Small is Beautiful

Building a flexible software factory using small DSLs and Small Models

Jos Warmer
Partner, Ordina
jos.warmer@ordina.nl
Modeling Maturity Levels

- MML 0: No specification
- MML 1: Textual Specification
- MML 2: Text with Models
- MML 3: Models with Text
- MML 4: Precise Models
- MML 5: Models only
MDA Overview

Platform Independent Model

PSM SQL ➔ SQL-Business ➔ Business ➔ EJB - ASP ➔ PSM ASP

SQL Code ➔ Business Code ➔ ASP Code
Core MDA Concepts

Abstraction

Automation
MDA is a trademark of the OMG

Main model types:
- PIM: Platform Independent Model
- PSM: Platform Specific Model
- Code: well … , just code

Uses OMG standards
- MOF: Meta Object facility
- UML: Unified Modeling Language
- OCL: Object Constraint Language
- QVT: Queries, Views, Transformations
MDA Experience

- All (acclaimed) MDA tools are disappointing 😞
  - Lack of flexibility (All or nothing / not adaptable)
  - Cumbersome support for re-generation
  - Not enough code generated
  - Lack of integration in IDE
  - Standards appear too slow (e.g. QVT)
    - And might not even work very well …
UML As a Basis For MDA?

- UML is not suitable for large scale use
  - Multi user modeling support is horrifying
  - Version control of UML models is complex

- UML is not suitable for code generation
  - It's too big
  - It's too complex
  - It often does almost what you need, but never quite

- UML assumes one huge monolithic “Main Model”
- And history has shown monolithic solutions do not scale up
Because MDA is a trademark of the OMG …

- I rather speak of **Model Driven Development**
- or **Model Driven Software Development**
- or **Model Driven Software Engineering**
- or **Model Driven Engineering**
Domain Specific Languages
A DSL

- Contains special concepts from the domain and is very powerful within the domain (and useless outside)
- Can be defined “on the fly”
- Is executable
  - Often through code generation
  - Sometimes through interpretation
  - Sometimes a hybrid of both
- A DSL often comes with a Domain Specific Framework or a Domain Specific Engine
Why Domain Specific Modeling

- Tailor the modeling language to the goal
- Much more powerful than general purpose language
- Users work with their concepts they understand
- Users only get what they need, no more, no less
  - less complex for users
  - Less complex for language designers
  - Less complex for language support tool developers
  - Less complex for code generators

- Diversity in the IT and business world
- Flexibility to follow the speed of changes
Why not DSL’s earlier?

- Domain Specific Languages have been around before
- Building a language isn’t that hard
- Building tool support is very hard
- Much work for textual DSLs
- Even more work for visual DSLs
DSLs based on UML

- Use UML as platform for creating Domain Specific Languages
- Define a UML profile
  - Find the concepts in your DSL
  - Find UML elements that are close to your concepts
  - Define stereotypes for those UML elements
  - Add tagged values for additional properties
  - Define OCL constraints to define what a correct model is
- Almost all UML based MDA tools work like this !!!
  - Generating code from UML is a fairy tale
- Why ?
  - Because building visual editors is (was !!!) time-consuming
  - I.e. reuse of UML visual modeling tools
UML is a Complex Language

- Creating UML profiles is complex
  - i.e. needs to understand the UML2 metamodel 😞

- Defining the constraints on a profile is even more complex
  - i.e. needs to understand the UML2 metamodel 😞

- “Configuring” UML tools to validate the profiles is complex
  - Most tools do not support it
  - Some tools support it by their own proprietary scripting language
  - Some tools support OCL

- Code generation is complex
  - i.e. needs to understand the UML2 metamodel 😞
DSL Workbenches

- **DSL Designer:**
  - Definition of meta (or domain) model
  - Definitions of visual appearance (e.g. boxes and lines)
  - Definition of validation constraints
  - Definition of code generation templates

- **DSL Workbench automatically generates**
  - Generation of visual editor
  - Execution of code generation
  - Storage of DSM’s in files
  - Integrates into development environment

- **E.g.** Eclipse GMF, Microsoft DSL Tools, MetaEdit, GME
UML versus DSL’s

- UML is a general purpose language
- UML always has one big interconnected model for the complete application (The so-called “Main Model”)
- UML is closed

- A DSL is a special purpose language
- DSLs uses multiple loosely coupled models for one application
- In one application multiple DSLs may be used
- DSLs are open
MDA versus DSL’s

- MDA ➔ abstraction
- MDA ➔ code generation
- MDA assumes that not everything can be generated
- MDA ➔ you can define your own language
- MDA is owned by the OMG

- DSL ➔ abstraction
- DSL ➔ code generation
- DSL assumes that not everything can be generated
- DSL ➔ you define your own language
- DSL is owned by no-one

- MDA is most often associated with UML

- DSL is flexible
How to build DSLs

Experience from building SMART-Microsoft
Choose your environment

- Microsoft's answer to MDA and UML is their own flexible (meta-) modeling environment:
  - Domain Specific Language Tools
  - Solves the problems with UML 😊
  - Available as tools
  - Create your own modeling languages
  - Generation of visual editor within Visual Studio

- Part of the Microsoft Software Factory initiative
A Well Defined Process

- Process
  - What
  - Refine and divide
  - Define
  - Build
  - Test
  - Use

- Best Practices Summary
What ➔ Goals of using DSLs?

- Generate code that runs
- Code always conform a reference architecture
- Being able to always re-generate
- Modeling must be less work than coding
What does it need to generate

- Step 0: define your domain
- Step 1: decide on the target architecture
- Step 2: write the code that needs to be generated by hand using a reference application
- Step 3: test and review the reference application
- Reference application is divided into:
  - Part of code is static
    - ➔ put into framework
  - Part of code is dynamic
    - ➔ generate code from a DSL
    - ➔ or … keep writing the code by hand

- Tradeoffs to be made:
  - Complex framework needs less generated code
    - What we are used to when writing code by hand
    - Case distinction inside the framework
  - Less complex framework needs more generated code
    - Case distinction by the code generator
Define

- Define the concepts that you need in your DSL to generate the required code
  - Focus on concepts above programming language level
  -Storyboard the models
  - Keep a DSL small and simple
  - Keep models for a DSL small
  - Allow separate models to have references
  - Allow validation of references
  - …

- For each concept in your DSL:
  - Identify the code that needs to be generated
  - Adjust the concept (and its properties) to ensure you can generate this code
Class MyClass
{
    public string Hello()
    {
        return "Hello world";
    }
}

CREATE TABLE MyTable
FIELD1 int
FIELD2 varchar(50)
Build

- Develop the domain model
- Develop the presentation and tool model
- Write the code generation templates
- Identify Code generation patterns
- Never change generated code, only extend it
  - Design extension points in generated code
  - Use patterns like abstract base class + partial concrete subclass

- Develop iteratively
  - Adjust DSL concepts, generated code and framework based on the understanding you get
Class MyClass
{
    public string Hello()
    {
        return "Hello world";
    }
}

Partial Class MyClass
{
    public string YourTurn()
    {
        return "Your world";
    }
}
We found that the DSL Tools were not enough … we added:

- **NDIP = Non persistent Dsl Information Provider**
  - To allow references between DSMs
  - Validation
  - Intellisense / picklists
  - Refactoring
  - Change propagation

- **Automatic code generation**
  - DSL Tools only has a “run all templates action”
  - We incrementally generate code (per DSM) when the model is saved

- One template per model type (not per model!)
Class MyClass
{
    public string Hello()
    {
        return "Hello world";
    }
}
Test

- Rebuild the reference application with the DSL and test it
- Do this iteratively for each DSL
Use

- DSLs are developed to be used by other developers
- Develop training material / workshops / walkthroughs
  - For the architecture
  - For the components used in the architecture
  - For the DSLs
  - For the extension points in the generated code
- Organize project support
  - First project needs it preferably by the DSL developers
  - Life Wiki for Q&A during the project
- Evaluate
  - Find good points
  - Find bad spots
Domain Specific Languages

Lessons Learned
Small DSLs & Small Models

- Multiple ‘independent’ DSL’s
- Multiple ‘independent’ models per DSL

Typical development situation: multiple models for each DSL.
References Between Models

- References always by name

Web Scenario Model 1

- <<web scenario>> Order Product
- <<action>> User gives name and address
- <<web scenario reference>> Select Product
- <<action>> Finalize order

Web Scenario Model 2

- <<web scenario>> Select Product
- <<action>> Show List of products
- <<action>> Select a product
Model Interface

Model 1

Model 2

Model 3

NDIP

Support for
- Cross model validation
- Intellisense in DSL
- Code generation
- Refactoring
- Propagation of model changes
Model Interface

Web Scenario Model 1

Web Scenario Model 2

Business Class Model 3

request

export Info 1

read Info 1

export Info 2

read Info 2

export Info 3

read Info 3

NDIP
Rules of Thumb

- Keep a DSL small
  - I.e., the number of concepts must all fit on one toolbar
  - Assume references are needed

- A DSL doesn’t live standalone, it is part of a more complex world,
  - Define what information the DSL needs from other components
  - Define what information from the DSL Model should be shared
  - A DSL is a component, use information hiding

- One DSL does not solve all problems (it’s small !)
  - Assume that you will end up with multiple DSL’s

- Model is the unit of version control, multiuser access, etc.
  - Reuse your source code control system for Models
Rules of Thumb

- Everything in a model is used for code generation
  - Not just documentation, same status as source code

- Modeling must be less work than coding
  - You will need manual coding as well

- Models are leading: never touch the generated code
  - Handwritten extensions through defined extension points
  - Design the generated code to support extension

- Perform code generation per model
  - No long waiting times for “Generate All”
  - Ensures DSL remains standalone unit
View DSM (Domain Specific Model) as Source Code File

- A DSM is the unit of multi-user access
- A DSM is the unit of version control
- Use familiar and proven version control systems for code
- References by name only
- Refactoring like source code
- DSM is unit of reuse
- DSM is source for nightly builds
- The DSM is always leading
- Code generation per DSM
- Project tasks per model
- Higher developer acceptance
- Etc. etc.
Domain Specific Languages

Future
How many levels are useful?

- Extension both horizontal and vertical
Generation of Models from Models

- Business Domain DSL
  - Business Domain Model
  - Business Domain Model
  - Business Domain Model

- Web Scenario DSL
  - Web Scenario Model
  - Web Scenario Model
  - Web Scenario Model

- Data Contract DSL
  - Data Contract Model
  - Data Contract Model
  - Data Contract Model

- Service DSL
  - Service Model
  - Service Model
  - Service Model

- Generated artefacts in grey, handwritten in yellow

ASP.NET, C# Code, Config files, Other artefacts, XSD, C# Code


Created at UCS in grey, handwritten in yellow
Domain Specific Languages

Advantages
Consistency and Quality in Architecture

- **DSL → Code transformation is guided by the underlying architecture patterns**
  - Architecture is consistent & follows the architectural rules

- **DSL → Code transformation is consistent**
  - Code is always 100% architecturally consistent and correct

- **Manual additions in the code**
  - Must always fit within the generated structure
  - Never change any generated code

- Working ‘under architecture’ becomes reality for the first time
Fast time-to-market

- Direct transformations of DSL $\Rightarrow$ Code
  - First version can be created quickly: jump start of project

- Continuous code generation
  - Agile and iterative methods are facilitated.
  - New versions can be delivered fast

- And this all while keeping quality and guaranteed architectural consistency!
Higher productivity

- Changes at business level can be implemented easily through models and code generation.
  - Regenerate code from DSL model
  - All manual coding remains when regenerating
  - Models keep their value
  - Model driven agile and iterative methods are facilitated

- … while quality and architectural consistency is guaranteed!
Surviving Technology Changes

- Model is technology independent
- Transformation to Code introduces technology
- Each technology change means
  - Model remains completely useable
  - Write new transformation from DSL to new technology
  - Write new transformation for bridge between new and old technology
  - ... that’s all

- Flexibility ➔ change is the only constant factor in the IT world
Productivity of DSL’s

- **Presentation**: 73%
- **Data Contract**: 27%
- **Service / Process**: 27%
- **Business**: 27%
- **Totaal**: 73%

- Custom
- Gegenerereerd
Release Management for the DSLs

- **DSLs will evolve**
  - New versions of components used in the framework
  - New features
  - Changes in code generation
    - To allow for more flexible extension points
    - To use new framework components
  - Changes in the DSM domain model
    - New concept discovered

- **Define a release strategy**
  - How often will new releases be done
  - Backwards compatible or not
  - When to update running projects to new releases or not?