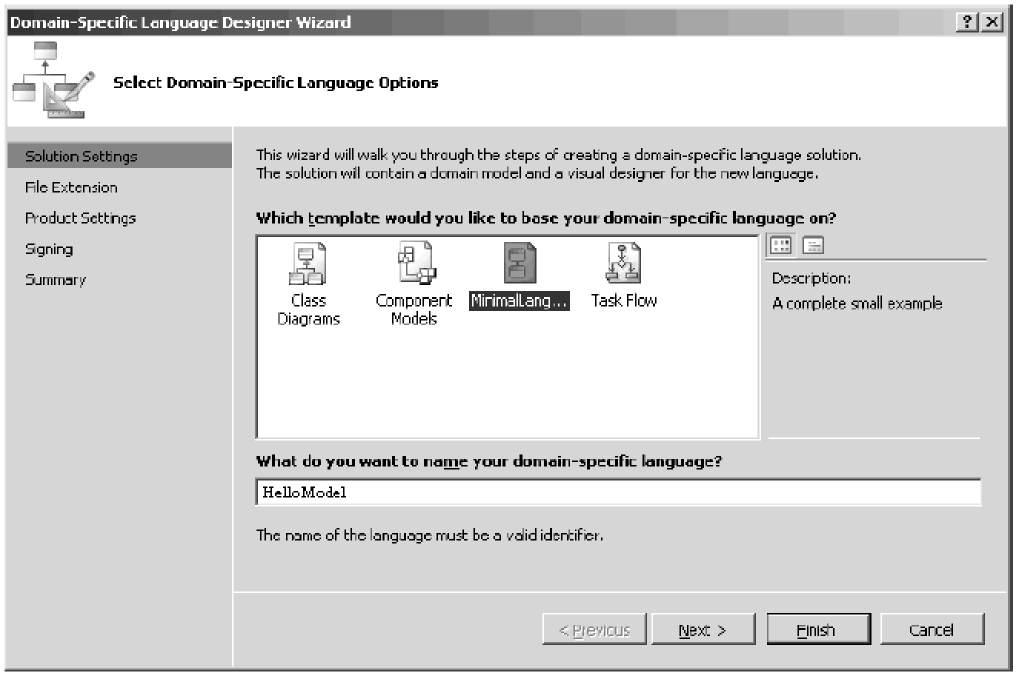
Solution Name: AdventureLanguage

[✓] Create directory for solution

[ ] Add to Source Control

Figure 48. Creating the project

Select a suitable template for the custom Designer. In our case the very basic Minimal Language template will do, and we can also accept the default name “HelloModel” for the DSL generated (Figure 49).



This wizard will walk you through the steps of creating a domain-specific language solution.
The solution will contain a domain model and a visual designer for the new language.

Which template would you like to base your domain-specific language on?

Description:
A complete small example

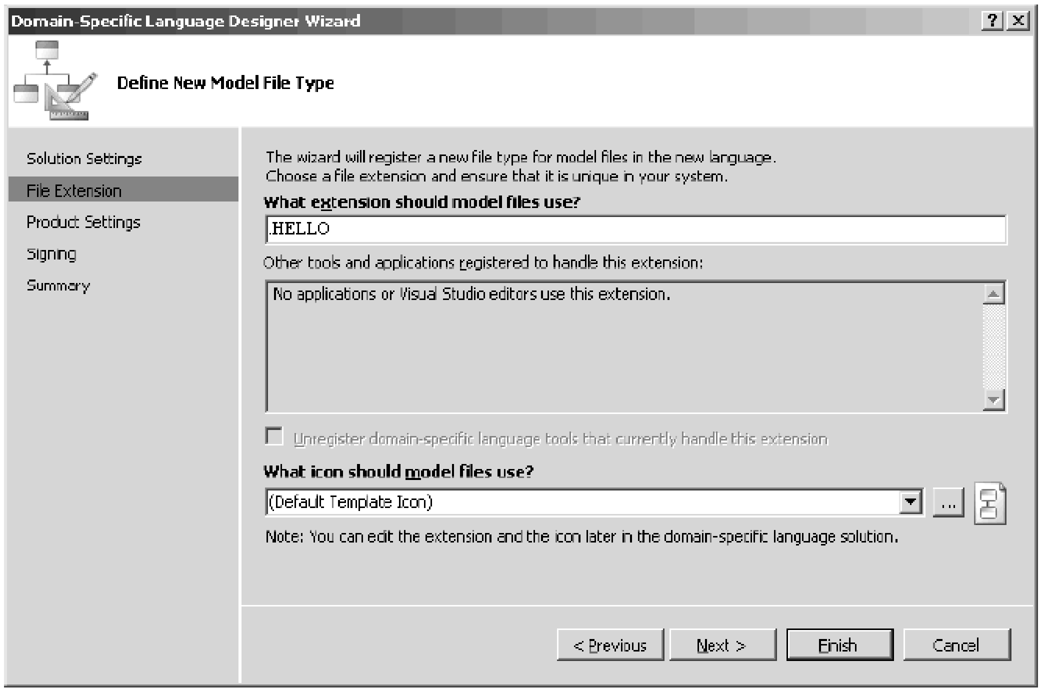
What do you want to name your domain-specific language?

HelloModel

The name of the language must be a valid identifier.

Figure 49. Selecting Domain Specific Language Options

Specify an extension to be associated with the custom editor – enter .HELLO in this case – and leave everything else as the default (Figure 50).



The wizard will register a new file type for model files in the new language.
Choose a file extension and ensure that it is unique in your system.

What extension should model files use?

.HELLO

Other tools and applications registered to handle this extension:

No applications or Visual Studio editors use this extension.

☐ Unregister domain-specific language tools that currently handle this extension

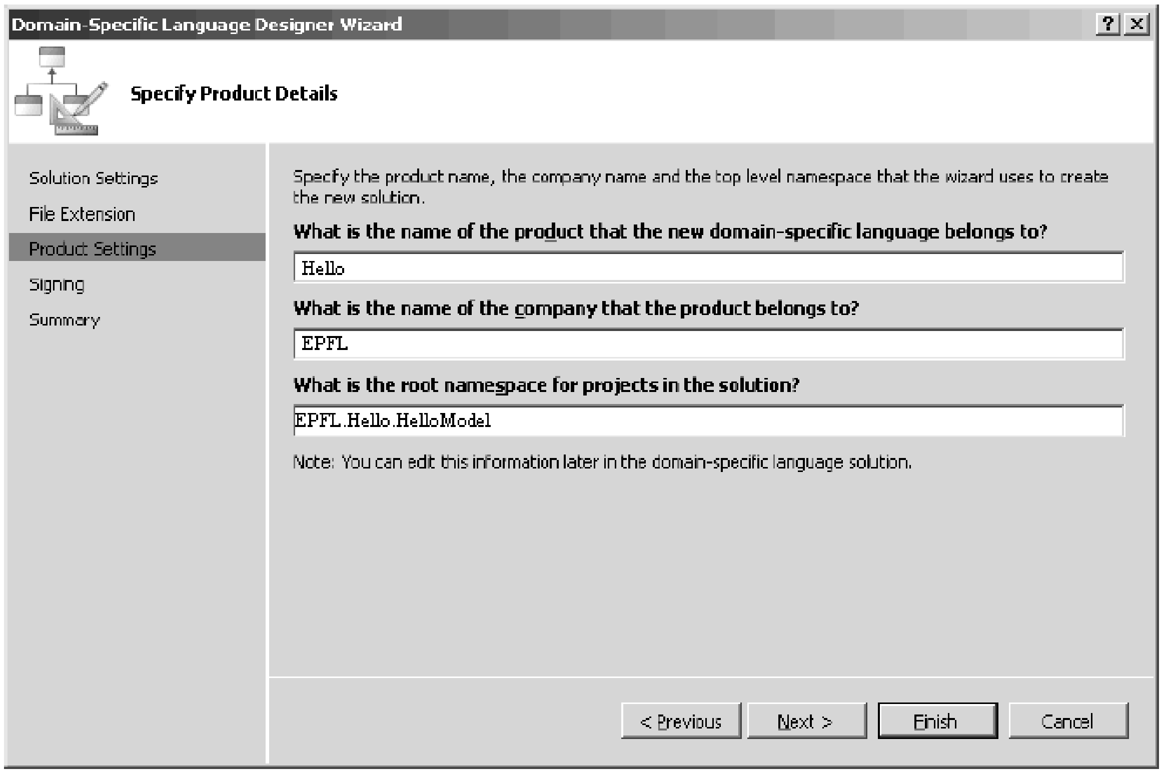
What icon should model files use?

(Default Template Icon)

Note: You can edit the extension and the icon later in the domain-specific language solution.

Figure 50. Defining New Model File Type

Set details of the names and namespace used by the model (Figure 51).



Domain-Specific Language Designer Wizard

Specify Product Details

Solution Settings
File Extension
Product Settings
Signing
Summary

Specify the product name, the company name and the top level namespace that the wizard uses to create the new solution.

What is the name of the product that the new domain-specific language belongs to?

Hello

What is the name of the company that the product belongs to?

EPFL

What is the root namespace for projects in the solution?

EPFL.Hello.HelloModel

Note: You can edit this information later in the domain-specific language solution.

< Previous | Next > | Finish | Cancel

Figure 51. Specify Product Details

The final page of the Wizard generates a strong name key file and completes the specification of the project. If you click finish the project will be generated.

Then you can see the *DSLDefinition.dsl* file displayed as a designer diagram. If you make any changes you might well need to re-generate the code by transforming before running the project. If you examine the diagram your first thoughts might be that this looks complicated.

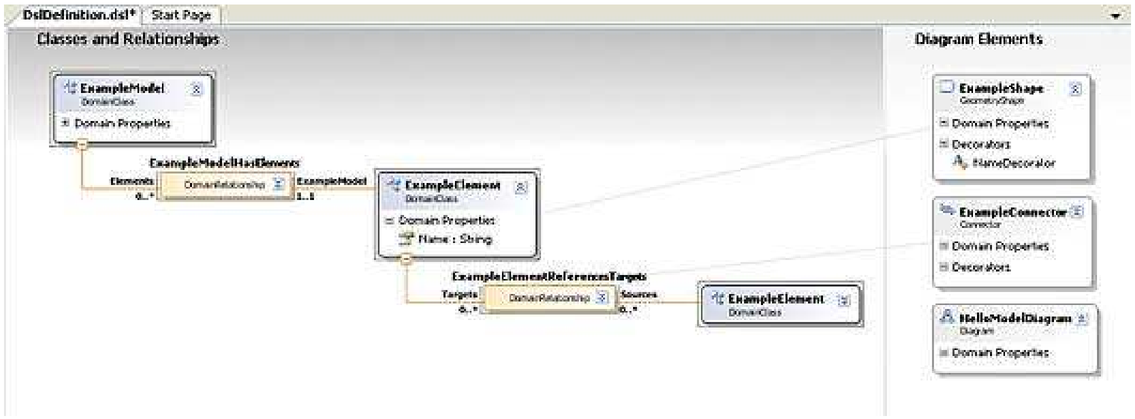
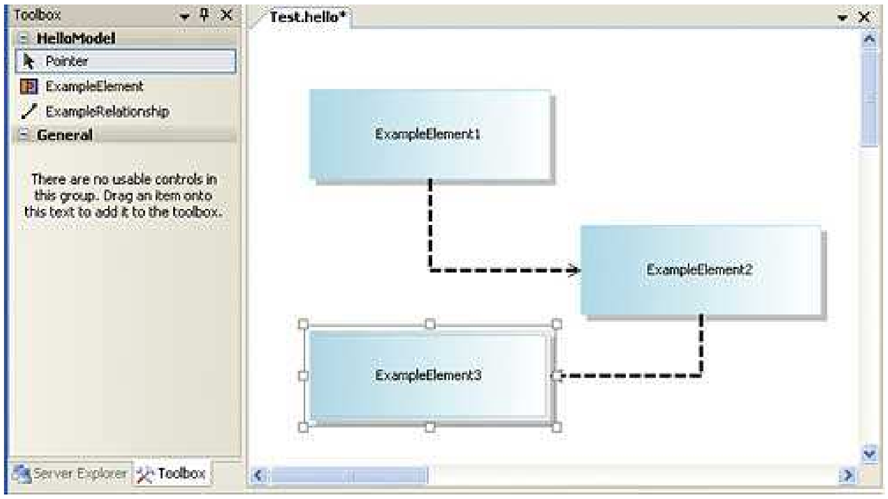


Figure 52. Designer Diagram (James 2007)

You can see that the diagram is made up of a number of “blocks” or Domain Classes linked by a number of Domain Relationships. The first block is called “ExampleModel” – you can think of this as the topmost block that contains all of the other blocks that the user might put on the diagram. These other blocks are related to the ExampleModel block by the ExampleModelHasElements relationship. If you look closely at Elements end of this relationship you will see the notation 0..*, which means that there can be 0 or many Elements connected to the ExampleModel. At the other end of the relationship you will see the notation 1..1, meaning that there is just 1 ExampleModel block, i.e. this relationship is one-many which is what you would expect (James 2007).

Moving on you can also see that there is an ExampleElement block with a single property “Name”. This just means that the user can place this block on the diagram and assign each one a name. You can add other user-settable properties to blocks, but for the moment a Name property is sufficient. Each ExampleElement can be related to other example elements by an ExampleElementReferencesTargets DomainRelationship. You can also see that in this case both Targets and Sources are set to be “many”, and this allows the user to place multiple ExampleElement blocks on the diagram and link them using this relationship in a many-to-many way. If you are finding this hard to follow it becomes a lot easier once you have seen the generated custom Designer. Exactly how the custom Designer’s code is generated will also become clear once we have added some generated code of our own (James 2007).

Now we can move to the next stage of creating a custom Designer. Make sure that all of the generated code is up-to-date – in the Solution Explorer click the new Transform All Templates button. Next start the project running using Debug, Start Debugging or press F5. What happens next might be slightly unexpected. A new copy of Visual Studio opens with your custom Designer already loaded plus a sample and a test diagram. To see how things work, open the Test.hello file using the Solution Explorer (Figure 53). You will see a blank diagram, and if you open the Toolbox you will see that it contains ExampleElement and ExampleRelationship. You can now drag ExampleElements onto the diagram and connect them up using the ExampleRelationship. You will discover that you can use the ExampleRelationship to create many-to-many connections (James 2007).



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

Figure 53. Custom Designer (James 2007)

The key to code generation in all DSL models is the use of TextTempate or .tt files. These can contain a mixture of code which is executed to generate more code, and code which can be simply quoted into the generated code. This “dual use” makes the TextTemplate difficult to fathom at first, but if you have had any encounters with meta-languages, or indeed with the way that HTML is generated using PHP or the DOM, then you should be in a better position to follow what is going on (James 2007).

Another puzzle to solve is exactly where the TextTemplates should be included in the project – in the original DSL Designer or in the custom Designer that it generated? Clearly the answer is that if you want to modify or augment the code generated by the DSL Designer, add to the large number of TextTemplates it already has. If you want to add code generation to the custom Designer then the TextTemplate needs to be included in the generated project. It seems obvious when put so clearly, but it is a little odd to be adding files to a generated project. The good news is that any files you do add are preserved if you re-generate the custom Designer (James 2007).

To add a TextTemplate to the project simply use the Add,New Item in the Solution Explorer and create a Text File called Hello.tt. You don’t have to use the .tt extension, but if you do, two things happen automatically. The first is that the files “Custom Tool” is set to TextTemplatingFileGenerator, and second this custom tool is run on the file to generate some code. If you use any other extension for the TextTemplate then you have to manually set the CustomTool property to the template processor – usually TextTemplatingFileGenerator (James 2007).

In this case of course the TextTemplate is blank and hence so is the generated code file. If you look in the Solution Explorer you should see Hello.cs just below Hello.tt in the hierarchy. By default generated code files are .cs files and are supposed to contain C# code – but you can change this (James 2007).

TextTemplate: To set the scene, templates consist of some directives followed by a mixture of literal blocks and control blocks. Literal blocks are just plain text in the template that you want to pass straight through to the output. Control blocks are things with some kind of <% %> marker around them. The content of these blocks in your template contributes to a static class which the templating system generates. This class has at least one static method, which when executed writes out the desired output of the transformation of the template. In the parlance of the current March release we'd call this the generated code, but we're moving the naming over to be 'Text Templating' and "Transformation" as it's not just code you can generate, rather it is any text-based artefact. If you do nothing else in your template, the static method will write out all of the text in literal blocks by simply writing out the raw text using a simple WriteLine() type statement. There can only be one class feature block and it must come immediately after the directives.

<% %> - Regular control block: Embeds some control code in the static method. Regular control blocks affect the literal blocks that they surround by means of the flow of control within them. This happens because the regular blocks' code is written verbatim into the static method. So if you have a regular block that begins a for loop that loops five times, then have a static block, then have another regular block that closes the for loop, you'll get the contents of the static block written into the final output text five times. You'd typically use some expression blocks inside this loop to do

something interesting with the loop variable. You can put anything in a regular block that you can put in the body of a static method.

<%! %> - Class features control block: Embeds control code at the static class scope. This block allows you to add new static methods, static fields, static properties etc to the static class. You can then use these from regular and expression control blocks. Trying to write all of your control code inside one static function can be tiresome if you have a big model to traverse - especially if the model has a recursive structure like a tree in it. Class features blocks allow you to add further static methods to the static class to make this sort of thing easier and to structure your control code well. You can also add static fields and properties to support those methods if you wish.

Directives: Directives provide instructions to the templating engine.

<%@ directiveName parameter="Value" %>: The current set for any template file is named in following:

<%@ generatedFile extension=".cs" %>: Specify the extension of the file that gets generated.

<%@ assembly name="System.Drawing.dll" %>: Reference the assembly in compilation of control blocks.

<%@ import namespace="System.Collections" %>: Import the namespace in compilation of control blocks. (i.e. a C# using statement)

<%@ modelFile path="UtilitiesModel.dmd" %>: Load the given domain model and provide a reference to it in the property Model on the context object.

<%@ modelFile path="xxx.yy" %>: Load the given designer definition and provide a reference to it in the property Definition on the context object.

The simplest code generator we can implement produces a single HTML page with no dynamic code, i.e. no code that depends on the diagram the user has created. The first thing we have to do is change the extension of the generated code file:

```
<#@ output extension=".htm" #>
```

This is an example of one of the five built-in directives that change the way the code is generated. In this case the output directive can change the extension, and optionally the encoding, used for the output file.

The remainder of the TextTemplate takes the form of HTML that is simply quoted directly into the generated code:

```
<html
xmlns="http://www.w3.org/1999/xhtml" >
        <head>
                <title>Hello World</title>
        </head>
        <body>
                <h1>Hello World</h1>
        </body>
</html>
```

If you enter the directive and the quoted HTML into the Hello.tt file, right-click and select Run Custom Tool, you will see that the Hello.cs file has turned into a Hello.htm file and if you right-click on this and select View in Browser you will see the Hello World Web page.

If this is as far as code generation goes it isn't very impressive, and certainly doesn't need the custom Designer! To do anything meaningful we really need to involve the diagram of the model that the user has created and use dynamic code generation. A TextTemplate can contain a number of different types of dynamic code generation. As already mentioned if you have text enclosed by <# and #> then the text is interpreted as code (C# by default) to be executed during code generation. For example, if you include:

```
<# WriteLine("Hello"); #>
```

…then the generated code contains the text Hello. An expression is an even more direct way of generating dynamic code. If you use

```
<#= ExpressionCode #>
```

This is all very well and you can probably now see how to generate code dynamically from the template, but how do you access details of the diagram the user has created?

The answer is that when the user creates a diagram the system creates a "directive processor" which contains classes and methods that give access to the model as created by the user. Before you can use these classes and methods you have to call the generated directive processor to add them to your template. The only problem with this task is the complexity of the names involved. The directive processor is by default called DirectiveProcessor.cs and stored in the GeneratedCode directory of the project, HelloModel\Dsl\GeneratedCode in the case of our example. The directive processor is used as a custom directive:

<#@ *DSLname* processor=

    "*DSLname*DirectiveProcessor"

    requires="fileName='*diagram.ext*'" #>

The *DSLname* is replaced by the name of your custom DSL – HelloModel in this case. The "requires" clause specifies the instance of the custom diagram that you want to process e.g. Test.hello. You can also specify a "provides" clause to specify what the resulting generated class model should be called. By default it's given the same name as the DSL, i.e. *DSLname*. To make use of the generated class we also need to specify a base class that it will be derived from and this requires a "template" directive.

So to add access the model in the generated code we need to first add two lines to the start of the Hello.tt file:

```
<#@ template inherits=
 "Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation"#>
<#@ HelloModel processor="HelloModelDirectiveProcessor" requires="fileName='Test.hello'" #>
```

Now the code within the template can access the model corresponding to the diagram that the user has drawn in Test.hello.

Now we can write a simple loop that retrieves the name of each box placed on the diagram by the user and uses it to create a "Hello World" message from each box:

```
<html xmlns=
        "http://www.w3.org/1999/xhtml" >
```

```
          <head>
                    <title>Hello World</title>
          </head>
          <body>
          <#
                    foreach(ExampleElement box in
                              this.ExampleModel.Elements)
                    {
          #>
          <p>Hello World from
                    <#=box.Name#></p>
          <#
                    }
          #>
          </body>
</html>
```

Notice that we have a mixture of static and dynamic code generation. The first part of the template always produces the same HTML. The dynamic generation starts at the foreach loop. Notice that the "this" is necessary to make it work with the diagram instance, and that the ExampleModel is the topmost domain class as specified in the DSL designer. The ExampleModel has an Elements collection, which contains all of the ExampleElement domain classes corresponding to each block the user has placed on the diagram. We simply step through each one and use an expression to insert the block's name into the HTML code generated. If you select the TextTemplate, right click and select Run Custom Tool then a new HTML file will be generated, and if the user has placed three blocks on the diagram it will contain:

```
<html xmlns=
          "http://www.w3.org/1999/xhtml" >
          <head>
                    <title>Hello World</title>
          </head>
          <body>
                    <p>Hello World from
                              ExampleElement1</p>
                    <p>Hello World from
                              ExampleElement2</p>
                    <p>Hello World from
                              ExampleElement3</p>
          </body>
</html>
```

The next step is to look into ways of using the generated model in more sophisticated ways. For example, we could write code that generates greetings in the order that the blocks are joined together. This raises a few practical problems because in this case the relationships can be many-to-many, and it is possible to draw a diagram with no unique single route through from block to block. Such problems can be overcome either by making the code clever or by going back to the DSL Designer and putting restrictions on the type of diagram the user can generate, e.g. by making the links one-to-one, say.

There is also the more prosaic reason that moving on to work with more sophisticated aspects of the model is more difficult than you might think. There is virtually no documentation of the class structure, only very specific examples that really don't help. The only solution I've found to the documentation problem is to read the generated code files in the DSL Designer. For example, if you open the file DomainClasses.cs in the original project you will find the definition for the classes ExampleModel and ExampleElement. Looking closely at ExampleElement reveals that it has a Targets property which returns a LinkedElementCollection. If you lookup the definition of LinkedElementCollection you will further discover that this is a collection of ExampleElement objects. With this information we can now generate code that lists all of the elements that each element is linked to.

The changes needed to the generated code are:

```
<body>
    <#
        foreach(ExampleElement box in
            this.ExampleModel.Elements)
        {
    #>
    <p>Hello World from
        <#=box.Name#></p>
    <#
        foreach(ExampleElement targ in
            box.Targets)
            {
    #>
    <p> is connected to
        <#=targ.Name#></p>
    <#
            };
        }
    #>
</body>
```

Notice that the outer foreach loop still steps through each of the blocks placed on the diagram, but the second foreach loop uses each block's Targets property to list all of the blocks it is connected to.

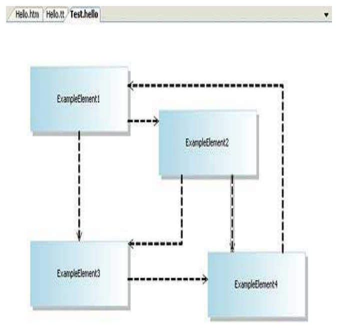


Figure 54. A Complicated Connected Diagram (James 2007)

For example, if the user draws the diagram shown in Figure 19 then the resulting generated HTML is:

```
<html xmlns=
    ”http://www.w3.org/1999/xhtml” >
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <p>Hello World from
            ExampleElement1</p>
        <p> is connected to
            ExampleElement3 </p>
        <p> is connected to
            ExampleElement2 </p>
        <p>Hello World from
            ExampleElement2</p>
        <p> is connected to
            ExampleElement3 </p>
        <p> is connected to
            ExampleElement4 </p>
        <p>Hello World from
            ExampleElement3</p>
        <p> is connected to
            ExampleElement4 </p>
        <p>Hello World from
            ExampleElement4</p>
        <p> is connected to
            ExampleElement1 </p>
    </body>
</html>
```

Similarly, by reading the other generated code files you should be able to work out how to generate code based on just about any aspect of the diagram. Notice that it’s very easy to add custom properties to the model elements and to use this to change the generated code. For example, adding a “country” property to the ExampleElement allows the code generator to test it and generate a “Hello World” in the appropriate language (James 2007).

Appendix B

Course Management System Models

Model in Human-Usable Language

```
application CourseMng {
     database CourseMngDB{
          server = "LTIPC14" ;
          table Courses {
               CourseName ("varchar(50)", "*", "*");
               Password ("varchar(50)", "*", "*");
               Teacher ("varchar(50)", "*", "*");
               Department ("varchar(50)", "*", "*");
               Room ("varchar(50)", "*", "*");
          }
          table Marks {
               StudentID ("varchar(50)", "*", "ID");
               StudentName ("varchar(50)", "*", "*");
               StudentLastName ("varchar(20)", "*", "*");
               CourseName ("varchar(50)", "*", "*");
               StudentGrade ("varchar(50)", "*", "*");
          }
     }
     webpage Home {
          title = "Home";
          link Login ( "LoginPage.aspx");
     }
     webpage LoginPage {
          title = "LoginPage" ;
          operation login {
                destinationPage = "CoursePage";
                input CourseName;
                input Password;
          }
     }
     webpage CoursePage {
          title = "CoursePage";
          initialization CoursePageIni {
               input CourseName ;
               input Password ;
               input Teacher;
               input Department ;
               input Room ;
               sql coursePageIniSql {
                    server = "LTIPC14";
                    database = "CourseMngDB";
                    sqlString = "SELECT CourseName, Password, Teacher,
Department, Room FROM Courses WHERE CourseName = ^CourseName AND
Password = ^Password;";
               }
          }
          operation Update {
               destinationPage = "Updated";
               input CourseName ;
               input Password ;
               input Teacher;
               input Department ;
               input Room ;
               logic updateLogic {
                    sql updateSQL {
                         server = "LTIPC14";
                         database = "CourseMngDB";
                         sqlString = "UPDATE Courses SET CourseName =
@CourseName , Password = @Password , Teacher = @Teacher , Department =
@Department , Room = @Room WHERE CourseName = ^CourseName ;";
                    }
               }
          }
          operation getList {
               destinationPage = "ListPage";
               input CourseName ;
          }
     }

     webpage ListPage {
          title = "ListPage" ;
          outputs list {
               source = LTIPC14.CourseMngDB;
               id = Marks.StudentID ;
               Marks.StudentID ;
               Marks.StudentName ;
               Marks.CourseName ;
               Marks.StudentLastName ;
               Marks.StudentGrade ;
               where "CourseName = ^CourseName" ;
               del(true);
               edit(true);
          }
     }
     webpage Updated {
          link HomePage ("Home.apsx");
          link LoginPage ("LoginPage.aspx");
     }

}
```

Intermediate Code

```
APP EASYWEB_CourseMng
DB CourseMngDB LTIPC14 % 10 UNRestricted
TABLE Courses
COlUMN CourseName varchar(50) * *
COlUMN Password varchar(50) * *
COlUMN Teacher varchar(50) * *
COlUMN Department varchar(50) * *
COlUMN Room varchar(50) * *
END_TABLE
TABLE Marks
COlUMN StudentID varchar(50) * ID
COlUMN StudentName varchar(50) * *
COlUMN StudentLastName varchar(20) * *
COlUMN CourseName varchar(50) * *
COlUMN StudentGrade varchar(50) * *
END_TABLE
END_DB
WEBPAGE Home Home ltr #FFFFFF
LINK Login LoginPage.aspx
END_WEBPAGE
WEBPAGE LoginPage LoginPage ltr #FFFFFF
OP LoginPage_login CoursePage
IN CourseName
IN Password
END_OP
END_WEBPAGE
WEBPAGE CoursePage CoursePage ltr #FFFFFF
PRE
IN CourseName
IN Password
IN Teacher
IN Department
IN Room
SQL LTIPC14 CourseMngDB SELECT CourseName, Password, Teacher,
Department, Room FROM Courses WHERE CourseName = ^CourseName AND
Password = ^Password;
END_PRE
OP CoursePage_Update Updated
IN CourseName
IN Password
IN Teacher
IN Department
IN Room
BL
SQL LTIPC14 CourseMngDB UPDATE Courses SET CourseName = @CourseName ,
Password = @Password , Teacher = @Teacher , Department = @Department ,
Room = @Room WHERE CourseName = ^CourseName ;
END_BL
END_OP
OP CoursePage_getList ListPage
IN CourseName
END_OP
END_WEBPAGE
WEBPAGE ListPage ListPage ltr #FFFFFF
OUT list CourseMngDB.LTIPC14
Marks.StudentID *
Marks.StudentID
Marks.StudentName
Marks.CourseName
Marks.StudentLastName
Marks.StudentGrade
WHERE CourseName = ^CourseName;
DEL
EDIT
END_OUT
END_WEBPAGE
```

```
WEBPAGE Updated untitled ltr #FFFFFF
LINK HomePage Home.apsx
LINK LoginPage LoginPage.aspx
END_WEBPAGE
END_APP
```