Thanks for attending my presentation about model-driven development. I hope you gained some insight into how MDD is applied successfully. Michael Eichberg will make the slides available to you. Feedback of any kind is highly appreciated, just send me an e-mail: martin.girschick@capgemini.com. You can also send me a photo/scan of the contact sheet or if you have any others questions concerning Capgemini.

As mentioned, I organize student events and workshops. Upcoming next is an "after work" event on 20th of March, where you can meet young colleagues and talk about their work at Capgemini. Details will be posted on https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewforum.php?f=293. If you are interested you can also just send me an e-mail.

Model Driven Development in industrial practice

Dr. Martin Girschick
February 2018
• Study and PhD at TU Darmstadt
• Since 2008 working for Capgemini
• Projects in Public Sector, Telecommunications, Finance, Logistics
• Different Roles: Developer, Architect, Quality Assurance, Project lead, Consultant, …
• Lead of german Capgemini Community for model-driven development
• University Relations for TU Darmstadt
Capgemini in numbers

- ~200,000 people worldwide
- Over 120 different nationalities
- More than 40 countries

Revenue worldwide for 2016: 12,5 M€

2017: 50th anniversary
What do you know about MDD?
Five arguments against model driven development

- Models are models, real life is different
- No-one knows, how to do it (right)
- Performance woes
- All in and no way out
- High effort with low return

This presentation refutes these arguments and gives examples on successful use of MDD.
Standardization and formal specification helps to solve complex problems.

- **Advantages of MDD approach**
  - Reusability, reproducability
  - Modeling is closer to problem domain
  - Important things get into focus

- **Disadvantages of manual approach**
  - Less formality and precision
  - Long development cycles
  - High maintenance efforts
  - High dependency on runtime environment

---

**Processes**

- System design
- Manual
- Runtime environment
- Software
- Formal metamodel
- Reusable asset
- Project specific asset
Model Driven Development

Models are models, real life is different.

**Top-Down**
- “Full-scale” MDD project
- higher setup effort
- high customer involvement

**Closed System**
- Vendor controlled runtime.
- Good tool support.
- Integration platform, often with analytical tools.
- Examples: SAP, BPM-Suites, ...

**Bottom-Up**
- selected areas are modelled and generated
- often heterogeneous tool landscape
"Model driven development uses formal models to generate derived artefacts." – So what does that mean?

<table>
<thead>
<tr>
<th>The <strong>generated artifacts</strong> can be models or source code</th>
<th>or simply data in the same or another format as the input model</th>
<th>so documents, XML or images can be created as well.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>model</strong> is a primary development artefact</td>
<td>but it is not the only one</td>
<td>because not everything can be put into the model.</td>
</tr>
<tr>
<td>A <strong>formal metamodel</strong> is required to generate artefacts</td>
<td>but the model is not limited to graphical representations</td>
<td>because text quite often allows for more concise representations.</td>
</tr>
<tr>
<td>The <strong>modeling language</strong> should be chosen carefully</td>
<td>and is not limited to UML</td>
<td>Because domain specific languages are often suited better.</td>
</tr>
</tbody>
</table>

**Models are models, real life is different**
Don’t be afraid of metamodeling.
The concepts might sound strange, but they help to build a formal basis.

**an example**

- MOF
- UML
- class diagram
- class instances

**the model**

- $M_0$: data
- $M_1$: model
- $M_2$: meta model
- $M_3$: meta-meta model

**conforms to**

- $M_1$ conforms to $M_2$
- $M_2$ conforms to $M_3$

**defines**

- $M_1$ defines abstract syntax and validation and transformation rules
- $M_2$ describes concrete syntax

**is instance of**

- $M_1$ is instance of data
Let’s take a look at a few example…

Domain specific languages are tailored towards specific applications.
The UML can be extended in two ways.

- The MOF meta-metamodel is used to define the Unified Modeling Language.
- The UML consists of different viewpoints on software systems (e.g. class diagrams).

- UML profiles offer a lightweight extension of UML using stereotypes and tagged values.
- Heavyweight extensions, which add new graphical objects are possible as well, but there are nearly no tools available.

- The OMG propagates MDA as a paradigm for model driven development using UML profiles.
Defining the right domain specific language is the key to success with MDD.

- In some cases, existing languages are sufficient but often defining your own languages provides greater flexibility and can be tailored to the needs of the customer.

existing languages → custom made DSLs

- Extensive tool support for custom DSLs is already available:
  - Languages with integrated DSL support (e.g. Scala, .NET/LINQ)
The type “ID” serves as an identifier for the type system
The type “INT” is used for integer type attributes.
You can use “|” to separate expanding literals, e.g. a: b | c;
The rules not only define the abstract syntax (metamodel structure) but also the concrete syntax (how actual model instances look like).
Choosing the right platform is important.

- Today’s software systems consist mostly of standard components, which have to be configured correctly.
- The actual business logic is only a minor part.
- The idea of MDD is not to generate the complete application but only parts, which are not part of the platform.
- In addition, configuration for the platform can be generated!

Source: From HTTP up to JDBC as a picture
The multistage process from model to code.

- **Modelling**
  - Enterprise Architect
  - XML Editor
  - UML Model (XMI)
  - XML Document

- **Generating**
  - Model reader (XML2Ecore, XML2Ecore)
  - Model instance (ECore)
  - Templates engine XPand
  - Transformation XTend

- **Implementing**
  - Generated code (Java, OR-Mapping, Spring configuration)
  - Manually written code
  - Target platform (Java, Hibernate, Spring, ...)

**Language**
- Input model
- MDD tool
- Output model

**Eclipse**
Up to 50% can be generated on certain platforms.
Four examples illustrate the potential for MDD.
“Software factory” for retirement provisioning (german: Altersversorgung)

Customer A

Customer B

Customer C

- Base- and business entities
- In-/Exkasso
- Rules and Formulas for provisioning calculation
- Tariff- and Account-Management
- Printing
- Historation component
- Support for Multitenancy

Generated code for three different instances

- Persistence (Hibernate)
- Transaction (Hibernate)
- DI-Framework(Spring)
- Middleware (Webservices, Axis)

High effort with low return

Framework

Platform

Java
Quasar-Client
Security (Acegi)
Example: Mapping from Specification to Design

Fachklassendiagramm

Component-Package Mapping

Transformation Fachklassenmodell (A→D)

Komponente: Person

Package: ~.fallgrunddaten.person.personen

Primary keys

Inheritance

Compositions and cardinalities

Name conventions

Person_in_BG

Krankenversicherung_Person

Gesetzliche_KV

Krankenkasse

KVGesetzlich

KV_GesetzlichData

KVGesetzlich

Person_KV

PersonInBG

PersonInBGData

AbstractKVPerson

AbstractKVPersonData

PersonInBG

PersonInBG

InsurancePerson

InsurancePerson

AbstractKVInsurance

AbstractKVInsuranceData

Inheritance

Compositions and cardinalities

Name conventions

Primary keys

Package: ~.fallgrunddaten.person.personen

Component-Package Mapping

Transformation Fachklassenmodell (A→D)

Komponente: Person

Package: ~.fallgrunddaten.person.personen

Primary keys

Inheritance

Compositions and cardinalities

Name conventions

Primary keys

Inheritance

Compositions and cardinalities

Name conventions
Simplification of the generation process

Old generation pipeline

Data in EA model
Run check scripts in EA
EA to XML
SVN Update target workspace
• Takes ~30min

Convert XML to UML
Run the generator
Check errors
Commit all files to SVN.

• Takes 1 min
• Takes 45 min

In the meantime

Simplified new generation pipeline

Make changes to data in Java model
Run the generator
Model Validation errors, if any
Commit all files to SVN

• Takes 10 min

No-one knows, how to do it (right)
More examples from a large project in the public sector.

<table>
<thead>
<tr>
<th>Service Gateway Generator</th>
<th>Document Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology: Groovy, Velocity, Ant</td>
<td>Technology: Enterprise Architect, .NET-Application</td>
</tr>
<tr>
<td>Copies a parameterizable project template using ant.</td>
<td>The specification is modelled in the UML tool Enterprise Architect.</td>
</tr>
<tr>
<td>Generates code for authorisation, dispatching and error handling using wsimport and a groovy script, which parses a WSDL and control the velocity template engine.</td>
<td>Conventions for modelling include certain stereotypes and other aspects.</td>
</tr>
<tr>
<td></td>
<td>A COM-based application reads the model from EA and controls Word to create a specification document.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Business Process and Rule Engine</th>
<th>Model Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology: JBoss JBPM and Drools, MS Excel</td>
<td>Technology: Enterprise Architect</td>
</tr>
<tr>
<td>Validation rules for data are written using Excel.</td>
<td>Code generation using proprietary EA template language</td>
</tr>
<tr>
<td>Macros and a converter creates native drools rules, which are parsed and startup of the application.</td>
<td>Model transformation from specification to implementation model using so called “MDA style transformations” (also EA proprietary).</td>
</tr>
<tr>
<td>Business processes are modelled using an Eclipse-based graphical editor, which creates XML.</td>
<td>Parts of the transformation script are generated using formulas and macros within an excel sheet.</td>
</tr>
<tr>
<td>A JBPM tool creates an SQL script from it.</td>
<td></td>
</tr>
</tbody>
</table>
Business Rule Engine: JBoss Drools

- uses RETE algorithm to boost execution performance
- Runs on application server (e.g. Tomcat)
- Library approach
- Open source
- Homepage: http://www.jboss.org/drools
- Current Version: Drools 5
  - Drools Guvnor (BRMS/BPMS)
  - Drools Expert (rule engine)
  - Drools Flow (process/workflow)
  - Drools Fusion (event processing/temporal reasoning)
  - Drools Planner

The Rete algorithm is an efficient pattern matching algorithm for implementing production rule systems. ...The word 'Rete' is Latin for 'net' or 'comb'. The same word is used in modern Italian to mean network. Charles Forgy has reportedly stated that he adopted the term 'Rete' because of its use in anatomy to describe a network of blood vessels and nerve fibers.

Performance woes
## Success factors for MDD projects

<table>
<thead>
<tr>
<th>Working Knowledge Management</th>
<th>Customer acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Consistent tool chain</td>
<td>▪ Models are accepted artifacts</td>
</tr>
<tr>
<td>▪ Community support</td>
<td>▪ Customer are actively involved in modelling</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distinct team roles</th>
<th>Early planning and project initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Permanent team members with detailed knowledge of generator chain and modelling environment</td>
<td>▪ Consider MDD during bid phase</td>
</tr>
<tr>
<td>▪ Capable offshore team</td>
<td>▪ Early setup of tool chain with competent team</td>
</tr>
<tr>
<td></td>
<td>▪ MDD is not limited to the construction phase, consider all project phases</td>
</tr>
<tr>
<td></td>
<td>▪ Think about later: Migration, Merging, Lifecycle</td>
</tr>
</tbody>
</table>
Let’s revisit the five arguments against model driven development:

- **Models are models, real life is different**
- **No-one knows, how to do it (right)**
- **Performance woes**
- **All in and no way out**
- **High effort with low return**

With *organizational structures* in place, an *experienced team* and *early setup* of a project *tailored tool chain* *MDD* provides several advantages over “classical” development.
„If you are interested in Capgemini, don’t hesitate to contact me or hand in the contact form!“
Appendix
The abstract syntax – defining the right metamodel
Distilled from Markus Völter: “MD*/DSL Best Practices”

- **Understand** the business and the language they use. Take a look at the documents they write.
- Ensure that it can **properly be translated to code** (or whatever derived artefact you want to create)
- Think of **modularisation** and **viewpoints** (or even annotation concepts) to cover certain aspects of the complete model. Find well defined connection points between them, make sure those “interfaces” are unidirectional and simple.

- **Limit expressiveness**
  - Stick to declarative languages.
  - Often, DSLs can be categorized in two types:
    - **customization DSLs** provide a vocabulary to express facts
    - **configuration DSLs** provide values to parameters, they are often simpler to design but less expressive
  - The languages is the “what”, the generator creates the “how”. Domain experts often only know the “what” but not necessarily the “how”.
  - If the language needs to be turing complete, a DSL might not be a good idea. Define a proper API instead or provide hooks in the generated code to add expressiveness in a standard programming language. Internal DSLs or languages which can be properly extended might be an alternative as well.

The concrete syntax – Notation matters!
Partly distilled from Markus Völter’s paper.

- Stating the obvious (or maybe not)
  - Stick to **existing notations**, if possible.
  - Make sure, that **appropriate tooling** is available.
  - **Textual or graphical** - choose carefully! Sometimes mixed forms or separate viewpoints (with the same or a different representation) help. Think of the different user groups.
  - Provide proper defaults, try to make models small.

- **Textual notations**
  - Appropriate tooling is often easier to find (e.g. proper editors, multiuser-support, build integration).
  - Not limited to structured text. Tables or forms are possible as well.
  - It’s often easier to structure large models using text, beautifying can be automated.

- **Graphical notations**
  - Might be necessary, if relationships exist (e.g. dependencies, flows, sequencing).
  - Not all cases require a specialized editor – providing templates and convention might be enough.
  - Specialized tools often offer GUI prototyping to create an appropriate editor (e.g. Eclipse-based GEF-Tools).
The semantics are encoded in the generator or interpreter.
However, the language user needs a description as well!

- Keep generated code **separate** from manually written code.
- Some systems offer “**protected regions**”, which are retained upon regeneration. Refrain from using them, uses appropriate design patterns and APIs instead.
- User **versioning** for primary artefacts, only (models, transformation rules, manually written code).
- Generate **beautified code** (higher acceptance, easier debugging).
- Generate **templates** as a basis for manually written code. Do that only once.