Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

the way we see it

Pragmatic model-driven development using smart use cases and domain-driven design

Contents

Foreword

Model Driven Architecture (MDA) is a fractal system pattern, not only in terms of software development, but in systems development in general. It focuses on defining systems behaviors and structures at quite high levels of abstraction. Assuming that sufficient precision can be maintained, semi-automated (or even fully automated) software systems can either directly execute, or in other ways interpret, those abstractions as necessary to achieve implementation of a network structure or process.

Implementation may entail the completion of a running software system, a nuclear power plant, a business process, a semiconductor chip layout, or even something as simple as the documentation and guidelines needed to understand or upgrade a system at a later date. This remarkably simple, yet powerful, idea took some time to catch on in the software world, but the Object Management Group (OMG) began trumpeting this concept (and its related standards) in 2001. Although there was some resistance to change at first, as there always is, examples of successful MDA implementation based on OMG standards such as UML, BPMN and MOF, have proven the value and power of the MDA approach. This paper, outlining efforts that took place wholly outside the OMG community, takes that proof one step further, showcasing practical, commonsense ways towards advanced software development, quality and efficiency; proving the value of MDA. Such a mature level of adoption makes MDA safe for everyday use, and the clear and practical explication of MDA outlined in this paper makes it worthwhile reading for any engineer, even those outside the software field.

Richard Mark Soley, Ph.D.
Chairman and Chief Executive Officer, Object Management Group, Inc.

Introduction

Ever since I started my career, as a young and aspiring C programmer, I have been fascinated by the struggle that the software engineering profession has been going through in its long journey towards maturity. Programming languages came (and some went), later accompanied by ever-advancing techniques for coding, modeling, analysis, design, and accompanying toolsets. All sorts of schools of thought have emerged, some of them rigid and formal, others more agile and flexible, and sometimes their dedicated evangelists seemed to spend more time on writing books and debating with their opponents than were actually delivering projects.

If you are an evangelist, it may not be so difficult to become a crusader. And the search for the "Holy Grail" of information technology really started off with the quest for re-use and productivity: finding ways to produce more code in less time, on the one hand by building on prefabricated proven components (or even frameworks of components), and on the other, by generating code from conceptual or logical models. This introduced even more methods and tools, more methodological schools of thought, and more methodological wars. Actually, the art of generating code became so complex, that it introduced a problem space in itself, completely decoupled from the business challenges that it once aimed to address. Indeed, nowadays, we may find authors writing books about dealing with the alignment and mapping problems that they themselves caused in the first place...

And, just as the Holy Grail was never really meant to be found, it seems that we are still a long way from reaching a mature profession in which we create flexible software solutions in an industrialized and predictable fashion.

Having said this, I am pleased to confirm that there are plenty of experts out there who actually prefer to deliver solutions to real-life clients. These experts test their approaches in hard practice, learn from it, calibrate it and collect best practices from it. And then they will be willing (and able) to share these pragmatic lessons with the outside world. Sander Hoogendoorn and Robert de Wolff (both Capgemini people), and Rody Middelkoop (Avisi) are such experts. Based on the ideas and experience of Sander, an internationally-recognized thought-leader in software engineering, they have been delivering software in a highly-productive, yet down-to-earth style, together with a much bigger team of dedicated professionals. They model solutions from the user's perspective, applying smart use cases: a style of requirements modeling that is particularly repeatable and measurable, making it much easier to estimate and plan workloads. Combined with simple, yet effective domain object models, they create a foundation from which a substantial part of the code is generated. They do not generate everything: that would not be practical. But the power of simple, focused models, a large and growing reusable library of over 150 proven patterns and robust code generation can do miracles to the productivity of a project.

The writers of this paper do not seek to convert the entire world to their approach (although they may appreciate it!) nor do they claim to have a one-size-fits-all solution. But the success of their way of pragmatic, model-driven development is evident, and so it is good to share these insights.

In fact, this is highly recommended reading.

Ron Tolido
Chief Technology Officer, Capgemini Netherlands
Board Member, The Open Group

Delivering on the promise. Pragmatic model-driven development using smart use cases and domain-driven design

With the economy at a low point, organizations and project managers are clearly resetting their goals. Long-term multimillion dollar projects are being halted in favor of short, agile projects, with feasible goals and good time-to-target. To realize these stringent goals, projects are striving for lower costs and higher productivity. On the one hand, these goals might be reached by outsourcing, on the other hand, standardization, industrialization and re-use could also contribute. A long-term promise in the field of increasing productivity and re-use is to hand-craft less code, and to generate code from the project's design or model. In general, this is called model-driven development, and maybe the promise has taken just a bit too long, rather than delivering results. In this paper, we introduce a particularly pragmatic approach to model-driven development that has proven itself many times in practice and will speed up virtually any software development project.

When is model-driven development effective?

Although many alternative strategies exist towards model-driven development, we feel that a pragmatic approach in many cases works best. The more theoretical an approach becomes, the harder it may be to apply to projects. In this paper, we present such a highly pragmatic approach to model-driven development. We model and standardize smart use cases and the domain model for a project, and generate substantial amounts of code and other deliverables from these models using a straightforward tool set. This approach has proven to be very effective in many projects.

Model-driven development can be effective when a substantial part of the code and other deliverables can be generated rather than written or hand-coded. Thus, tedious repetitive work is avoided, such as creating Web forms, user interface panels, domain objects, table create statements, service interfaces and even writing use case documentation. Developers can now focus on developing business logic, or tuning the user interface. We have witnessed very high productivity in projects that apply pragmatic model-driven development. For example, a recent project (.Net) at a large Dutch government agency, was realized in 20% of the time that the customer initially estimated.

As it is also fairly easy to create new types of deliverables simply by investigating existing code and deriving a template from this, we can save time and effort in many scenarios and development environments. This style of model-driven development has proven to be successful in (partly) generating solutions for various types of projects, including ASP.Net, Silverlight, SharePoint, Visual Basic Window Forms, Java, service oriented applications (.Net and SAP), and even back end functionality.

Moreover, as code generation is based on proven templates, generated code is of higher quality than hand written code. Thus, pragmatic model-driven development also contributes to lowering the maintenance costs of delivered software. This effect is further increased because the documentation as presented in the model is directly traceable to the application's code.

Finally, but no less relevant in this approach to model-driven development, analysts and designers actively participate in delivering working software, as the models that they help to create are now directly converted into code. We consider model-driven development to be a collaborative scenario, where, especially in agile projects, even end-users can participate directly in software development. To illustrate this, we usually model the smart use cases and the domain model, that underlie the software we produce, during interactive workshops with end users, and generate working software either in these workshops or shortly after.

Modeling smart use cases in a workshop

Source: Capgemini

Code generation scenarios

Generating code from a model can be done in a wide variety of scenarios and with an even wider range of goals and deliverables to strive for:

- **Proprietary techniques**. Traditionally there are tools, such as Oracle Designer and Developer or Cool:GEN, that host both the model and the coding environment. Using these environments, high productivity can be achieved, because a lot of assumptions can be made about the target environment. In general however, these development environments deliver highly proprietary models and code.
- **DSLs**. Domain Specific Language (DSL) are textual or visual representations of custom domains for which a specific modeling language can be defined; think of Web service definitions or even a language for defining the mortgages or life assurance domain. With DSLs, again there is the promise of high productivity if and only if the target environment can be clearly defined, for instance in the software architecture and framework that are being used.
- **UML**. The Unified Modeling Language (UML) provides a set of techniques for modeling behavior and structure of applications. These standardized techniques, such as use case or class diagrams, are used in projects around the world, although in different variations and dialects. Generating code or other deliverables from a UML model is highly possible, again if the target environment is well defined.
- **Database**. With proper modeling techniques lacking, many tools take the approach of generating (parts of) applications from the database structure. Although this approach seems appealing and fast, there are limitations. The generated application will mimic the database, and as such, will be very data-centric, instead of providing support for the work processes to be automated. Besides this, in the current era of service orientation and cloud computing, many applications do not even have their own database any more.

A pragmatic approach to model-driven development

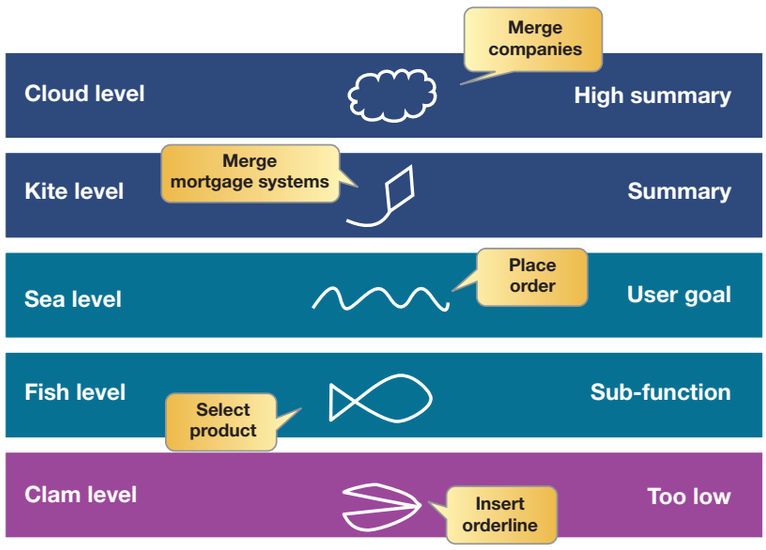
In projects that are executed using the *Accelerated Delivery Platform* (a Capgemini open knowledge platform for agile software development), a particularly lightweight, pragmatic approach to model-driven development is applied. This approach can be explained in a small number of steps:

1. **Model smart use cases**. We model our functional requirements using highly standardized smart use cases. A growing collection of over 30 standard types of smart use cases exist, making life easier when it comes to requirements analysis, project estimation, testing, but also code generation.
2. **Create the domain model**. Next, we create the domain model, containing the entities, value objects and other classes from the problem domain of the project.
3. **Decorate the model**. To be able to generate code, we combine the smart use cases with the elements from the domain model and add the appropriate stereotypes. For instance, we associate the smart use case **Search Customer** with the **Customer** class and decorate this association with the stereotype "**search**".
4. **Import model**. Then we import the model in Tobago MDA, our code generator of choice for this approach. Tobago MDA can apply text templates to the model, and generate any desired text-based deliverable, ranging from Word documents and Excel spreadsheets to the actual code.
5. **Generate code based on reference architecture**. To sort maximum effect, we base most of the application we develop on a straightforward reference architecture that is supported by a number of frameworks, both in Java and in .Net, although other technology may also apply.

Levels of use cases

In general, we model the behavior of the software to build in use cases. We define use cases as existing at different levels of granularity. Alistair Cockburn, leading authority on use cases, describes five levels of granularity.

Different levels of use cases



Source: Capgemini

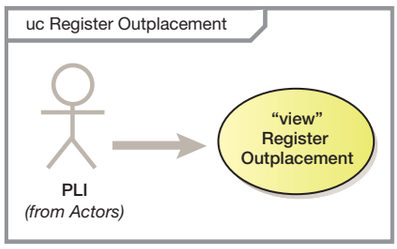
These five levels are known as:

- **Cloud level**. Very high level summary, such as **Sell books online**. Too high up to be of interest to software development projects, and far too high level to generate code.
- **Kite level**. Summary level. Again, too high level to be interesting to software development projects and code generation.
- **Sea level**. People familiar with the Rational Unified Process (RUP) should be familiar with this level of use cases, also referred to as *user goal level*. A single use case at this level describes a single elementary business process and realizes a single user (actor) goal. An example could be **Place order**.
- **Fish level**. At this level, use cases are often used and re-used as part of a sea level use case and to describe processes that, for instance, handle selecting a customer or validating credit cards with a credit card company. This level is also referred to as sub-function level and includes use cases such as **Select product** and **Pay order**. These use cases, including sea level use cases, are a good head start for generating code that handles the software's behavior.
- **Clam level**. This is where you hit rock bottom. Use cases at this level should not be defined as use cases, but rather, should appear as steps in another use case, most likely at fish level. A clear example can be the insertion of new customers in the database.

Our *smart use cases* comprise both the use cases at sea and fish level. Together, these two levels form an ideal and equal-granular technique for capturing functional requirements, and for estimating, planning, generating, building and testing your software product.

Modeling smart use cases

A single user goal level use case



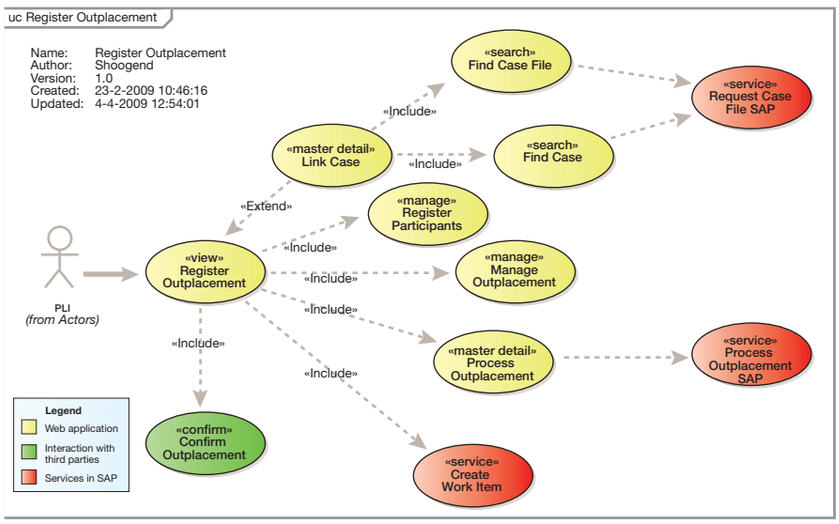
Source: Capgemini

Primarily, use cases are identified at user goal level. Here, each use case typically captures a single elementary business process. Normally, for this use case, the successful scenario is described (also called the happy day scenario), and additionally, all deviations to this scenario are described as alternative flows.

Using this approach, limited use is made of the UML use case diagram modeling technique. Requirements are captured in text, and are mostly described in Word documents. As a good example of these types of use cases, at a large international financial institution a use case called **Change Address** was written. It covered 65 pages of text, 12 screens and numerous alternative flows. As these types of use cases vary enormously in scope, complexity and size, it is quite challenging to generate code from user goal level use cases.

Alternatively, when the use cases at user goal level are identified (even better: when the elementary business processes for the project have been identified), it is possible to use the use case diagram technique to add use cases at fish level to the diagram. We have defined a number of clear guidelines on when these additional use cases apply. These include not handling more than one form per use case, handling complex calculations, handling services in service oriented projects, or even handling ETL in BI projects. The following diagram is a good example of smart use cases. Please note that this model is not a work breakdown structure, the user goal level use case has its own responsibilities.

A (smart) use case diagram, with a single user goal level use case, and a number of sub-function level use cases



Source: Capgemini

Using this approach, each of the elementary business processes is modeled out in such a diagram, with any number of actors, a single user goal level use case, and a number of accompanying use cases at sub-function level. The collection of these use cases (at both levels) is referred to as smart use cases. Again, for each of these smart

use cases, a description is written. However, these are much more lightweight than in the previous scenario and often contain only a few (or even no) alternative flows. We not only model these smart use case diagrams in traditional multi-tier scenarios, but also host service oriented and even cloud scenarios.

Stereotyping smart use cases

An important step towards generating code from use cases is the recognition of smart use case stereotypes, which describe standard behavior. This extreme requirements standardization streamlines the requirements analysis, but also enables us to define templates per stereotype that implement this behavior in, for instance, Java or C#. Currently, we have harvested over 30 of these smart use case stereotypes, including maintenance use case in **Manage, Master Detail, Select** or **Search**, such as those in the depicted diagram, but also **Collect** in BI projects, and several types of service handling stereotypes in service oriented projects.

Creating the domain model

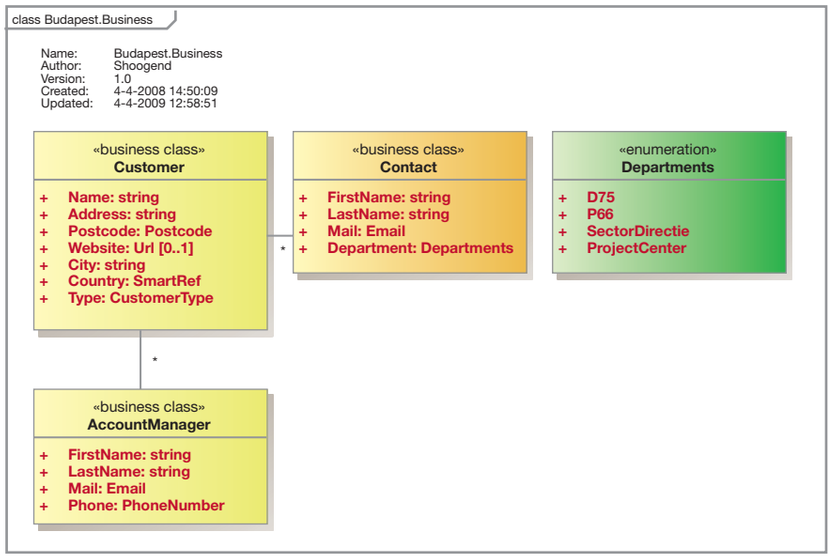
The next step towards generated code and other deliverables, is to create the domain model for the project. Next to the smart use cases, which capture the desired behavior, the domain model provides a structural view of the system. Roughly speaking, the domain model captures the key concepts, types, rules, validations, and services of the business domain, expressed in customer terminology. Furthermore, the model identifies the relationships between all major types.

In this pragmatic approach, we follow and extend the principles stated by Eric Evans in his domain-driven design paradigm. As one might expect, the domain model is expressed in UML class diagrams. Primarily, the domain model models domain objects (or entities), such as **Customer, Order, Subscription** or **Course** and their relationships, expressed as associations. Next, the properties and their types are modeled.

We consider five different categories of property types that are all applicable in different situations:

- **Basic types**. In general, people tend to model property types in basic types, such as **string, integer, boolean** or **date**. However, we have more knowledge on most property types than simply just string or integer. In this case it is better to use *value objects*.
- **Value objects**. A value object holds no identity, but reflects a value. In short, value objects are good for modeling property types that can be validated, such as **Bsn, Isbn** or **Email**.
- **Enumerations**. We model property types as an enumeration, if the property can only have a limited, *fixed* number of different values. Think of **ContractType, Gender** or **Level**.
- **Smart references**. Another pattern we apply to modeling property types is the smart reference. We have agreed to model a property as smart reference if the property can only have a limited, but possibly *changing* number of different values. Here, you could consider properties such as **Department, Prefix** or **Country** (if no other information on countries is required).
- **Associations**. If a property has a type that itself is a domain object, we model an association. Note that in UML an association is actually defined as properties on both ends of the association.

A sample domain model



Source: Capgemini

Each of these categories of property types can be used nicely in code generation. For instance, enumerations and smart references can be used to fill drop-down lists, and associations, of course, can be used to generate master-detail behavior or object-relationship mapping configuration files. Moreover, the multiplicity, role names and composite parts of associations can all be used in code generation.

Importing the model

Basically, there are three scenarios for code generation that apply UML models:

- **Modeling tool code generation**. In the first scenario, the UML modeling tool used embeds functionality to generate code. On one hand, this approach allows you to stay close to the original model, but on the other hand, it also may lock you into using only that specific modeling tool.
- **Development tool code generation**. In the second scenario, the development tool you apply embeds functionality to design the (UML) model and generate code from that model directly. You could also have a plug-in for your development tool that helps you out. In both cases in this scenario, however, you are fixed to a specific development tool and platform.
- **Intermediate code generation**. The third scenario uses an intermediate code generation tool that imports the UML model and spits out code in any way you see fit. To this end, UML supports a standard export format called XMI. The code generation then interprets the model elements and can apply templates or patterns to these elements. Here, although you more or less settle for a specific code generator, you have the freedom of choice when it comes to modeling and development tools.

Working with Tobago MDA

As we operate in many projects, Java and .Net or other (such as SAP, BI or SharePoint), and the customer can decide on the modeling and development tool set to use, the third scenario often applies quite well. In practice, we may tend to use Enterprise Architect for any of the Rational modeling tools, combined with either .Net using Visual Studio, or Java using the Eclipse platform. Although other code generators equally apply to our approach, in each of these scenarios a single, freely available code generator, called Tobago MDA, is being used. Tobago MDA was developed by the core team of Capgemini's agile Accelerated Delivery Platform (ADP) and is freely available for everybody who registers on the platform's wiki.

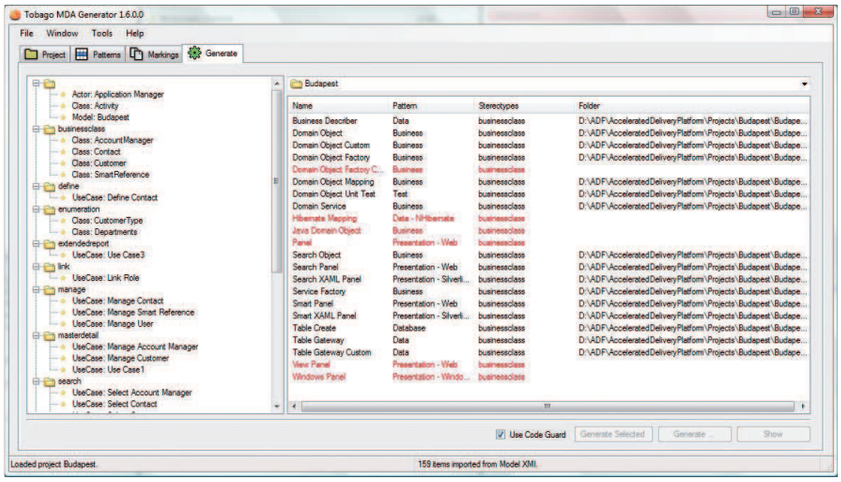
Tobago MDA, as other intermediate model-driven development tools, performs just a few activities:

1. **Define project**. When you start a new project, you will need to define its settings.
2. **Define patterns**. A pattern is a set of text-based files that, together, can be used to generate code or other deliverables. This pattern consists of a pattern definition, the main template, and possibly a small directory structure with associated templates.
   Each pattern can be applied to generate one type of deliverable, for example a domain object .NET, a domain object in Java, an Excel spreadsheet with a smart use case estimate, or a Web page, based on a single use case.
   Needless to say, the collection of patterns that is available for Tobago MDA grows quickly from the projects that apply it. Currently, this collection holds over 150 different patterns, targeted at different programming languages and frameworks, and even contains Word documents for testing and Excel spreadsheets for estimation.
3. **Import model**. Next step is to export the model from your UML modeling tool, using the standard XMI format, and import it in Tobago MDA.
4. **Generate deliverable**. Last but not least, the code generator merges a (set of) pattern(s) with a (set of) model element(s). To define which patterns apply to which model elements, stereotypes are added to the model. For instance, if a domain object is stereotyped "**domain object**", a number of patterns apply, including .Net and Java domain classes, a domain object unit test class, a table gateway, both Hibernate and NHibernate configuration files, and even a SharePoint Web part.

Although these steps describe the basic process of generating deliverables using Tobago MDA quite well, there is of course a lot more to tell. A few notes follow:

- **Combine model elements**. Different model elements can be combined to generate even more powerful structures; two associated domain object will generate relationships in code, and in the database as well.
- **Combine types of model elements**. Moreover, combining different *types of* model elements presents even more possibilities. We normally add domain objects as properties to smart use cases, and stereotype these properties for instance with "**manage**", "**search**" or "**master**". Decorating a use case Search Customer with a property of type **Customer** and stereotype "**search**", allows us to generate a fully functional Web page.
- **Remember location**. A nice feature of Tobago MDA is that the tool, when applying the same pattern multiple times, remembers the directory where the deliverables are located.

Generating code with Tobago MDA



Source: Capgemini

Changing the model without losing code

An important notion in generating code is that, due to new insights, the underlying model can and will change during the project. In most cases, these changes require a new version of the generated code. With this in mind, an important feature for code generation tools is the ability to re-generate code, without losing code that has been manually added by the developers.

Different mechanism can be applied that allow re-generation of code. Basically, most tools applying one or more of the following techniques:

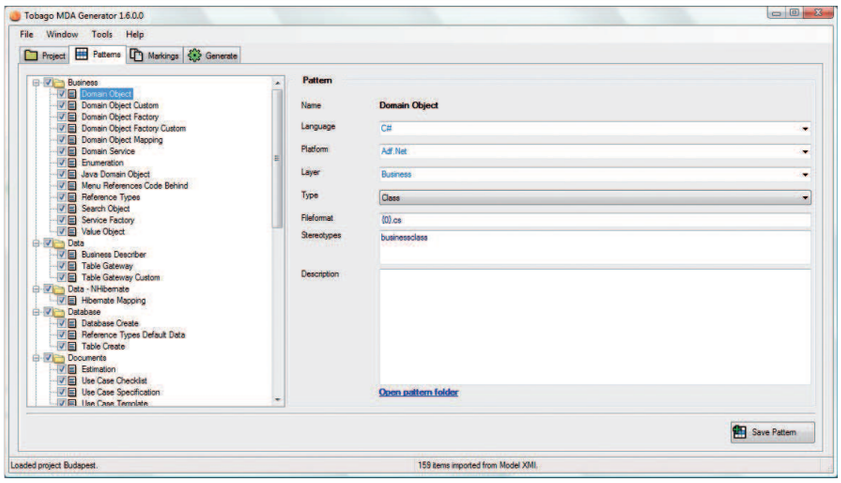
- **Partial classes**. In .Net for instance, partial classes exist. Using this construction, the generated code can be placed in one file, and the manual code in a second file, belonging to the same class.
- **Inherited classes**. In Java, where partial classes are not available, an inheritance structure can be applied. There, the manual code is added to a class that inherits directly from a generated base class. There are a number of design patterns that support this technique, such as the generation gap design pattern.
- **Marking code**. A third option, which can be applied in any programming language, would be to place code manually between special markers. When re-generating a particular class, the code generator validates all code between these markers and merges this old code with the newly generated code.

Note that none of these techniques is waterproof. There are always situations where a specific technique does not apply. For instance, none of the above works well if your target deliverables are contained in a Word document or an Excel spreadsheet.

Using patterns with Tobago MDA

There are several (model-driven) code generation tools in the marketplace. A certain number may prove to be complex in use. When we designed Tobago MDA to support our approach, our primary goal was to achieve simplicity of use. The main template in each pattern is based on a real code example. Basically, the trick is done by applying a small domain-specific language (DSL) combined with a smart directory structure.

Managing code generation patterns in Tobago MDA



Source: Capgemini

This small domain-specific language contains tags that map directly to properties in the meta model of the imported XMI. For example, the **UseCase** object in the meta model has a property called **Name**. In the template below, the tag **$UseCase.Name$** is used. When Tobago MDA merges this template with an imported use in the model, the tag is replaced by the actual name of the use case.

Next to the tags, the Tobago's domain-specific language also applies a number of operations. Some of these can be applied to the tags directly, such as **$UseCase.Name.Trim$**, which trims spaces from the use case name. This might be needed to be able to create a valid class name from the use case when generating code.

Other operations are applied inline and use a tag from the meta model as attribute. Such operations are invoked by Tobago MDA when a template and a model element are merged into new code. One such example is **Tobago.Loop()** operation in the template below. This operation loops through the attributes (or properties) of a use case, and looks for an additional template in, respectively, the **\DAOs** and the **\Attributes** sub-directories. These templates are then applied to the attributes. The resulting code is inserted into the main template.

An example of a template file is represented below, in this case generating a Java service implementation for a use case

```
package nl.$UseCase.Model.Name.Lower$.service;

import javax.ejb.EJB;
import javax.ejb.Stateless;
import nl.$UseCase.Model.Name.Lower$.model.*;
import nl.$UseCase.Model.Name.Lower$.dao.*;
import org.jboss.annotation.ejb.LocalBinding;
import java.util.List;

@Stateless
@LocalBinding(jndiBinding = "$UseCase.Model.Name.Lower$/$UseCase.Name.Trim$Service")
public class $UseCase.Name.Trim$ServiceImpl implements $UseCase.Name.Trim$Service {

<Tobago.Loop(UseCase.Attributes, "DAOs")>

<Tobago.Loop(UseCase.Attributes, "Attributes")>
}
```

Source: Capgemini

A second example template, working on a single attribute of a class in C#, is shown below. This template is typically found in a sub-directory

```
<Tobago.If($Attribute.IsNullable$,"", "[NonEmpty]")>
public $Attribute.Type$ $Attribute.Name$
{
  get
  {
    return state.GetValue<$Attribute.Type$>($Attribute.Owner$Describer.$Attribute.Name$);
  }
  set
  {
    state.SetValue($Attribute.Owner$Describer.$Attribute.Name$, value);
  }
}
```

Source: Capgemini

When does a pragmatic model-driven development scenario fit?

We refer to the described approach as a highly pragmatic one. It is not difficult to create new patterns, even for people who are new to the platform. You simply start with existing pieces of code that represent the type of work you want to generate. Next, you identify which parts of this code (or other deliverable) should be replaced by tags. And you then pull out the repetitive parts, such as properties or operations, add in Tobago's looping or conditional constructs and move the repetitive parts into sub-directories. That's it. You are now ready to generate code and import that code into your application. This whole process takes about 15 minutes and can be repeated for all re-occurring elements in your software architecture.

Glancing over this scenario makes you realize that this type of code generation is easy to achieve, given some minimal settings:

- **UML**. You will need to know *how* to model in UML; which modeling techniques to use, and how to use them. Here, we apply and combine two of the most basic modeling techniques available, the use case diagram and the class diagram.
- **Software architecture**. You will need to know what code to generate. This largely depends on the underlying software architecture; which layers it is comprised of; and which layer supertypes appear in those layers. In general, it is these layer supertypes that are transformed into code generation patterns. Think domain objects, repositories, factories, table create statements, controllers, views...
  Besides code generation, having a sound software architecture in place is a must in building good and maintainable software.
- **Frameworks**. To speed up development further, it is recommended to use standard vendor or open source frameworks, such as Spring, Enterprise Library, ADF, Castle or NHibernate. Mind you, it is considered good practice to have these frameworks match your software architecture, and not the other way around.

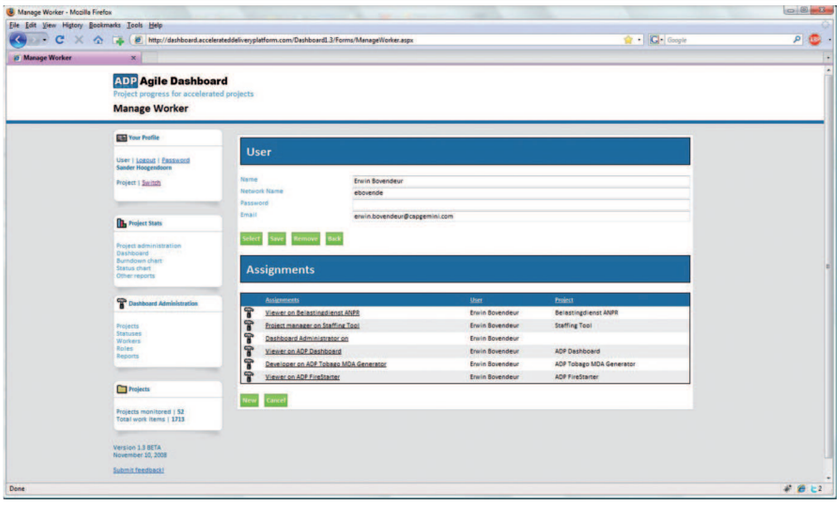
Given these minimal settings, there are hardly any limitations to what you can do with this technology. Whether you are generating an Excel spreadsheet summing up use cases and estimates, Word documents with test scenarios, complete functional domain layers or actor-based side bar menus, it is fairly easy to accomplish. A quick example follows.

Generating .Net at Capgemini

Projects that are run using accelerators from Capgemini's agile Accelerated Delivery Platform quite often rely on the Agile Dashboard to monitor project progress. This online dashboard is developed as an ASP.NET Web application, using a multi-tier software architecture in combination with a number of custom frameworks (such as ADF). The dashboard features managing projects, teams and work items.

The form presented here is a typical master-detail scenario that shows the form **Manage Worker**, which shows a worker and his assignments in project. The form was generated from a smart use case with the same name that was associated with two domain objects **Worker** and **Assignment**.

A generated master-detail scenario



Source: Capgemini

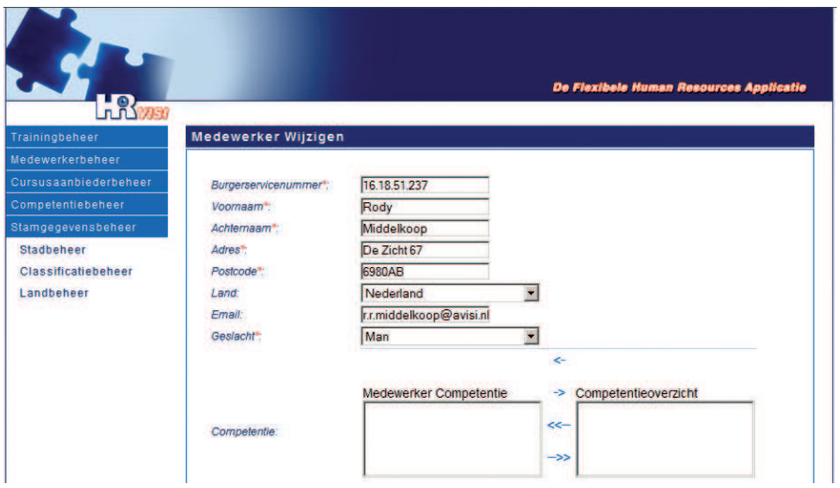
To once more demonstrate the usefulness of this style of model-driven development, note that the first operational version of the Agile Dashboard was modeled during a two-hour workshop, before being generated from the model and rounded up in a mere four hours.

Generating Java at Avisi

At Avisi, a Java software development company based in Arnhem, Tobago is applied to kick-start software development for a human resources (HR) application that is to be used to manage competences, employees, administrate agreements with employees, etc. The application is generated and built on a Java open source stack, containing the following components: Freemarker, Sitemesh, Struts2/XWork, Spring, JPA/Hibernate, MySQL and JBoss. Using this stack, the application is generated and deployed on JBoss and can be used instantly.

The form presented is generated from the appropriate use case **Add Employee**, which was combined in the model with a domain object **Employee**.

A generated form for adding employees



Source: Avisi

Note that the **Country** field (**Land** in Dutch) is presented in a drop-down list that was filled using the associated smart reference **Country** on the domain object **Employee**. Moreover, all the required fields get validated through regular expressions that were included in the UML model. Furthermore, an employee can add competences and apply for training. This particular form elaborates on the associations between **Employee**, **Competence** and **Training** in the domain model.

As simple as possible, but not simpler

In our experience, applying a highly pragmatic approach to model-driven development, like the one described here, can be very effective. Although many alternative strategies towards model-driven development exist and may work fine, we feel that a pragmatic approach often pays off very well. One thing is certain: the more theoretical an approach becomes, the harder it may be to put it into practice. Here, we mainly model smart use cases and the domain model, and decorate that by combining the two, and adding stereotypes.

Even though some vendors show particular interest in adding code generation facilities to their modeling or development environments, in this case we specifically chose to use an independent intermediate tool, such as Tobago MDA, to be able to target multiple platforms with the same strategy. All patterns that we collected for use with Tobago MDA come from live code, and thus apply proven technology. Generating code from such patterns raises the quality of the software at a fast pace. Furthermore, it increases traceability between design and code, and lowers the costs of application maintenance.

We do not expect code generation to be perfect, but it does not have to be. Code and other deliverables are generated to avoid having to do tedious, repetitive work, such as Web forms, panels, domain objects, table create statements, service interfaces and even documentation. This allows developers to focus on business logic, or on fine tuning the user interface. Because it is fairly easy to create new templates simply by investigating existing code, this approach saves time and effort in a varied number of scenarios and development environments. It has proven successful in (partly) generating solutions for various types of projects, including ASP.Net, Silverlight, SharePoint, Visual Basic Window Forms, Java, service oriented applications (.Net and SAP), and even back end functionality.

It is an approach that adheres perfectly to one of Albert Einstein's famous quotations: *"Everything should be made as simple as possible, but not simpler."*

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

www.capgemini.com

About Capgemini and the Collaborative Business Experience™

Capgemini, one of the world's foremost providers of consulting, technology and outsourcing services, enables its clients to transform and perform through technologies. Capgemini provides its clients with insights and capabilities that boost their freedom to achieve superior results through a unique way of working, the Collaborative Business Experience™. The Group relies on its global delivery model called Rightshore®, which aims to get the right balance of the best talent from multiple locations, working as one team to create and deliver the optimum solution for clients. Present in more than 30 countries, Capgemini reported 2008 global revenues of EUR 8.7 billion and employs over 90,000 people worldwide.

More information is available at **www.capgemini.com.**

Authors:

Sander Hoogendoorn,
Principal Technology Officer, Capgemini

Rody Middelkoop,
Senior Technology Consultant, Avisi

Robert de Wolff,
Senior Technology Consultant, Capgemini

For further information contact:

Sander Hoogendoorn
Principal Technology Officer, Capgemini
The Netherlands

Email:
sander.hoogendoorn@capgemini.com

Web:
www.acceleratteddeliveryplatform.com
www.smartusecase.com
www.sanderhoogendoorn.com