L4. DDD and MDE

S1. Domain-Driven Design

Dr A Boronat

Part II: Case Study

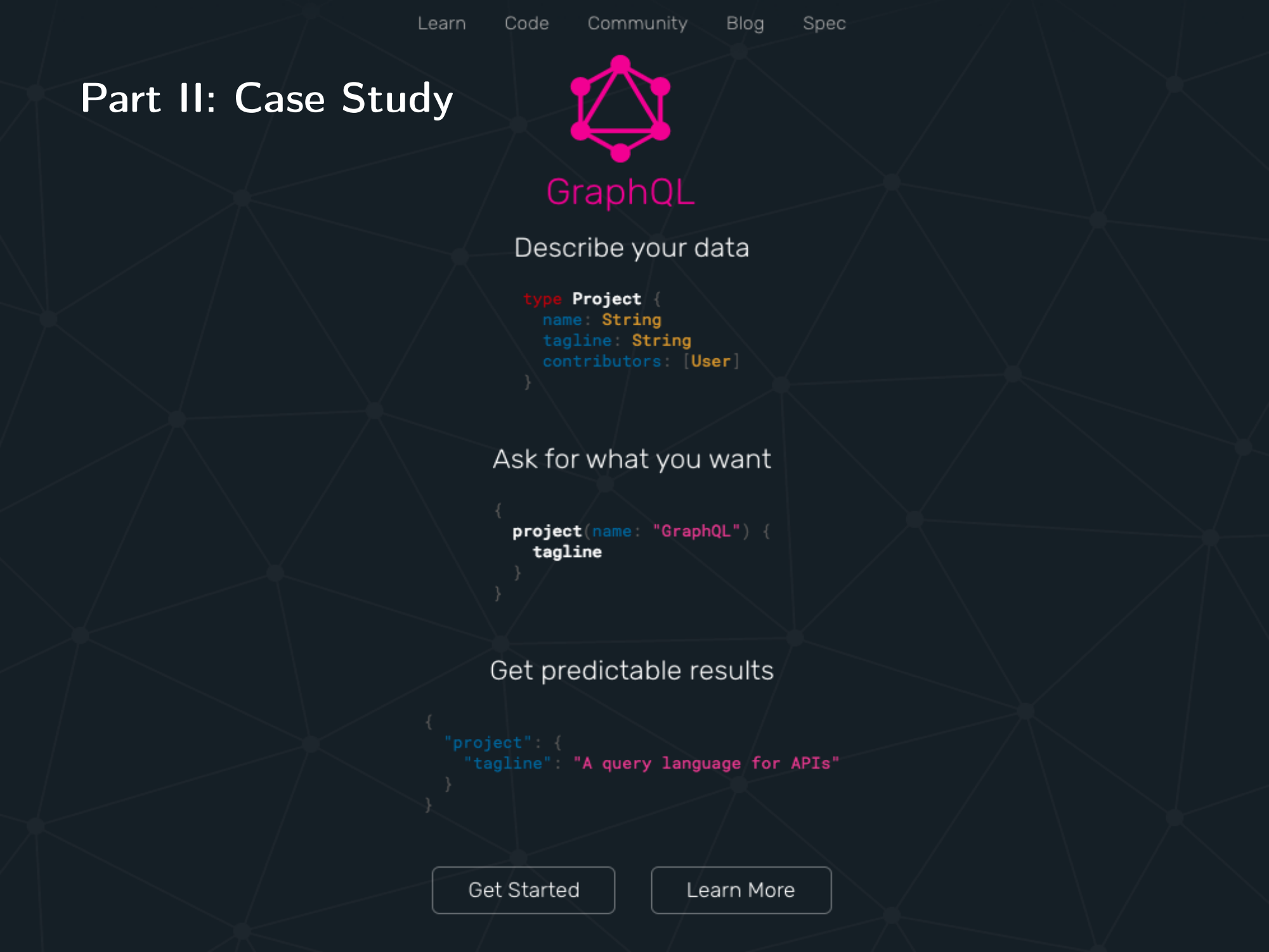
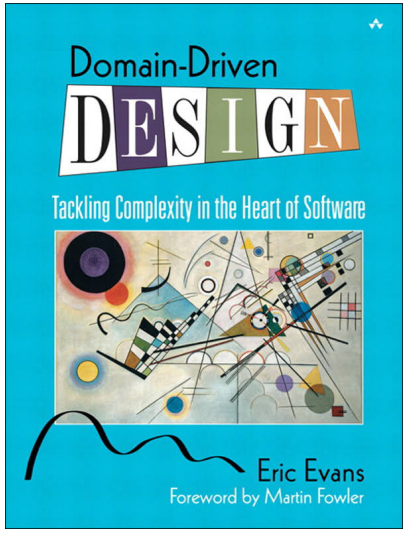


Table of Contents

Domain-Driven Design

Approach to software development where implementation is influenced by an evolving model



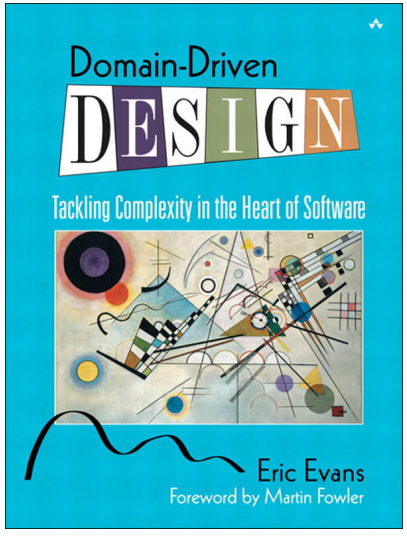
Developing software amounts to mapping a problem domain into a solution domain

- domain modelling: to analyse, understand, and identify the participants involved in a domain
- problem domain: processes, entities and constraints that are part of the business that you are analysing (e.g. an online shop)
- solution domain: tools, frameworks, libraries, methodologies, and techniques

Domain

- A domain is an area of knowledge [2]
  - scoped to maximize the satisfaction of the requirements of its stakeholders,
  - including a set of concepts and terminology understood by practitioners in that area, and
  - including knowledge of how to build software systems (or parts of software systems) in that area.
- Domain model: conceptual model of the problem domain, covering relevant entities, their attributes, roles, and relationships, plus the constraints and interactions that describe and grant the integrity of the model elements comprising that problem domain
  Ex: banking (customer, personal loan, mortgage, LTV ratio, booking fee, interest rate, capital repayment,...)
- Solution domain: specific working contexts for the specification, implementation, and deployment of applications
  Ex: software architectures (components, interfaces, messages, dependencies, processes, shared resources,...)
  Ex: software development processes (task, activity, dependency, milestone, stakeholder, resource, skill,...)

Domain-Driven Design



Ubiquitious Language

A domain-specific language structured around the domain model

- evolves as a collaboration between the domain experts and the software experts
- used by all team members to connect all the activities of the team with the software

Domain-Specific Languages (DSLs)

- A domain-specific language is a computer programming language of limited expressiveness focused on a particular domain [3]:
  - computer programming language: a DSL is used to humans to instruct a computer to do something, as well as helping communication between humans;
  - language nature: a DSL is a programming language, and as such should have a sense of fluency where the expressiveness comes not just from individual expressions but also from the way they can be composed together;
  - limited expressiveness: a DSL supports a bare minimum of features needed to support its domain. You cannot build an entire software system in a DSL, rather you use a DSL for one particular aspect of a system;
  - domain focus: a limited language is only useful if it has a clear focus on a limited domain.

Examples of DSLs

- Unix shell scripts:
  - DSL for data organization
  - domain abstractions include streams (stdin and stdout) and operations on streams (redirection, pipe)
- Regular expressions:
  - DSL for detecting patterns in strings
  - concise syntax to specify patterns
- wiki languages: markdown
- markup languages: HTML
- build automation: Make, Gradle
- persistence stores: MongoDb API in Groovy
- requirement specification: Gherkin
- more examples

DSL classification [3]

- External DSL:
  - programming language with tool support of its own
  - Ex: Awk, SQL, XML configuration files for systems like Struts or Hibernate
- Internal DSL:
  - embedded in a general-purpose language and, thereby, reuses features of the host language
  - Ex: Gradle in Groovy
- Language workbench:
  - specialized IDE for defining and building DSLs
  - Ex: Eclipse IDE for Java and DSL Developers, MetaEdit

DSL classification [1]

- Focus:
  - Vertical DSLs for a specific industry or area. Ex: configuration languages for home automation systems, modelling languages for biological experiments, analysis languages for financial applications, etc.
  - Horizontal DSLs have a broader applicability. Ex: SQL, WebML, etc.
- Style:
  - Declarative DSLs: follow a specification paradigm that allows to capture the application logic without being too deterministic
  - Imperative DSLs: define an executable algorithm that states the steps and control flow that needs to be followed to successfully complete a job
- Notation:
  - Graphical DSLs provide graphical primitives such as blocks, arrows and edges, containers, symbols, etc.
  - Textual DSLs provide a textual surface language
- Execution:
  - Model interpretation: execution of the DSL script at run time, one statement at a time
  - Model translation: application of model-to-text transformations at deployment time

Advantages and disadvantages [5]

advantages

- expressiveness: abstractions with precise semantics in the application domain
- conciseness: well-defined scope
- high level of abstraction: low-level language constructs, data structure optimization and implementation techniques are hidden from the DSL programmer
- high payoff in the long run
- DSL-based development is scalable

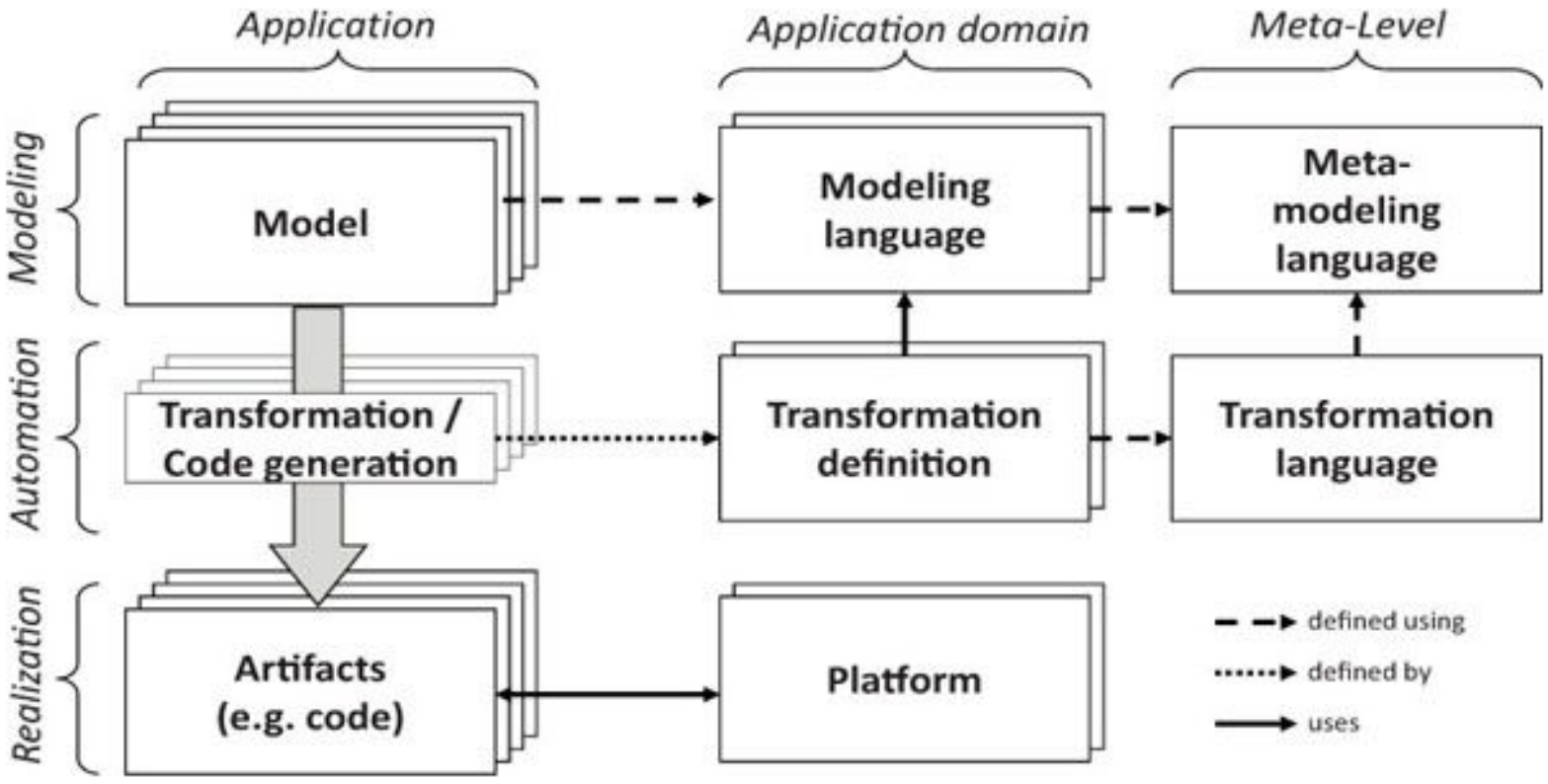
disadvantages

- language design can be hard
- upfront cost
- non-technical domain experts may not be able to write or maintain DSL programs by themselves [4]
- using DSLs can lead to performance concerns: one more layer of indirection
- sometimes they lack adequate tool support
- yet-another-language-to-learn syndrome: a new language with limited capabilities

Agile Automation using Model-Driven Engineering

Software development paradigm that uses models as first-class citizens in the development process

Components



Software model: standard definitions

A software model is

- A related collection of instances of meta-objects, representing (describing or prescribing) an information system, or parts thereof, such as a software product.

[ISO/IEC 15474-1:2002]

- A semantically-closed abstraction[1] of a system or a complete description of a system from a particular perspective.

[ISO/IEC/IEEE 24765:2010]

[1] An abstraction focuses on the essential attributes of the subject, removing any unnecessary details from the user [5, ch 1].

Software Modelling and DSLs

- Informally, a model can be viewed as a simplified or partial representation of a reality, that is as an abstraction, which can be used to:
  - generalize specific features of real objects (generalization);
  - classify the objects into coherent clusters (classification);
  - aggregate objects into more complex ones (aggregation).
- Main roles of models as abstractions:
  - reduction: only a relevant selection of properties of a reality are considered
  - mapping: a model is based on a particular prototypical individual, abstracting and generalizing it
- Purpose of models:
  - descriptive: describe the reality of a system or a context
  - prescriptive: define how a system shall be implemented

Software Modelling vs Drawing

- Software modelling is not about defining diagrams, although they may be involved.
- In modelling, diagrams and drawings (or textual descriptions) have implicit but unequivocally defined semantics which allow for precise information exchange and many additional usages.
- Modelling also facilitiates:
  - syntactical validation
  - model checking
  - model simulation
  - model transformations
  - model execution (either through code generation or model interpretation)
  - model debugging

Modelling languages

- Language used to specify system models that consists of
  - a well-defined notation, either textual or graphical
  - a precise semantics
- Two big classes of modelling languages[6]:
  - Generic-purpose modelling languages: UML, state machines, Petri nets, etc.
  - Domain-specific modelling languages: HTML, SQL, etc.

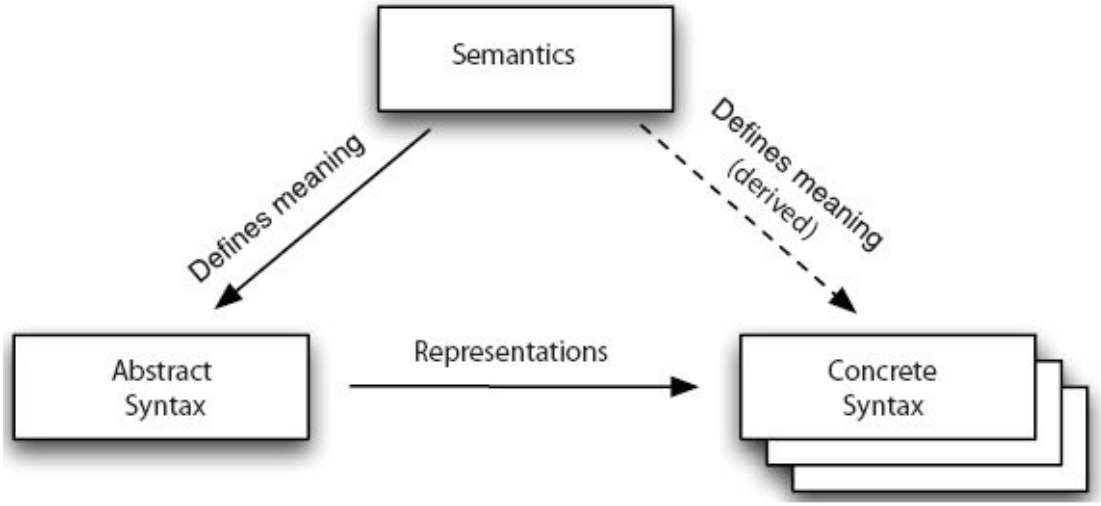
	GPLs	DSLs
domain	large and complex	smaller and well-defined
Turing completeness	always	most often not
user-defined abstractions	sophisticated	limited
lifespan	years to decades	driven by context
designed by	guru or committee	a few engineers and domain experts
user community	large, anonymous and widespread	small, accessible and local
evolution	slow, often standardized	fast-paced and agile

Modelling Languages: approaches to DSLs

- Extending UML (Unified Modeling Language)
  - internal DSMLs: vertical domains (e.g. finance, e-commerce) or horizontal domains (e.g. Web applications, service-oriented architectures)
  - UML extensions through UML profiles
  - tool support available through UML-compliant technology
- MOF-based DSLs
  - external DSL tailored to a particular domain without the additional machinery of a host language
  - standalone definition of a DSL (to be discussed in the next session of this lecture)
  - tool support available through MOF-compliant technology

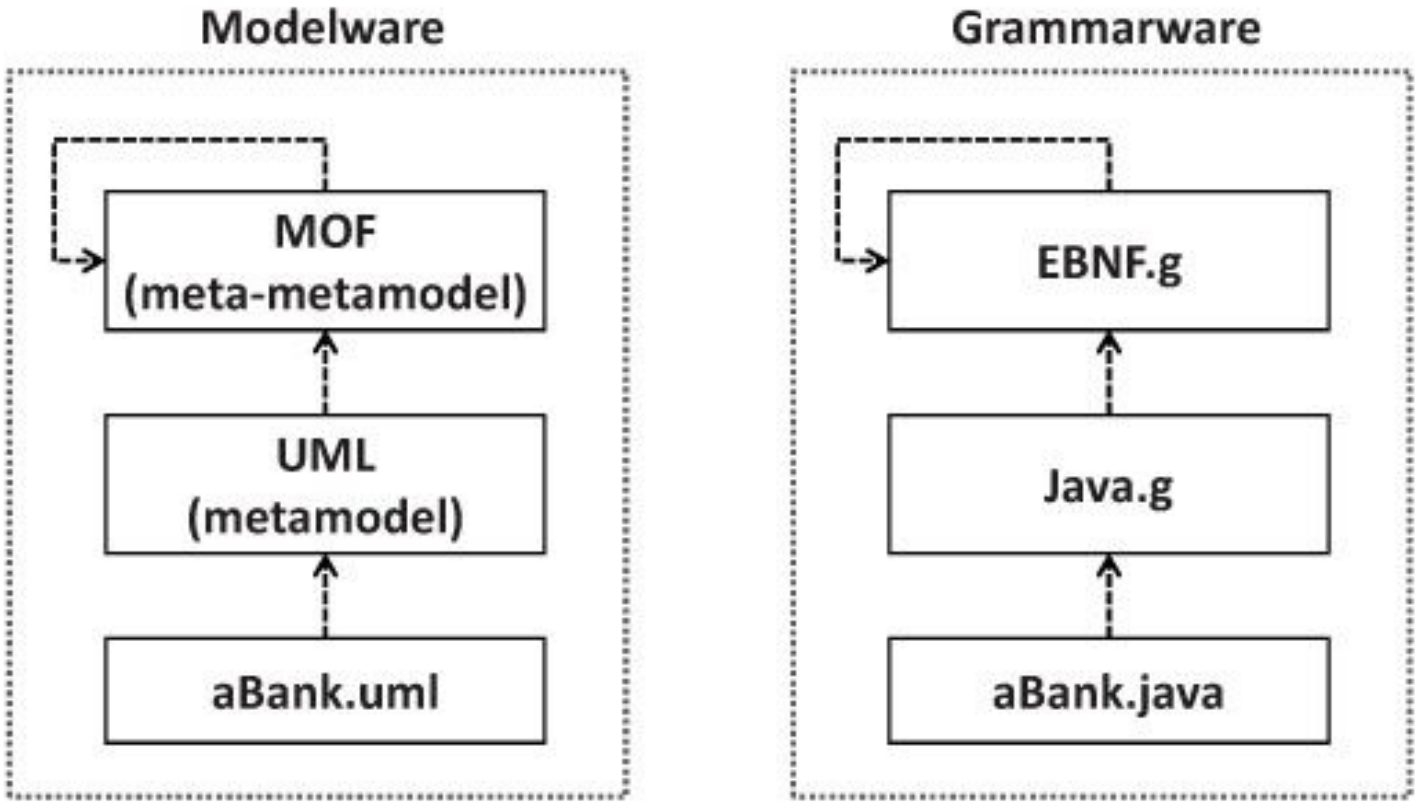
Components of a modelling language [1]

- Abstract syntax:
  - structure of the language and mechanisms to combine the primitives of the language, independently of any representation or encoding
- Concrete syntax:
  - specific representations of the modelling language, covering encoding and/or visual appearance issues.
  - textual or graphical
- Semantics:
  - meaning of the elements defined in the language
  - meaning of the different ways of composing them



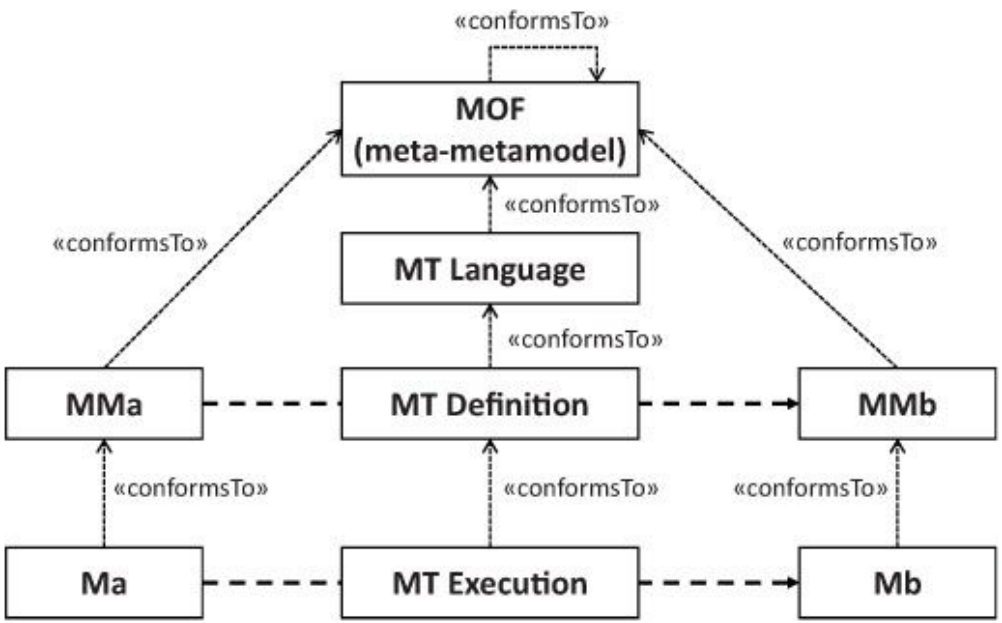
Metamodels

- Define the abstract syntax of a modelling language, whose extension corresponds to the whole class of models that can be represented by using that language.
- A model is said to conform to a metamodel if it belongs to the extension of the associated modelling language.



Transformations

- Goals:
  - model interpretation: a generic engine parses and executes the model
  - model translation:
    - to obtain running code from a higher level model
    - to reuse/extend existing libraries and frameworks in a particular domain
- Model-to-model (M2M):



- Model-to-text (M2T):
  - to automate the transition from the model level to the code level
  - often used in conjunction with M2M transformations

References

Marco Brambilla, Jordi Cabot, and Manuel Wimmer.
Model-Driven Software Engineering in Practice.
Morgan & Claypool Publishers, 1st edition, 2012.

Krzysztof Czarnecki and Ulrich W. Eisenecker.
Generative Programming: Methods, Tools, and Applications.
ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

Martin Fowler and Rebecca Parsons.
Domain-specific languages.
Addison-Wesley, Upper Saddle River, NJ, 2011.

M. Freudenthal.
Domain specific languages in a customs information system.
Software, IEEE, PP(99):1–1, 2009.

Debasish Ghosh.
DSLs in action.
Manning, Greenwich, Conn, 2011.

Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth.
DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.
dslbook.org, 2013.