

Model-Driven Software Development

Introduction and Overview



www.mdsd-buch.de



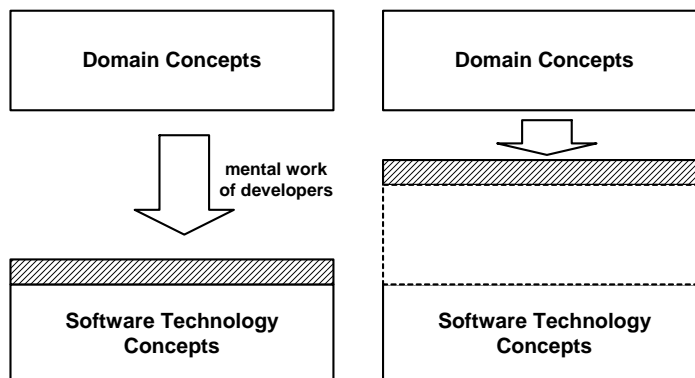
www.mdsd-book.org

Markus Völter
 voelter@acm.org
 www.voelter.de



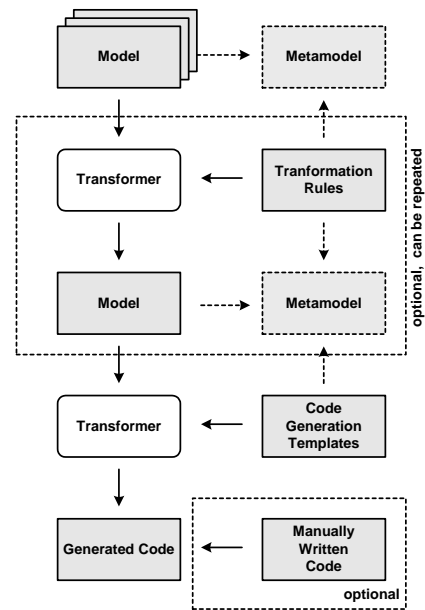
Domain Driven Development

- Domain Driven Development is about making software development more **domain-related** as opposed to **computing related**. It is also about making software development in a certain domain **more efficient**.



How MDSO works

- Developer develops **model(s)** based on certain metamodel(s), expressed using a DSL.
- Using **code generation templates**, the model is transformed to executable code.
 - Alternative: Interpretation
- Optionally, the **generated code is merged** with manually written code.
- One or more **model-to-model transformation steps** may precede code generation.



Reasons for DDD

- Software Development is too **complex** and too **expensive** (now, this is a really new finding ☺) ...
 - ... because:
 - There is **too little reuse**
 - **Technology changes** faster than developers can learn
 - Knowledge and practices are **hardly captured explicitly** and made available for reuse
 - Domain experts cannot understand all the **technology stuff** involved in software development
- DDD aims at attacking some of these problems. We shall see how on the following slides.

What is a „Domain“

- A **definition** could be:

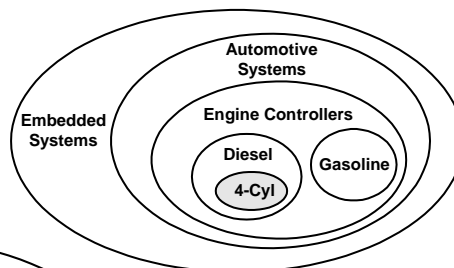
A domain is a bounded area of knowledge or interest.

- Examples (from the world of Software) include:
 - eBanking
 - Embedded Software
 - Web-Based eBusiness Applications
 - Control Software for 4-Cylinder Diesel Engines
 - Astronomical Image Processing Software
- Domains can have various „**scopes**“ as well as various „**flavours**“ – see next slides.

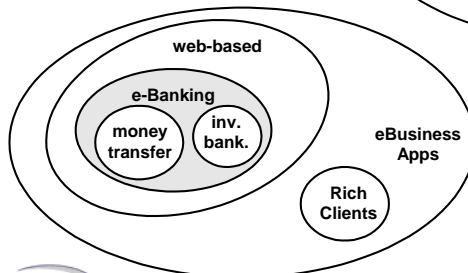
Hierarchical Structuring of Domains

- Since Domains can be of any size or granularity, it is useful to **structure domains hierarchically**.

- Automotive Example:



- eBanking Example:

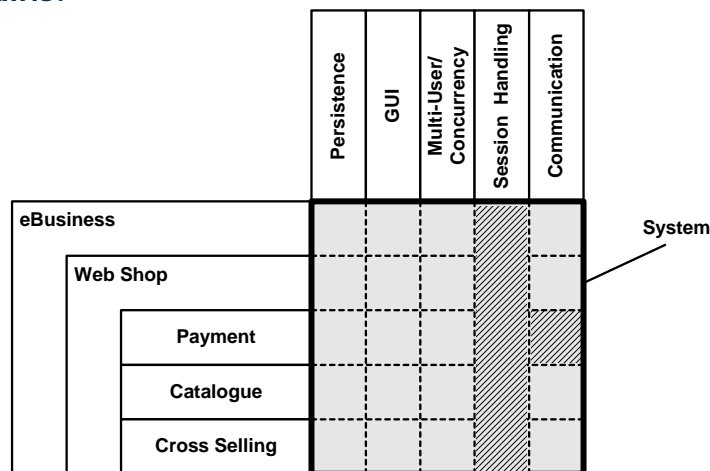


„Kinds“ of Domains

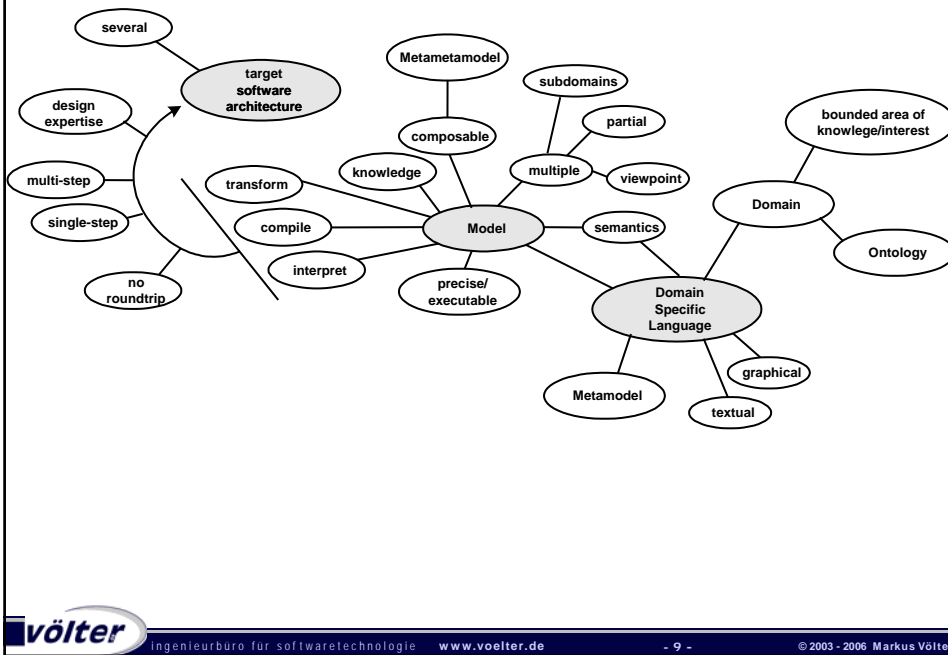
- In the context of software development it is also useful to distinguish (at least) **two kinds of domains**:
 - **Technical Domains** adress key technical issues related to software development such as
 - Distribution, Failover and Load-Balancin
 - Persistence and Transactions
 - GUI Design
 - Concurrency and Realtime
 - **Functional Domains** represent the business/professional issues; examples include
 - Banking
 - Human resource management
 - Insurance
 - Engine Controllers
 - Astronomical Telescope Control

Domains in a Software System

- As a consequence of the classifications on the previous slides, a **software system typically consists of several domains**:

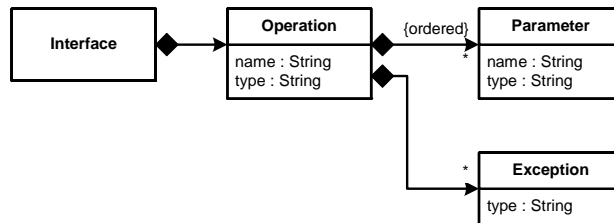


MDSO Core Concepts

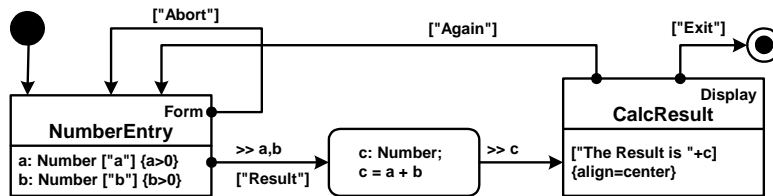


Example 1: Model and Metamodel

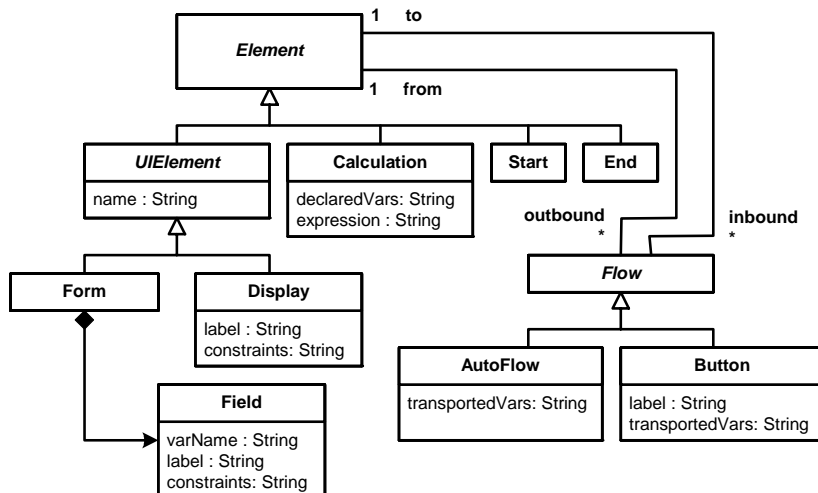
```
interface Sensor {
    operation start():void;
    operation stop():void;
    operation measure():float;
}
interface Controller {
    operation reportProblem(Sensor s,
        String errorDesc ):void;
}
```



Example 2: Model (J2ME apps)



Example 2: Metamodel (J2ME apps)



Example 3: Model (Components, Ports, Connectors)

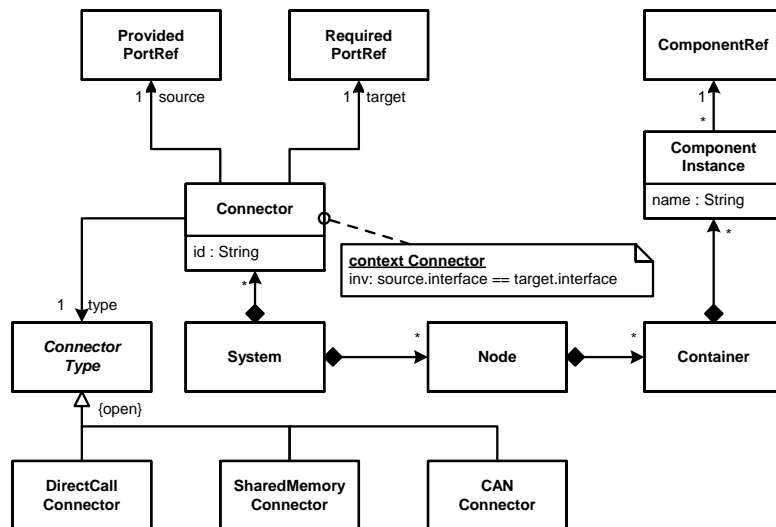
```

<system name="weatherStation">
  <node name="main">
    <container name="main">
      <instance name="controller"
        type="Control"/>
    </container>
  </node>
  <node name="in" >
    <container name="in">
      <instance name="tempOutside">
        <param name="measurementPort"
          value="tempOutside"/>
      </instance>
    </container>
  </node>
  <node name="out" >
    <container name="out">
      <instance name="controllerPort">
        <param name="controllerPort"
          value="controllerPort"/>
      </instance>
    </container>
  </node>
  <!-- temperature sensor outside -->
  <connector name="toSensorTempOutside">
    <providedPort instance="tempOutside"
      port="measurementPort">
    <requiredPort instance="controller"
      port="sensorsPort">
    </connector>
  <connector name="fromSensorTempOutside">
    <providedPort instance="controller"
      port="controllerPort">
    <requiredPort instance="tempOutside"
      port="controllerPort">
    </connector>
  <!-- humidity sensor outside -->
  <connector name="toSensorHumOutside">...</connector>
  <connector name="fromSensorHumOutside">...</connector>
  <!-- temperature sensor inside -->
  <connector name="toSensorTempInside">...</connector>
  <connector name="fromSensorTempInside">...</connector>
</system>

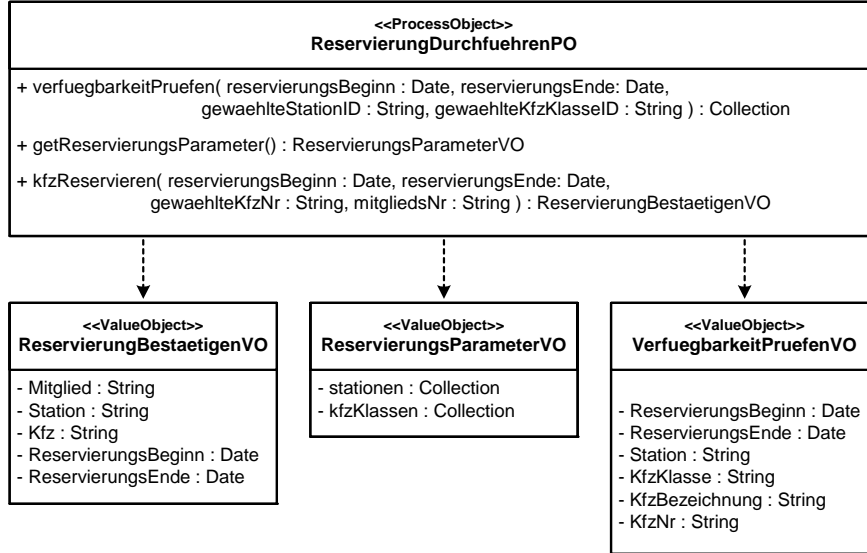
```



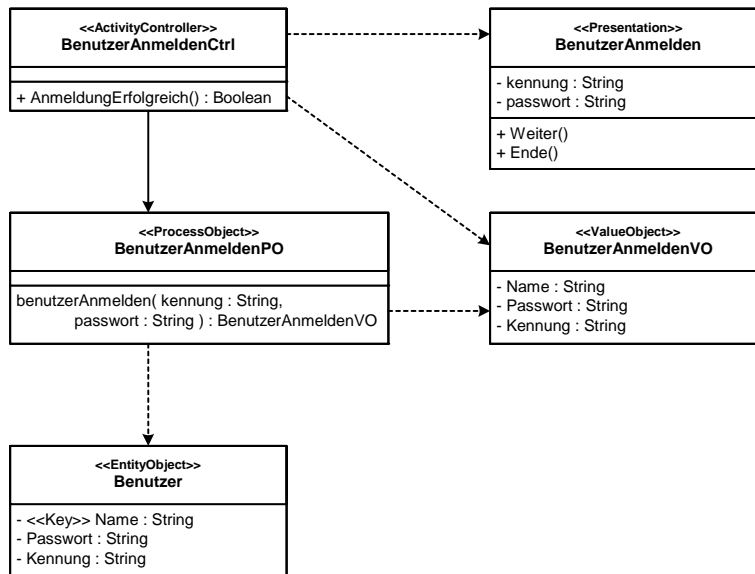
Example 3: Metamodel (Components, Ports, Connectors)



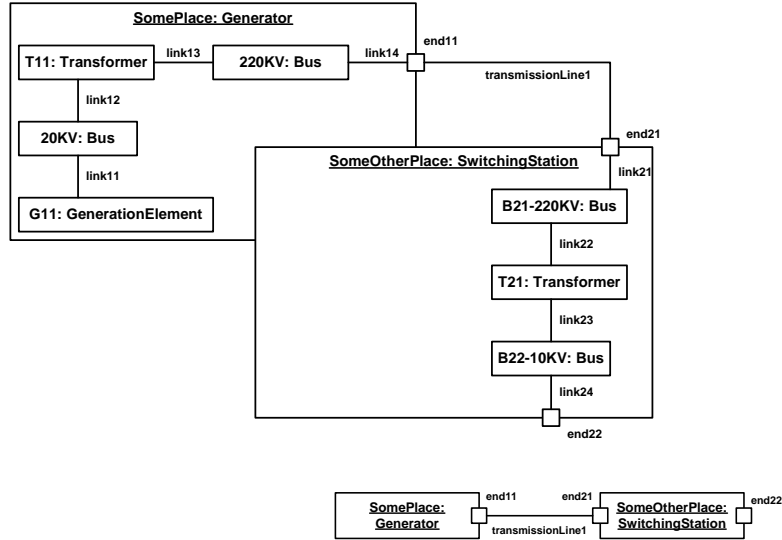
Example 4: Workflow Model (UML + Stereotypes)



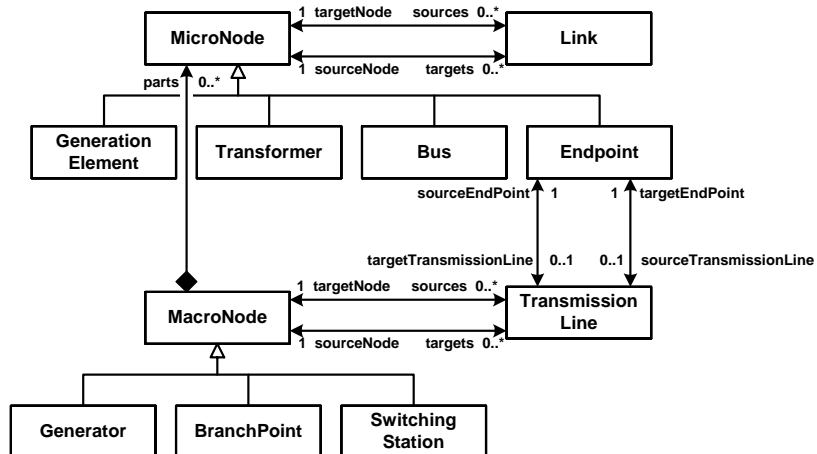
Example 4: Workflow Model II (UML + Stereotypes)



Example 5: Power Grid



Example 5: Power Grid Metamodel



MDSO Core Values

- We prefer to validate **software-under-construction** over validating software requirements
- We work with **domain-specific assets**, which can be anything from models, components, frameworks, generators, to languages and techniques.
- We strive to **automate software construction** from domain models; therefore we consciously distinguish between building software factories and building software applications
- We support the **emergence of supply chains for software development**, which implies domain-specific specialization and enables mass customization



MDSO Core Building Blocks

- domain analysis
- meta modelling
- model-driven generation
(and: model transformation)
- template languages
- domain-driven framework design
- the principles for agile software development
- the development and use of Open Source infrastructure



Other related approaches

- Microsoft's Software Factories:
Focus on Reuse, Efficient Development, DSLs
- Domain-Specific (Visual) Modelling:
Focus on (Visual) DSLs
- Generative Programming:
Focus on Efficiency, "Automatic Manufacturing", Software System Families
- Language-Oriented Programming:
Focus on DSLs instead of Frameworks, incl. Editor/Debugger Support

→ all basically the same ☺

MDSO and Agility

- **Agile Manifesto:**
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
 - Individuals and interactions over processes and tools
 - Working software over comprehensive documentation
 - Customer collaboration over contract negotiation
 - Responding to change over following a plan
 That is, while there is value in the items on the right, we value the items on the left more.
- **MDSO-Models are no „paperwork“, they *are* the code** which is translated to code automatically
- Agility does not oppose tools in general – compilers are ok, **model transformers are a kind of compiler**

MDSB and Agility II

- Project automation (ant, cruisecontrol) is ok in „agile minds“, so is automation of the writing of repetitive code
- Automation of the development process makes **responding to change** easier and faster (single source principle).
 - Changes in the model respond to changes in the functional requirements
 - Changes in the templates/transformations can be used to evolve the architecture
- The **customer on-site can be integrated better**, if we have languages that are better related to domain concepts as opposed to 3GL code or the like.
 - Pair programming between developer and domain expert is more realistic.



MDSB and Agility III

- **Tests** can still be written manually (even before generation), generators can help in building mocks or scenarios
- We do not recommend a waterfall that first builds generators and then builds apps, rather, both are iteratively evolved in parallel.
 - **Domains Architectures are based on experience**, not based on „big design upfront“
- **Developers can do what they can do best:**
 - Some deal with applications and customer requirements,
 - Others deal with technical architecture, platforms and generators
- **So: There is no problem with Agility and MDSB!**

