Software Engineering 2

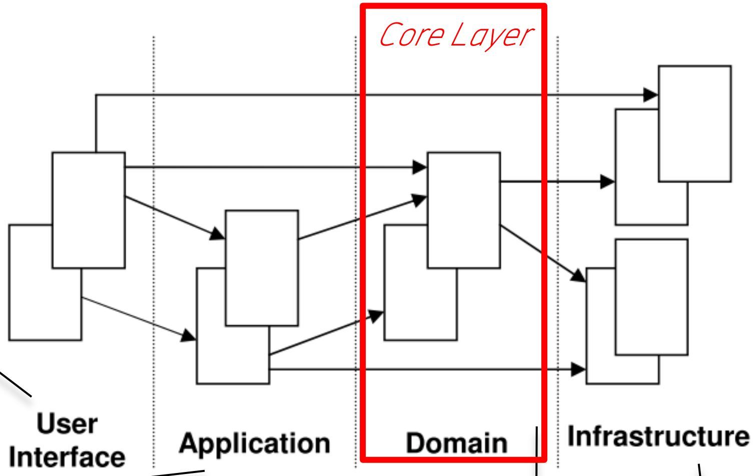
Prof. Dr.-Ing. Gerhard Wanner
Email: wanner@hft-stuttgart.de

MODEL DRIVEN SOFTWARE DEVELOPMENT- MDSD

Agenda

- Introduction
- Domain Specific Languages (DSLs)
- MDA
- Generation vs. Interpretation
- Motivation and requirements for MDSD
- MDSD and Agile Software Development.

A typical (suggested) Software Design



Presenting information to the user and interpreting user commands

Coordination of the application activity. Holds the state of an application task progress

Contains information about the domain. Holds the state of business objects. Delegates persistence of the business objects to the infrastructure layer

Supporting library for all the other layers: Communication between layers, persistence for business objects, user interface libraries, etc.

Abel Avram, Floyd Marinescu: Domain-Driven Design Quickly, http://sosa.ucsd.edu/teaching/cse294/fall2007/dddbook.pdf

How to Implement the Domain Layer – 1

- Traditionally
  - Domain Experts and Analysts …
    - specify the business domain and rules
    - define test-cases
  - Designers and Programmers …
    - design the system with the help of design methods/languages (e.g. UML)
    - implement the designed elements in a General Purpose Language (e.g. Java)
  - Test Team and Domain Experts …
    - test the implemented system.

How to Implement the Domain Layer – 2

- Agile
  - Domain Experts and Designers/Programmers …
    - specify the core business domain and rules
    - define test-cases
  - Designers/Programmers …
    - implement test cases
    - implement the specified elements in a General Purpose Language (e.g. Java)
    - continuous test and integration
  - Domain Experts and Designers/Programmers …
    - select the next set of business elements.

Problems with this approach

- No direct input of business domain / rules by domain experts
  - Transmission loss
  - Long feedback loop
- Business domain / rules tend to become hard-coded
  - Interwoven with system logic
  - Hard to change.

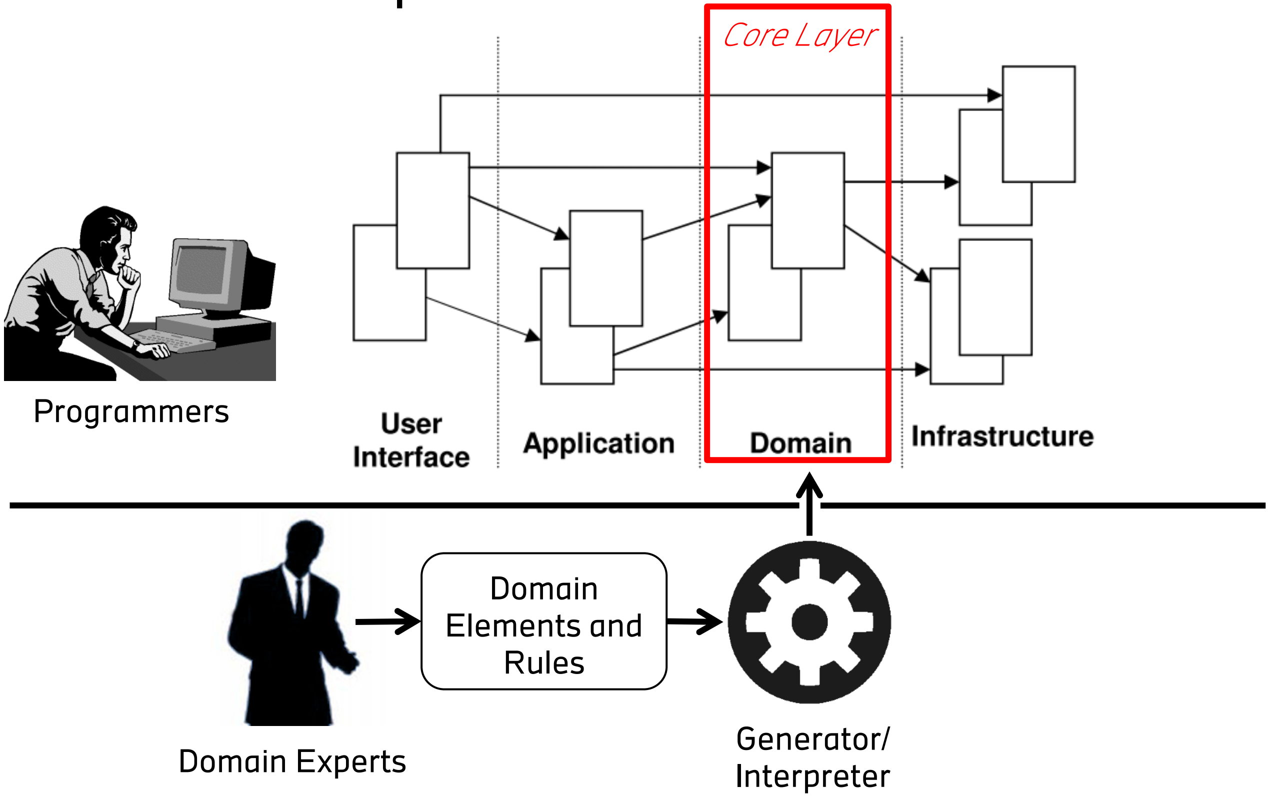
How to Implement the Domain Layer – 3

- Solution to these problems:

Model Driven Software Development

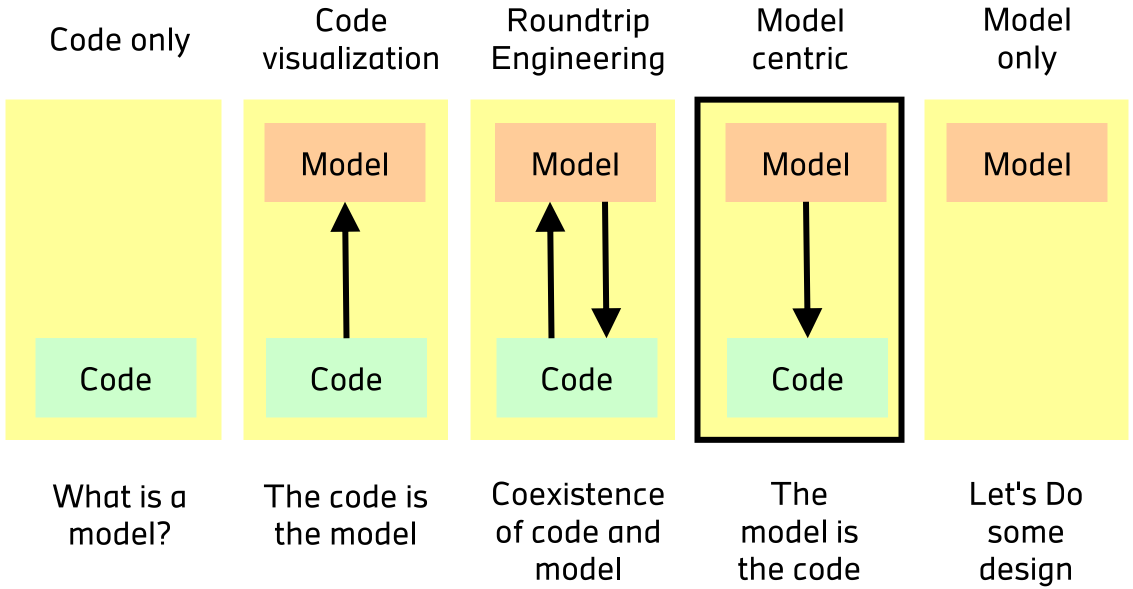
- Domain Experts (with help of Analysts / Designers) …
  - specify and define the core business domain and rules
  - specify and define the test cases
  - with the help of a Domain Specific Language (made to measure for this domain)
- Generator Tools …
  - generate code
- Designers/Programmers …
  - may add some integration code (e.g. interfaces, facades) to make it work within the whole system.

Model Driven Software Development



Synchronisation of Model and Code

- There are different approaches to synchronize model and code:



Model driven software development

Problem space
- Domain-specific terms or model
- Criteria

Configuration know-how
- Standard guidelines
- Invalid combinations
- Construction manual
- Optimization

Solution space
- Elementary components
- Combinability maximal
- Redundancy minimal

Domain Specific Language (DSL)
or
Modelling Language like UML

Generator

Components + family of system-architectures

Terms

- MDSD – Model Driven Software Development
  - Usual term for model driven software development
- MDD – Model Driven Development
  - Synonym to MDSD
- MDA – Model Driven Architecture
  - Flavor of the MDSD specified by the OMG
- MOF - Meta Object Facility
  - OMG's standard for metadata and model management
- PIM
  - Platform independent model
- PSM
  - Platform-specific model.

Agenda

- Introduction
- Domain Specific Languages (DSLs)
- MDA
- Generation vs. Interpretation
- Motivation and requirements for MDSD
- MDSD and Agile Software Development.

Domain-specific language (1)

- A domain-specific language (DSL) is a programming language or specification language dedicated to …
  - a particular problem domain,
  - a particular problem representation technique,
  - and/or a particular solution technique
- The concept has become popular due to the rise of domain-specific modeling
- The opposite is …
  - a general-purpose programming language, such as C or Java,
  - or a general-purpose modeling language such as the Unified Modeling Language (UML).

A DSL is a focused, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are natural/suitable for the stakeholders who specify that particular concern.

Domain-specific language (2)

- Examples of domain-specific languages include …
  - Logo (for children)
  - OCL (to define constraints for UML models)
  - Verilog and VHSIC (hardware description languages)
  - Mathematica and Maxima (for symbolic mathematics)
  - spreadsheet formulas and macros
  - SQL (for relational database queries)
  - YACC grammars for creating parsers, regular expressions for specifying lexers
  - Input languages of GraphViz and GrGen (software packages used for graph layout and graph rewriting)
  - Wiki-syntax (for defining Wiki-content).

```
context Vector::removeElement(d:Data)
pre: oclInState(notEmpty)
post: size = 0 implies
oclInState(empty)
```
OCL

```
SELECT books.title, count(*) AS Authors
FROM books
JOIN book_authors
ON books.isbn = book_authors.isbn
GROUP BY books.title;
```
SQL

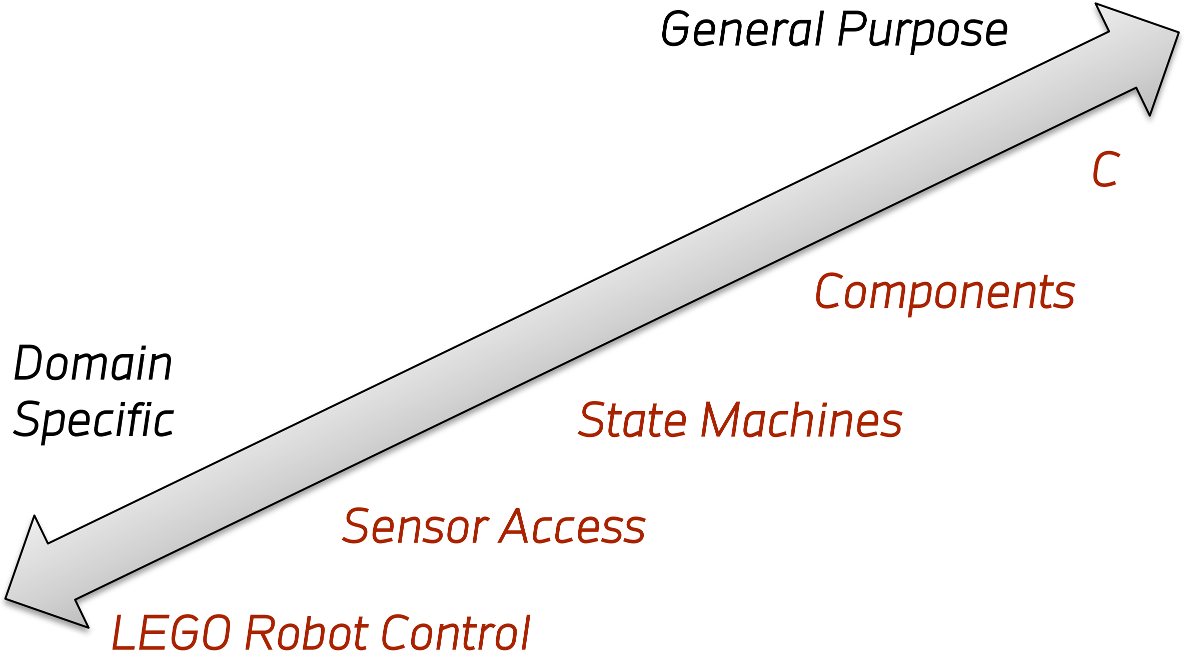
What is a DSL?

- In a common vocabulary, it's not only the nouns of the domain that get mapped to the solution space
  - You need to use the same language of the domain in describing all collaborations within the domain
  - The mini-language for the domain is modeled within the bounds of your software abstractions, and the software that you develop speaks the language of the domain
  - Example:

```
newOrder.to.buy(100.shares.of('IBM'){
          limitPrice 300
          allOrNone true
          valueAs {qty, unitPrice -> qty * unitPrice - 500}
}
```

    - This is a loud expression of the language a trader speaks on the floors of the exchange, captured succinctly as an embedded abstraction within your programming language
    - This is a DSL, a programming language targeted to a specific problem domain that models the syntax and semantics at the same level of abstraction as the domain itself.

General Purpose vs. Domain Specific Language



Comparison of GPLs and DSLs

	more in GPLs	more in DSLs
Domain size	large and complex	smaller and well-defined
Designed by	guru or committee	a few engineers and domain experts
Language size	large	small
Turing-completeness	almost always	often not
User community	large, anonymous and widespread	small, accessible and local
In-language abstraction	sophisticated	limited
Lifespan	years to decades	month to years (driven by context)
Evolution	slow, often standardized	fast-paced
Incompatible changes	almost impossible	feasible

How to develop a particular DSL?

- It's developed from the domain model and the common vocabulary business users speak
- It involved four major steps:
  - In collaboration with the business users, you derive the common vocabulary of the domain that needs to be used in all aspects of the development cycle
  - You build the domain model using the common vocabulary and the programming language abstractions of the underlying host language
  - Again, in collaboration with the business users, you develop syntactic constructs that glue together the various domain model elements, publishing the syntax for the DSL users
  - Then you develop the business rules using the syntax of the previous step. In some cases, the actual domain users may also participate in the development.

Classification of DSLs

- Martin Fowler classified DSLs based on the way they are implemented: Internal DSL vs external DSL

- Internal DSL
  - A DSL implemented on top of an underlying programming language is called an internal DSL, embedded within the language that implements it (hence, it is also known as an embedded DSL)
    - An internal DSL script is, in essence, a program written in the host language and uses the entire infrastructure of the host
    - RSpec (based on Ruby) or Gradle DSL (based on Groovy) are examples for internal DSLs
      - They use all the power of the underlying language to make you feel like using a language that was designed with a particular problem in mind
    - Internal DSLs bend and twist a host language to make it feel like a different one.

- External DSL
  - A DSL designed as an independent language without using the infrastructure of an existing host language is called an external DSL
    - It has its own syntax, semantics, and language infrastructure implemented separately by the designer (hence, it is also called a standalone DSL)
    - Cucumber, CSS, Sass are examples for external DSLs
    - External DSLs have their own syntax instead of being built on top of a language
      - All you need to make it work is a parser that interprets the language or that translates it to another one.

Advantages of using a DSL

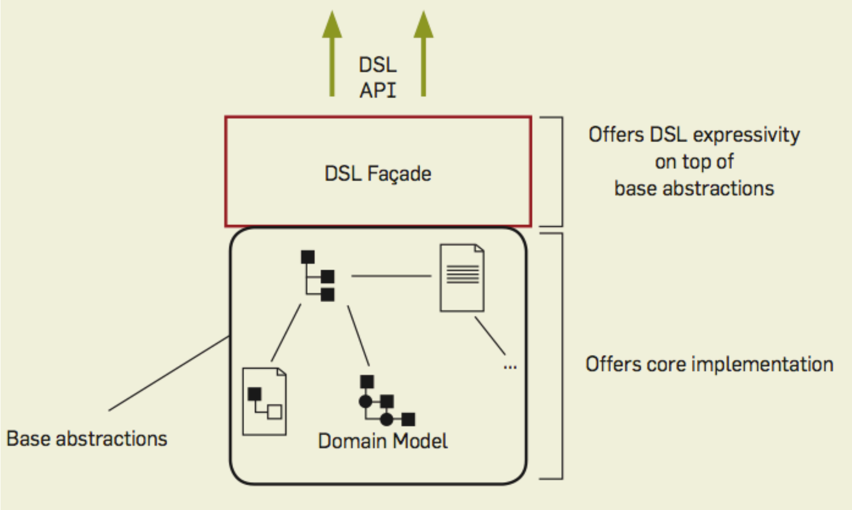
- DSL is designed to make the business rules of the domain more explicit in the programs
  - Easier collaboration with business users
    - Since a DSL shares a common vocabulary with the problem domain, the business users can collaborate with the programmers more effectively throughout the life cycle of the project
  - Better expressiveness in domain rules
    - A well-designed DSL is developed at a higher level of abstraction
  - Concise surface area of DSL-based APIs
    - A DSL contains the essence of the business rules, so a DSL user can focus on a very small surface area of the code base to model a problem domain artifact
  - DSL-based development can scale
    - With a nontrivial domain model, DSL-based development can provide higher payoffs than typical programming models
    - You need to invest some time up front to design and implement the DSL, but then it can be used productively by a mass of programmers, many of whom may not be experts in the underlying host language.

Disadvantages of using a DSL

- As with any development model, DSL-based development is not without its share of pitfalls. Your project can end up as a complete mess by using badly designed DSLs
  - A hard design problem
    - Like API design, DSL design is for experts
      - You need to understand the domain and usage pattern of target users and make the APIs expressive to the right level of abstraction
      - Not every member of your team can deliver good-quality DSL design
  - Up-front cost
    - Unless the project is at least of moderate complexity, designing DSLs may not be cost effective
    - The up-front cost incurred may offset the time saved from enhanced productivity in the later stages of the development cycle
  - A tendency to use multiple languages
    - Unless carefully controlled, this polyglot programming can lead to a language cacophony and result in bloated design.

Anatomy of a DSL

- How to design an internal DSL and embed it within an underlying host language?



Anatomy of a DSL
Source:
Debasish Ghosh,
DSL for the Uninitiated

- The base abstractions refer to the domain model designed using the idioms of the underlying host language
- The base abstractions are implemented independent of the DSL that will eventually sit on top of them
  - This makes it possible to host multiple DSLs on top of a single domain model.

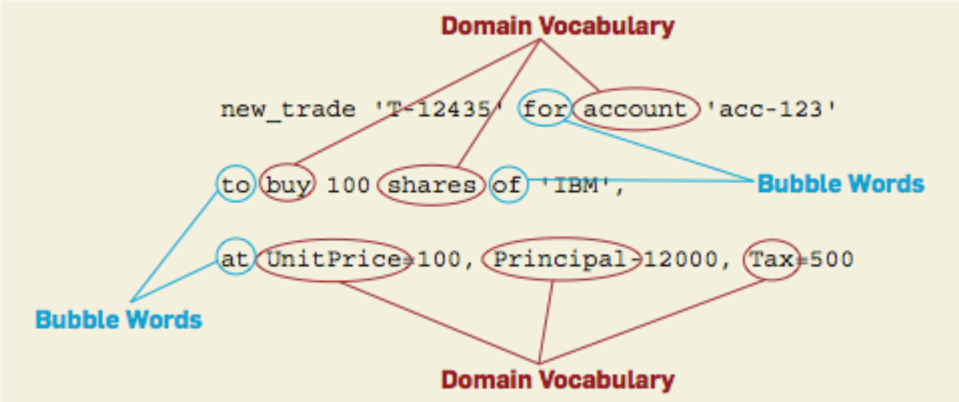
Domain vocabulary of a DSL

- Example

```
new _ trade 'T-12435' for account 'acc-123'
    to buy 100 shares of 'IBM',
    at UnitPrice=100, Principal=12000, Tax=500
```

  - This is an internal DSL embedded within Ruby as the host language and is very similar to the way a trader speaks at a trading desk

Source:
Debasish Ghosh,
DSL for the
Uninitiated



  - Annotation of that DSL, showing some of the domain vocabulary it uses and some of the “bubble words” that are introduced for the user, giving it more of an English-like.

Another DSL example

- Simpler then e.g. Java
- Probably faster than UML.

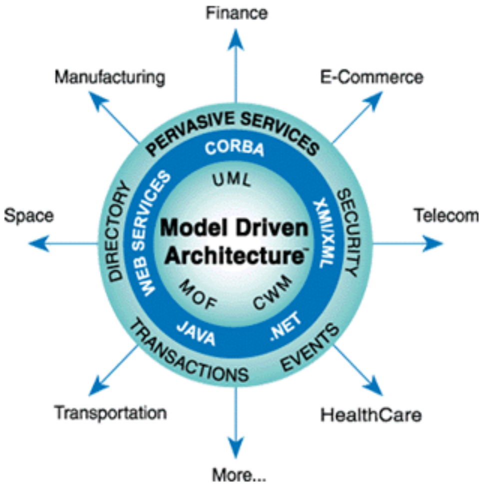
```
/*
 * DSL for management of employee-data in a company
 */
package test {
            type String
            type Date
            type float
            type int

            entity Party {
                        property name: String
                        property address: Address[]
            }
            entity Address {
                        property city: String
            }
            entity Company extends Party {
                        property legalForm: String
                        property employees: Clerk[]
            }
            entity Clerk extends Party {
                        property birthdate: Date
                        property worksFor: Company[]
            }
}
```

Agenda

- Introduction
- Domain Specific Languages (DSLs)
- MDA
- Generation vs. Interpretation
- Motivation and requirements for MDSD
- MDSD and Agile Software Development.

MDA Overview (1)

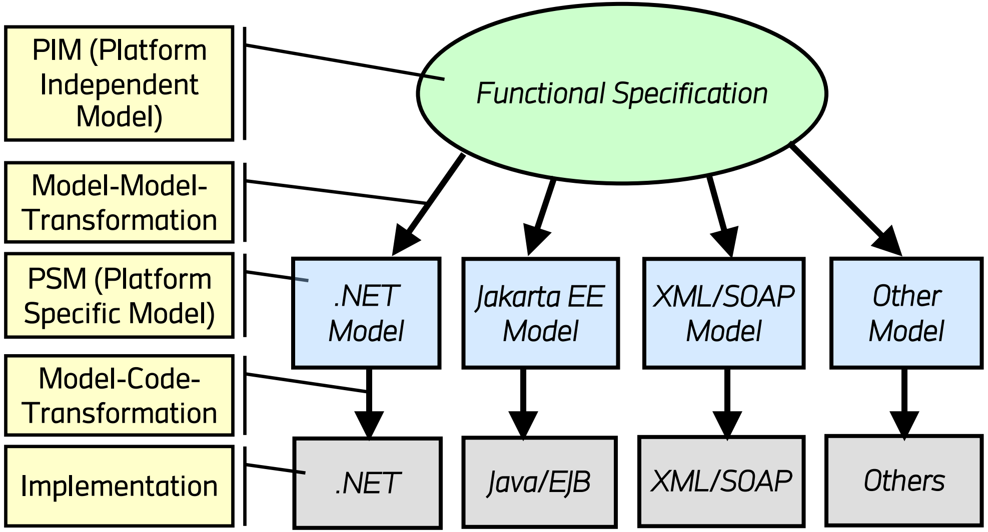
- Constantly changing techniques and new component-infrastructures complicate the development of applications
  - MDA is the answer of the OMG to that situation
- MDA consists out of 3 rings:
  - The MDA-kernel integrates the standards defined by the OMG and describes the modelling
  - The next ring of MDA contains all platforms and techniques that are necessary for the execution of the applications
  - The outer ring of MDA includes common services.



The MDA-model from the OMG that describes the interaction of the different parts [Source: www.omg.org]

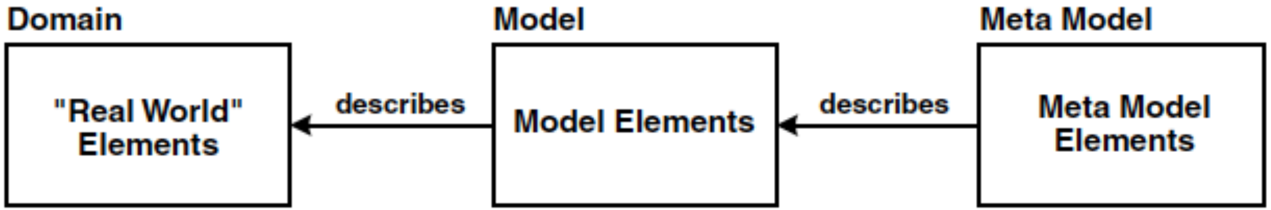
MDA Overview (2)

- The basic idea of MDA is, to generate as much code as possible for a specific platform through the usage of models
  - MDA is the consequent continuation of the idea of abstraction in the area of software-development.



MDA Metamodelling (1)

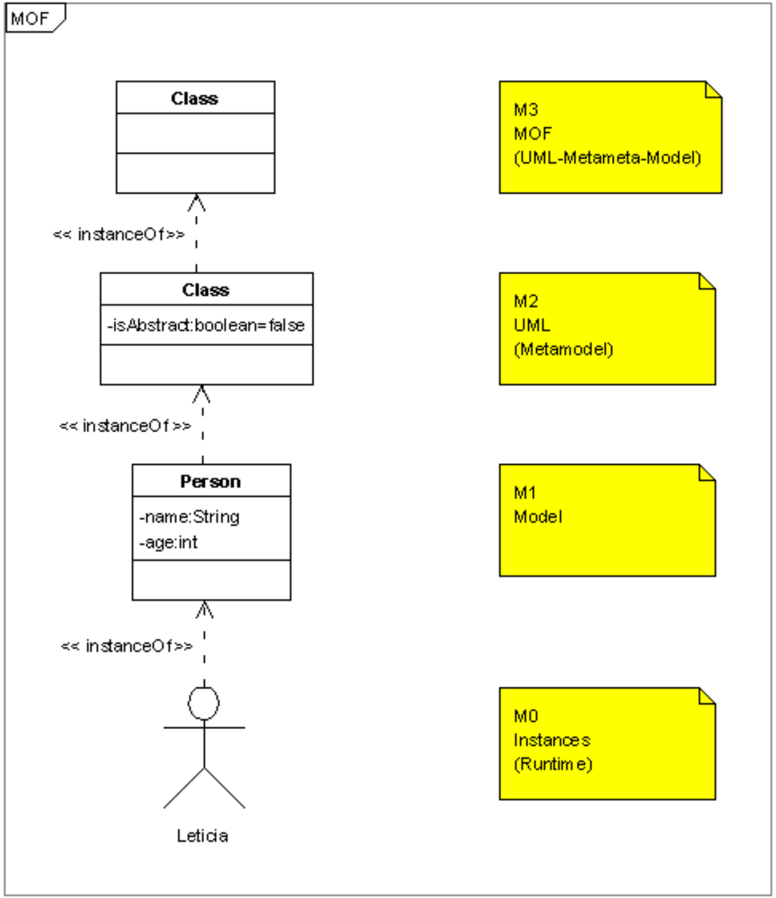
- Relationship between the real world, model and metamodel
  - Metamodels and models have a class-instance relationship: each model is an instance of a metamodel
  - To define a metamodel, a metamodeling language is therefore required that in turn is described by a meta meta model
    - In theory, this abstraction 'cascade' can be continued ad infinitum, but in practice other steps are taken.



Markus Voelter: Model-Driven Software Development

MDA Metamodelling (2)

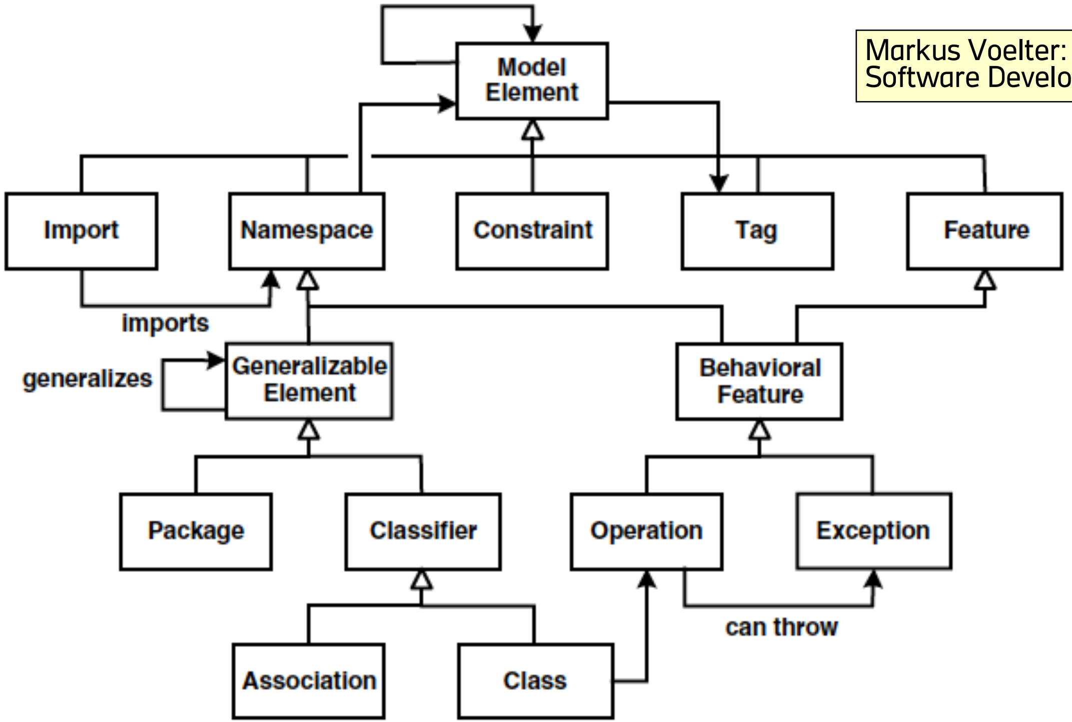
- MOF – Meta Object Facility
  - Layering of the meta-levels as described in UML 2.0 (M3 to M0)
  - The concept of the meta-levels is relative:
    - A metameta-level M3 only exists when looking from level M1
    - On the meta-level M2 it is "only" the meta-level
  - MOF is a prerequisite for model transformation
  - There is no meta-level in the OMG model above the MOF – basically, the MOF defines itself.



(Meta-)model-levels as described in MOF
[Source: www.omg.org]

MDA Metamodelling (3)

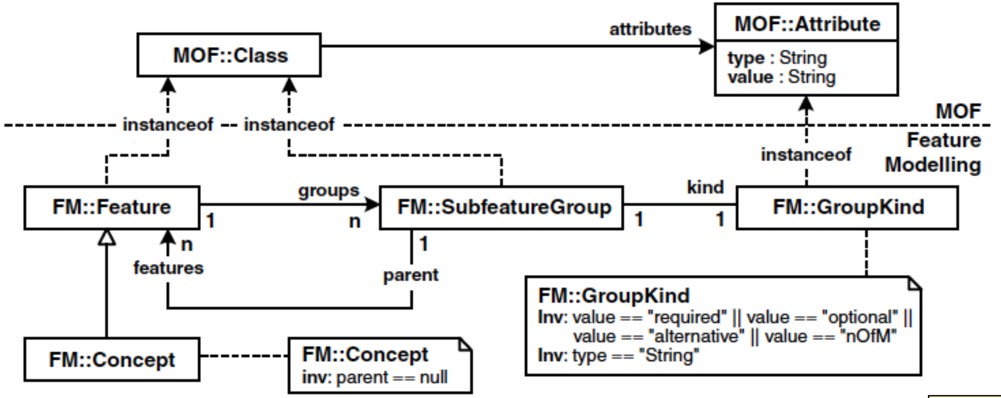
- An excerpt from the MOF



Markus Voelter: Model-Driven Software Development

MDA Metamodelling (4)

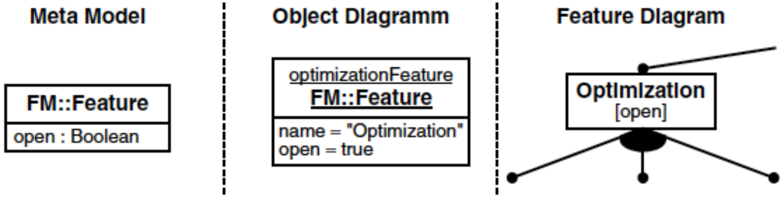
- Example of a metamodel that has nothing to do with the UML, that is, one that doesn’t extend the UML metamodel



- This example uses the feature models known from generative programming and the FODA method.

A metamodel for feature models

Metamodel and concrete syntax

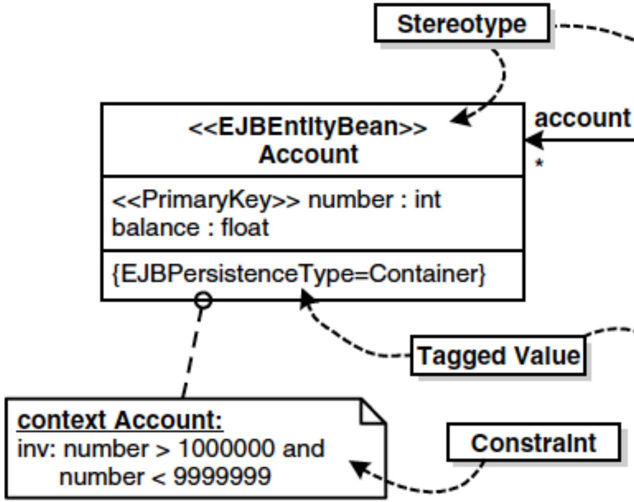


Markus Voelter: Model-Driven Software Development

MDA UML profiles (1)

- UML profiles are the standard mechanism for expanding the vocabulary of UML
  - They contain language concepts that are defined via basic UML constructs such as classes and associations, stereotypes, tagged values, and modeling rules (constraints)
  - Use of an UML profile:

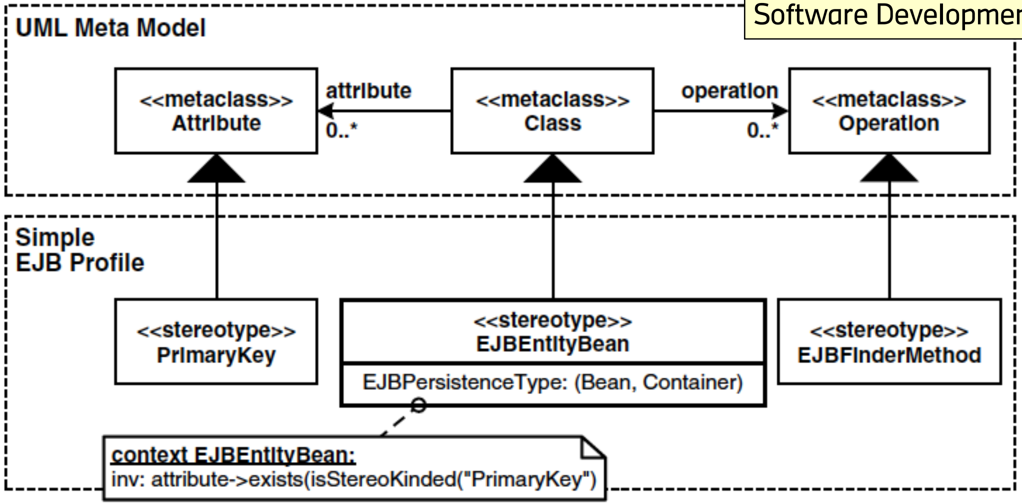
Markus Voelter: Model-Driven Software Development



MDA UML profiles (2)

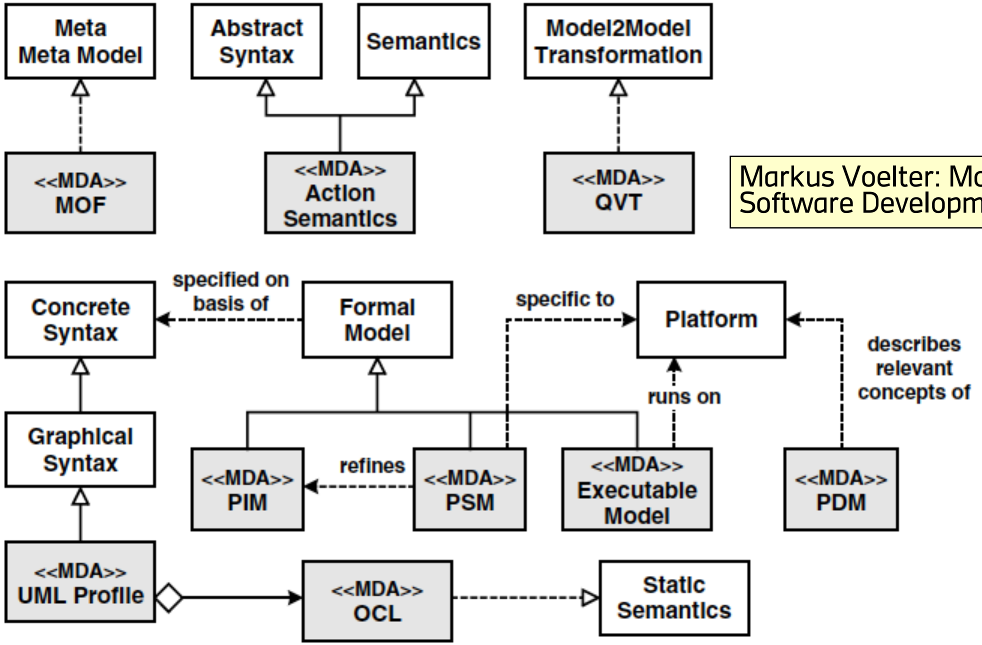
- Example: UML profile for Enterprise Java Beans (EJB)
  - In the UML profile, the standard UML concepts Attribute, Class and Operation are supplemented by the specific concepts PrimaryKeyField, EJBEntityBean and EJBFinderMethod
    - In addition, a UML 2.0 language construct, an extension, is used. This is indicated by the filled-in inheritance pointer.

Markus Voelter: Model-Driven Software Development



MDA in the context of MDSD

- Concept formation: placement of MDA concepts (grey) to MDSD concepts (white)



Markus Voelter: Model-Driven Software Development

Agenda

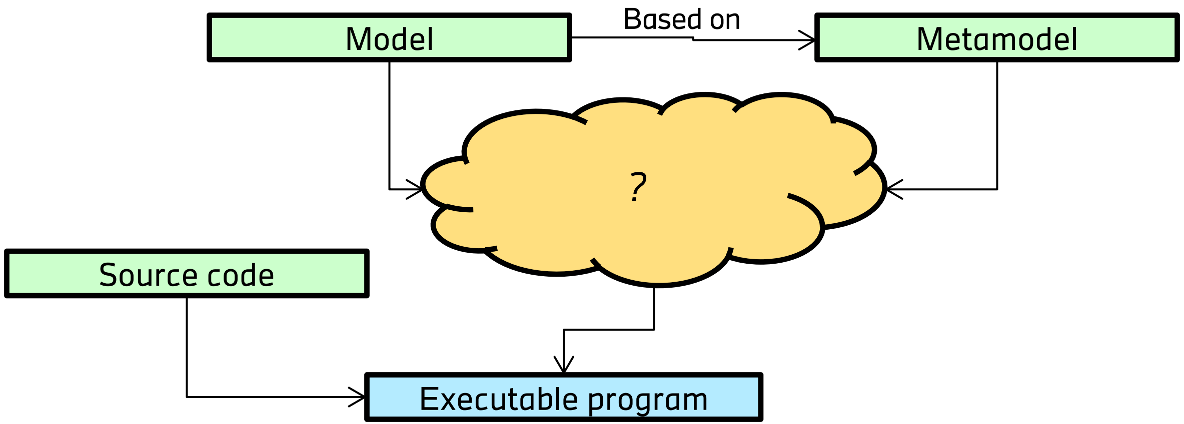
- Introduction
- Domain Specific Languages (DSLs)
- MDA
- Generation vs. Interpretation
- Motivation and requirements for MDSD
- MDSD and Agile Software Development.

Generation vs. Interpretation

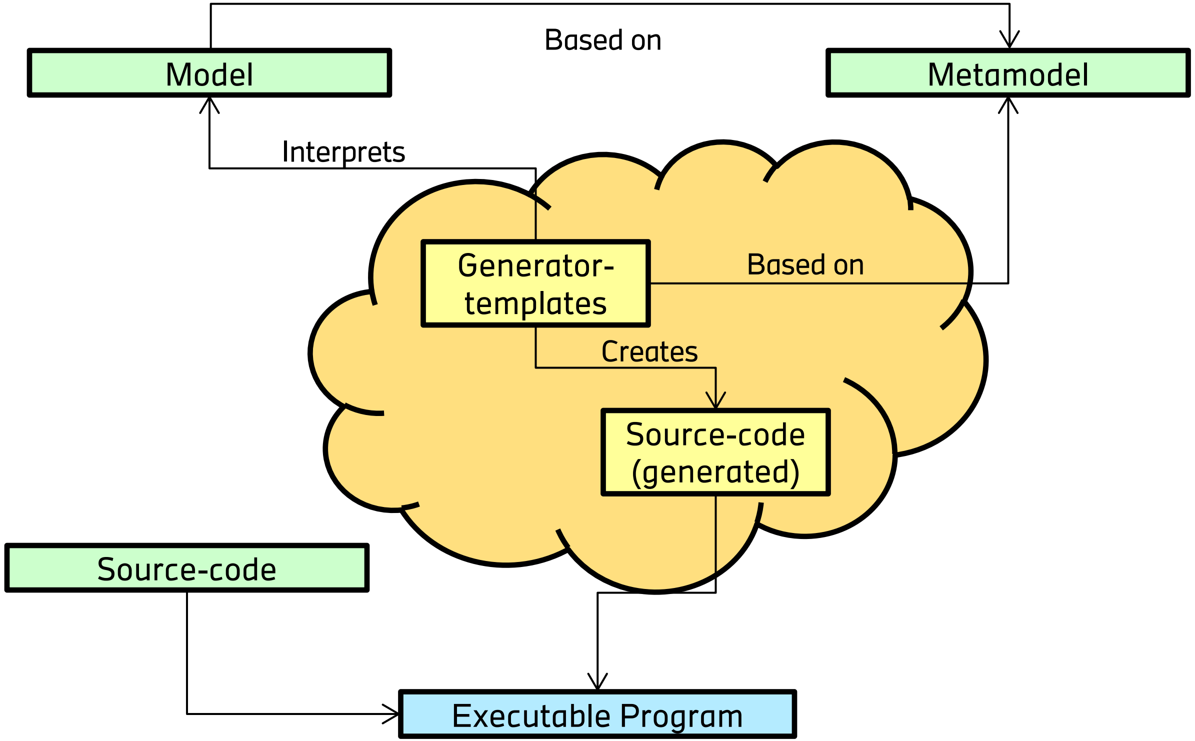
- Generated
  - *Creation* of source-code, configuration-files and other textual artifacts from models
  - *Static*
  - Before *Compilation*
  - Result: Much (generated) specific source-code
- Generic
  - *Interpretation* of (model-)information in the program
  - *Dynamic*
  - During *Execution time*
  - Result: Few (handwritten) common source-code.

Generation vs. Interpretation - Similarities

- Similarities of generative/generic approach
  - *Abstraction* of the problem-space
  - Definition of a *metamodel*
  - *Instantiation* of the metamodel
  - *Evaluation* of the metamodel
  - Same *target*: Reduction of repeated and manual programming.



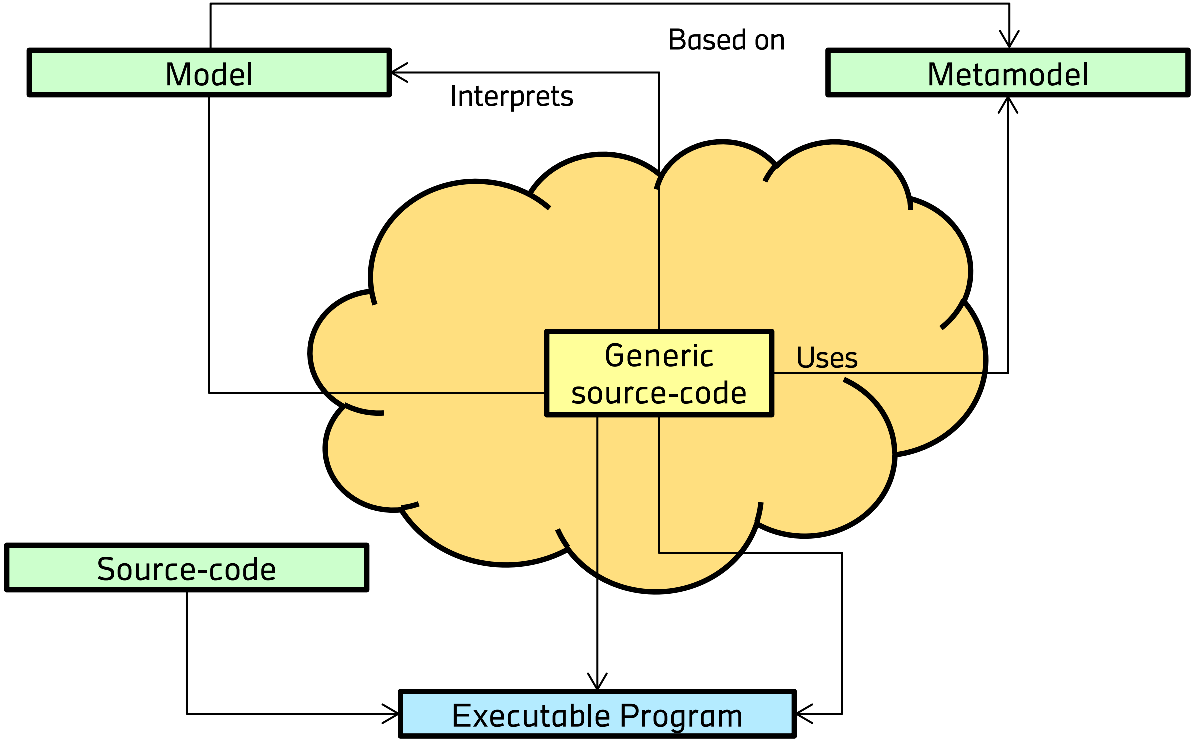
Generation - Overview



Generating - Approach

- Creation of a not-generic implementation
  - Also known as reference implementation
- Abstraction of the implementation
  - Definition of the *metamodel*
  - Creation of generator-templates to generate code out of the model
- In practice additionally:
  - Factorization of the *common parts* of all instances in super classes and libraries
  - Generation of *hooks*, to integrate hand-written code.

Interpretation - Overview



Interpretation - Approach

- Creation of a not-generic implementation
- Abstraction of the implementation
  - Definition of a *Metamodel*
  - Put common parts in parametric classes and methods → *Generic source-code*
- Creation of a base-component to load the model
- In practice additionally: Generation of ...
  - Base classes for hooks and callbacks
  - Access-classes for the model and the meta-model.

Comparison and evaluation

	Generated	*Generic*
Flexibility during runtime	--	++
Performance	+	-
Development cycles	-	+
Application start	+	-
Debugging	+	-
Development efficiency	++	++
Ability to verify	+	-
Amount of code	+/-	+/-

Agenda

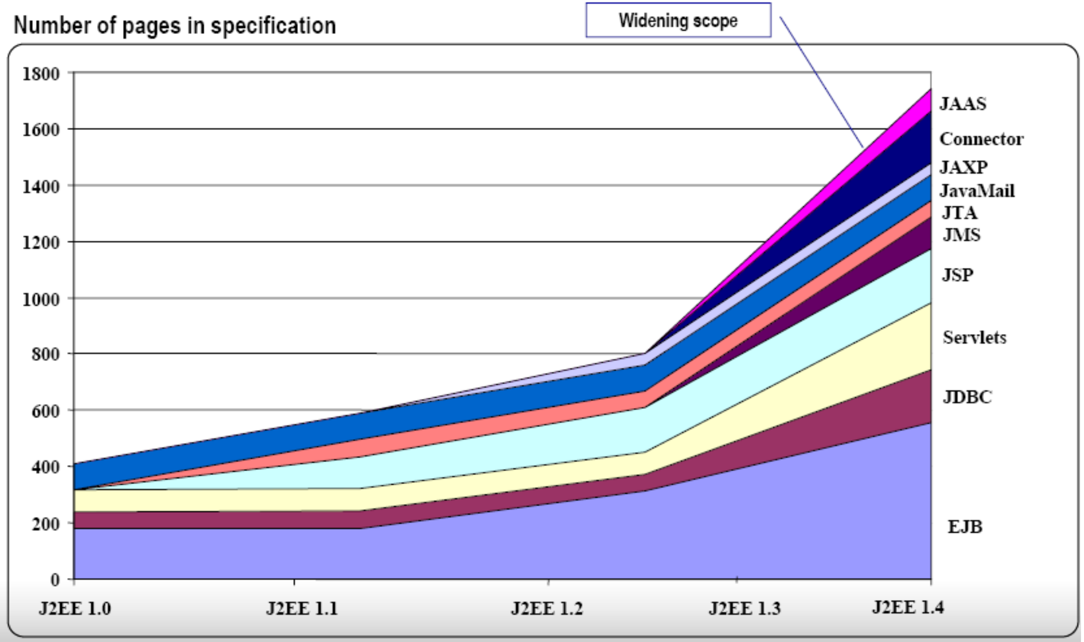
- Introduction
- Domain Specific Languages (DSLs)
- MDA
- Generation vs. Interpretation
- Motivation and requirements for MDSD
- MDSD and Agile Software Development.

Why use MDSD? (1/2)

- Models
  - Better reuse of models through their long-life-cycle
  - Actual documentation of the system
    - Discussions on different levels possible (models for specific roles)
- Generation
  - Modelling of central attributes of the architecture; no fixed generator
  - Constant code-quality for the generated parts (using good templates)
- Application area
  - Primary for development of new applications
  - Retroactive usage for an application developed without MDSD difficult
- Development process
  - High productivity, if architecture and tools is established
  - Good integration when using an iterative process model (if the tools supports it…).

Why use MDSD? (2/2)

- The knowledge about current standards for a single developer is nearly impossible.



Source:
Agile Model Driven Architecture
Alfred Bröckers, Volker Gruhn

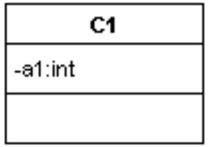
Requirements for the usage of MDSD (1/3)

- Model
  - At least „marked PIM“ (a.k.a. annotated PIM)
  - DSL
- Metamodel
  - Coherent and adjustable; no proprietary XMI-Format
  - Ideal: UML, MOF
- Template-language
  - High functionality and clarity; e.g. no JavaScript
  - Flexible template-system
    - Templates freely adjustable from the developer
  - Direct usage of the model-elements in the template-language
  - Ideal: Velocity, widely used
- When using UML: import-possibilities
  - Good import-possibilities from the modelling-tool
  - Ideal: XMI, but often incompatible because of different versions.

Insert: XMI (1)

- XMI – XML Metadata Interchange
  - Standardized language (OMG standard) to exchange models between development-tools
  - Many tools often only process XMI-information from class-diagrams
  - Also more and more supported:
    - Use Case-
    - Activity-
    - State-Chart-

    Diagrams.

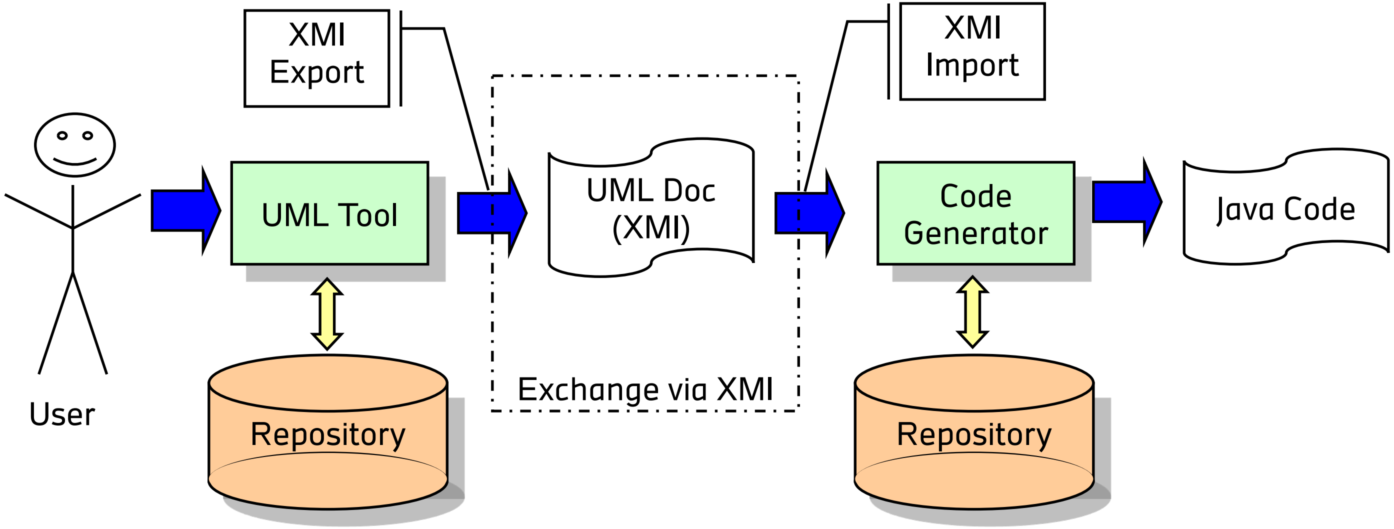



```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML'
xmlns:UML2 = 'org.omg.xmi.namespace.UML2'
  timestamp = 'Wed Aug 09 15:06:41 CEST 2006'>
  <XMI.header...</XMI.header>
  <XMI.content>
    <UML:Model name = 'Modell 1' isAbstract = 'false'>
      <UML:Class name = 'C1' visibility = 'public'
        isAbstract = 'false'>
        <UML:Classifier.feature>
          <UML:Attribute name = 'a1' visibility = 'private'
            ownerScope = 'instance' changeability = 'changeable'>
            <UML2:TypedElement.type>...</UML2:TypedElement.type>
          </UML:Attribute>
        </UML:Classifier.feature>
      </UML:Class>
  </UML:Model>
  ...
```

Example for a XMI-definition of a class C1 (heavily reformatted and shortened)

Insert: XMI (2)

- XMI – XML Metadata Interchange



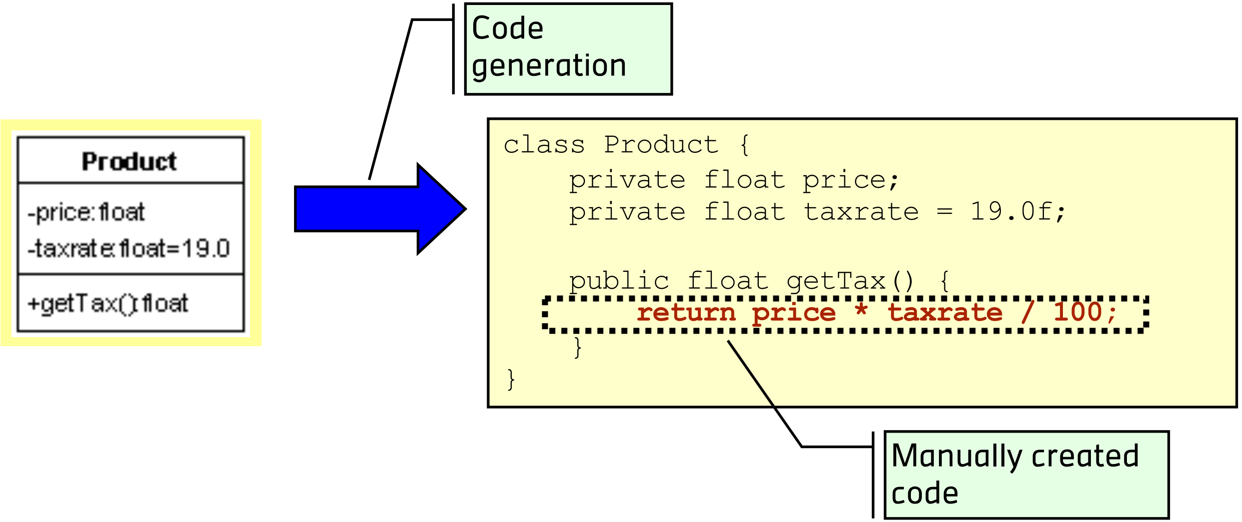
Example for the exchange of model information using XMI

Requirements for the usage of MDSD (2/3)

- Integration into development environment
  - Support for the creation of the complex configuration options
  - Ideal: Integration into the IDE
- Development process
  - Forward-Engineering with protected regions
  - Ideal: Forward-Engineering with subclassing
- Development cycle time
  - Adequate cycle time: model-transformations must be not to slow
  - Minimum: Splitting of models, so only changed parts have to be regenerated
    - Needs a good build-tool
  - Ideal: Tool recognizes dependencies and automatically creates the necessary parts.

Insert: How to manually insert code into generated files? (1)

- Problem: MDSD (all generative approaches) generate code
  - Usually this code is not sufficient, it has to be extended by the developer manually:



Insert: How to manually insert code into generated files? (2)

- Solutions
  - Subclassing
    - Superclass is generated
    - Subclass is written by hand or generated only once with an empty implementation
    - Problems:
      - Programming more complex (e.g. factory needed to create instance)
      - More restrictive (e.g. interface fix)
  - Protected regions
    - Are never overwritten by generator
    - Problems:
      - Name-change of artefacts
      - User deletes marks
      - Generated code has to be inserted into version control.

Generated

```
class Product {
    private float price;
    private float taxrate = 19.0f;
    public abstract float getTax();
}
```

Handwritten

```
class ProductImpl extends Product {
    public float getTax() {
        return price * taxrate / 100;
    }
}
```

Protected region

```
class Product {
    private float price;
    private float taxrate = 19.0f;
    /* @@@@ start user code @@@@ */
    public float getTax() {
        return price * taxrate / 100;
    }
    ...
    /* @@@@ end user code @@@@ */
}
```

Requirements for the usage of MDSD (3/3)

- Orientation/Handling
  - Quick usage of the tool for own development
  - Minimum: Clear and modular design of the tool
  - Ideal: Quick start-document, intuitive example-application / example-scenario
- Support/Literature/Community
  - Support for a productive usage under project conditions
  - Ideal: Commercial Support available, enough and actual documentation, active community.

Agenda

- Introduction
- Domain Specific Languages (DSLs)
- MDA
- Generation vs. Interpretation
- Motivation and requirements for MDSD
- MDSD and Agile Software Development.

MDSD and Agile Software Development (1)

- An iterative-incremental process is a strong ally for MDSD, and strict timeboxing helps to implement the feedback loop between architecture development and application development smoothly
- One of the highest priorities in agile software development is the development of runnable software that can be validated by both stakeholders and end users – as it is also in MDSD
- MDSD encompasses several techniques and methods that enable the use of principles of agile software development in complex projects
  - These techniques support agile requirements management and the regular validation of software under construction.

MDSD and Agile Software Development (2)

- Individuals and Interaction over Processes and Tools
  - After all this means that no over-formal processes that don’t heed people should be established. A team should define its own development process, suited to its specific conditions, and continue to evolve it over time
  - The use of tools such as versioning systems or compilers is obviously not being criticized here. Under the premise that a part of the programming in MDSD is done via DSL, the generator replaces the compiler and there is no contradiction with agile development
- Working Software over Comprehensive Documentation
  - In project practice it is more important to deliver runnable software instead of good-looking documents such as requirements, concepts, architecture, design. In MDSD, the model is the source code. Diagrams are not just adornments, but a central artifact. The diagrams and the software will not drift apart and are always up-to-date, because the application is directly generated from the model.

MDSD and Agile Software Development (3)

- Customer Collaboration over Contract Negotiation
  - This aspect expresses the wish to allow the customer to participate as much as possible in application development. Particularly, a fast response to changing customer requirements should be possible
  - Here MDSD can have a considerable advantage over traditional iterative, incremental development. This is especially true if a non-technical DSL is applied that can be (re-)used to communicate with the customer, thereby shortening feedback cycles: a DSL is otherwise independent of whether MDSD is applied or not
- Responding to Change over Following a Plan
  - This valuation is about incorporating the (changing) requirements of the customer flexibly during a project, instead of insisting on formally-defined requirements. MDSD makes this procedure much easier:
    - When domain-related requirements change the generative approach allows these changes to be implemented much faster than in traditional software development.
    - Technical aspects implemented by the transformations can be adapted in one place, and the change is automatically propagated in the entire application.

Literature

- Debasish Ghosh
  - DSL for the Uninitiated - Domain-specific languages bridge the semantic gap in programming
  - Communications of the ACM, 7/2011
- Martin Fowler
  - Domain Specific Languages
  - AddisonWesley, 2010
- Georg Pietrek, Jens Trompeter (Hrsg.)
  - Modellgetriebene Softwareentwicklung – MDA und MDSD in der Praxis
  - entwickler.press, 2007, ISBN: 978-3-939084-11-2
- Lorenzo Bettini
  - Implementing Domain-Specific Languages with Xtext and Xtend
  - Packt Publishing, 2013, ISBN-13: 978-1782160304