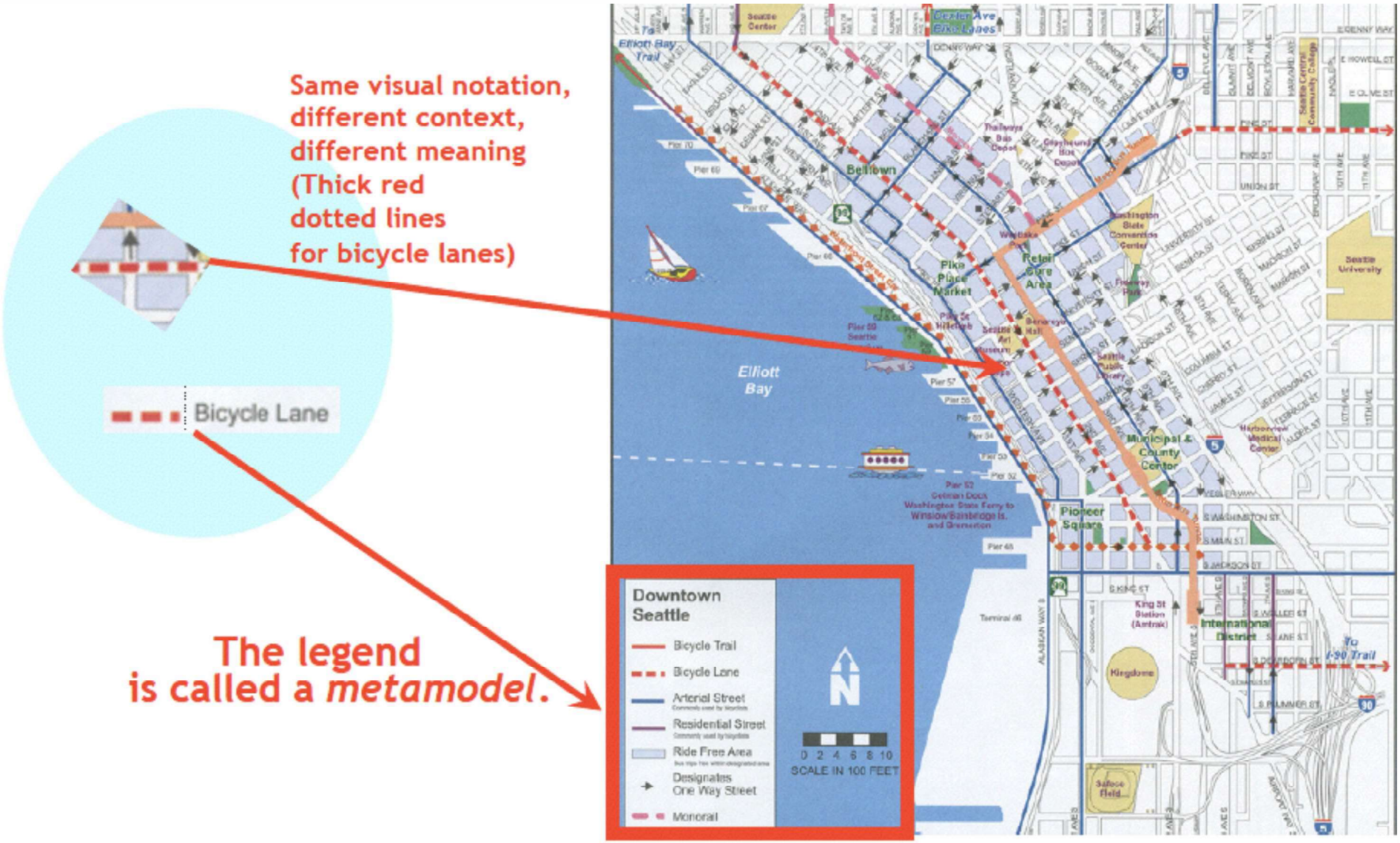
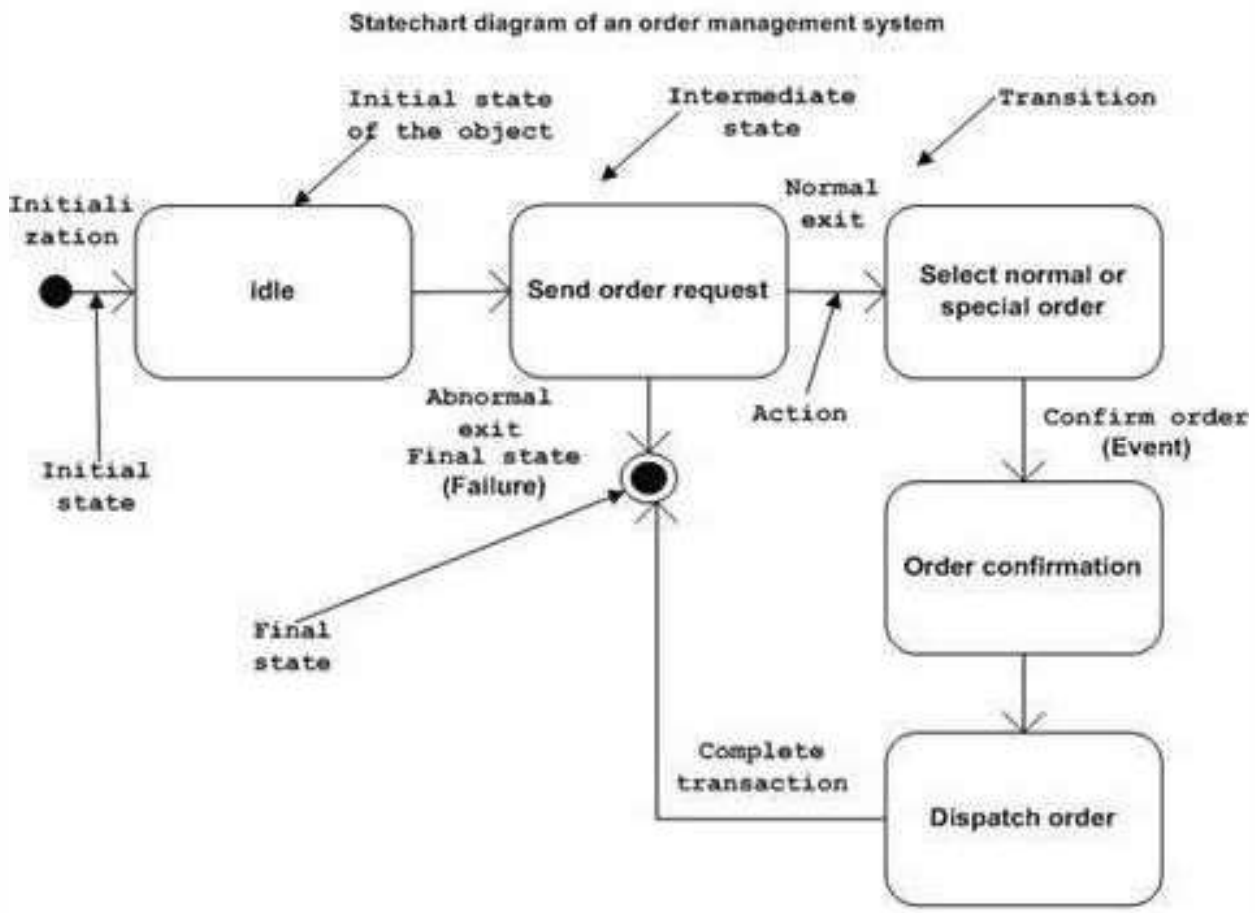
Model Driven Engineering

Second: Metamodels

Example: *Maps*



Example: *State Diagrams*



- How do we know this is a syntactically valid state diagram?
- The **metamodel** of the state diagram language expresses the rules that distinguish **valid** from **invalid.**

So, then, what is a metamodel?

- A description of the abstract syntax of a modelling language.
  - Models are instances.
  - In programming languages, these are sentences.
- What is an abstract syntax?
  - The language concepts, relationships between concepts and constraints.
  - Not usually the tokens, symbols, etc.… (these are part of the concrete syntax).

Example: XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<library>
  <book title="EMF Eclipse Modeling Framework" pages="744">
    <author>Dave Steinberg</author>
    <author>Frank Budinsky</author>
    <author>Marcelo Paternostro</author>
    <author>Ed Merks</author>
    <published>2009</published>
  </book>
  <book title="Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit" pages="736">
    <author>Richard Gronback</author>
    <published>2009</published>
  </book>
  <book title="Official Eclipse 3.0 FAQs" pages="432">
    <author>John Arthorne</author>
    <author>Chris Laffra</author>
    <published>2004</published>
  </book>
</library>
```

What was that?

- That was an **XML specification** of a very simple library, holding books (with authors), recording the page length of the book.
- We can load this into an XML editor, query it, change it, etc.
- It is **a model** of a simple library.
- But, how do we know if our XML model is **valid**?
  - We need **rules** to check it against!
  - One way to do this is using an **XML Schema (XSD)**.

Example - XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.example.com/Library"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
   xmlns:lib="http://www.example.com/Library" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:complexType name="Book">
      <xsd:sequence>
         <xsd:element name="title" type="xsd:string"/>
         <xsd:element name="pages" type="xsd:int"/>
         <xsd:element name="published" type="xsd:int"/>
         <xsd:element name="author" type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
<xsd:complexType name="Library">
      <xsd:sequence>
         <xsd:element name="name" type="xsd:string"/>
         <xsd:element maxOccurs="unbounded" minOccurs="0"
            name="books" type="lib:Book"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

What was that?

- A **Metamodel**!
- We can now check the library.xml model against the library.xsd Schema.
- The Schema contains rules that specify the **abstract syntax** of a valid library model.
- If the XML is valid (it is by the way) then we say it **conforms to** the XSD.
- Dually, we can say that the XML is **a valid instance of** the XSD.

Metamodels are Models themselves

- Metamodels are also models.
  - This is the so-called **unification property of MDE**:
    - everything's a model!
- So, in principle, models, metamodels and tasks applied to models can be implemented and managed using one set of tools

Interested in the formal definitions already?

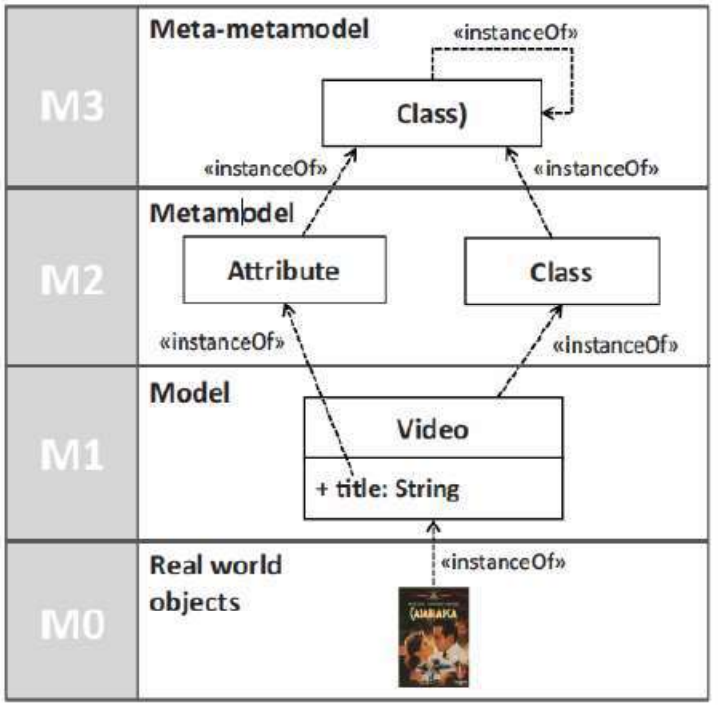
- A model M is a triple (G,ω,μ) where:
  - G is a directed multigraph,
  - ω is a reference model (metamodel) associated with a (potentially different) directed multigraph Gω,
  - μ is a function associating elements of G to nodes of Gω.
- The relationship between model and reference model is called **conformance**.

Conformance

- These definitions allow arbitrary levels of conformance!
- But, in practice, only **three** are used:
  - Terminal model (M1)
  - Metamodel (M2)
  - Metametamodel (M3)
- Examples:
  - **RDBMS**: instances (M1), schemas (M2), relational data model (M3)
  - **XML**: documents (M1), schemas/XSD (M2), schema definitions (M3)

OMG Standard

- A (**terminal) model** is a model such that its reference model is a metamodel.
  - Example: a UML class diagram
- A **metamodel** is a model such that its reference model is a metametamodel.
  - Example: the UML metamodel
- A **metametamodel** is a model that is its own reference model.
  - Example: MOF (OMG's MetaObject Facility)



Metamodelling *Example*: TED Conference Management

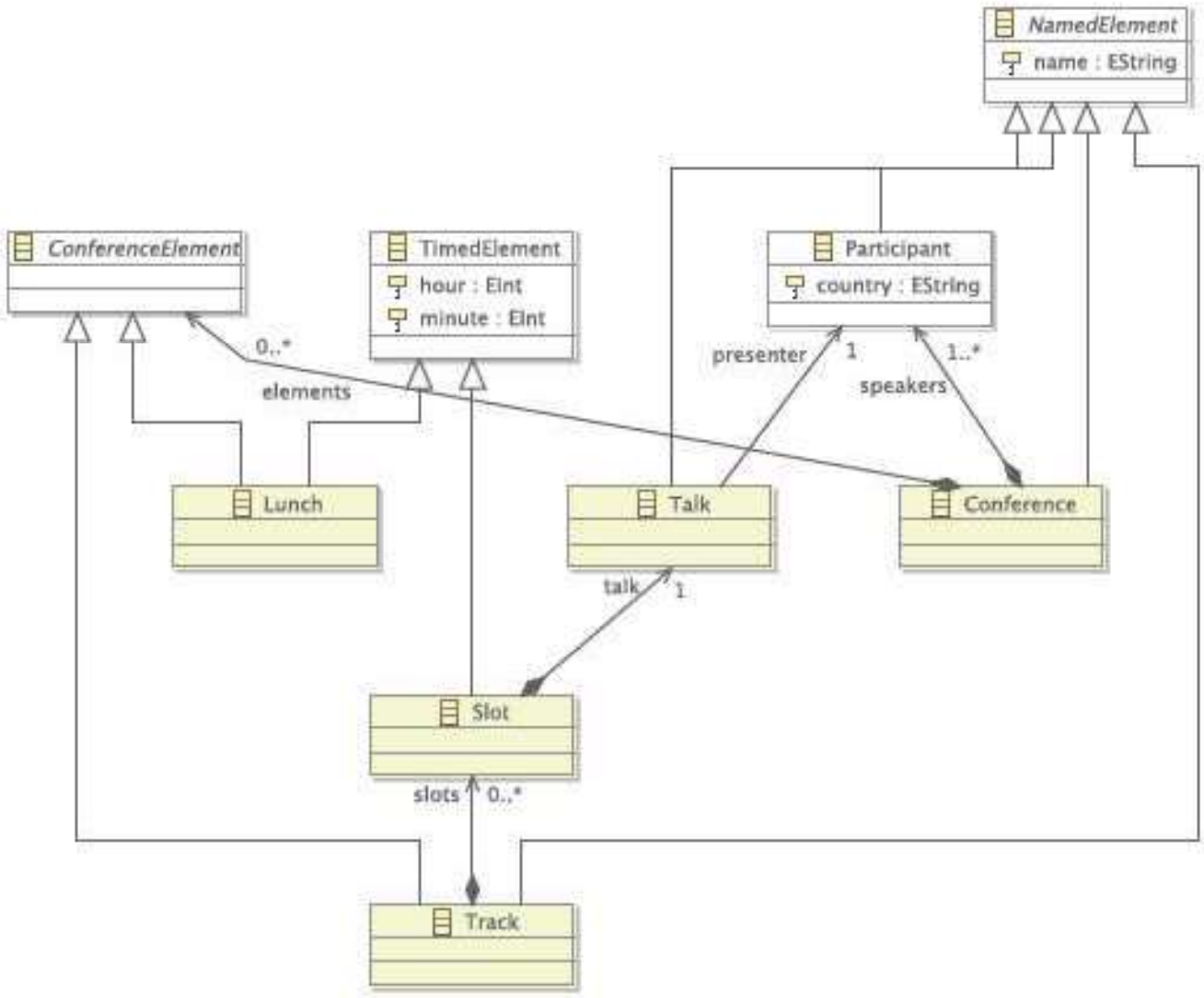
- Develop a customized editor for domain experts (conference managers).
- Lets domain experts build conference models that take into account important conference timetabling concepts.
- Follow standard metamodelling process.

    1. Abstract Syntax
    2. Constraints
    3. Concrete Syntax

1. Abstract Syntax

- Key domain concepts:
  - **Tracks**, consisting of a number of slots in which talks can be scheduled.
  - **Talks** have **participants** (who may have to give several talks, so we must avoid clashes)
  - **Lunch**
- In defining an abstract syntax we identify **recurring concepts**, including naming and timing (abstract these).

Abstract Syntax



2. Constraints

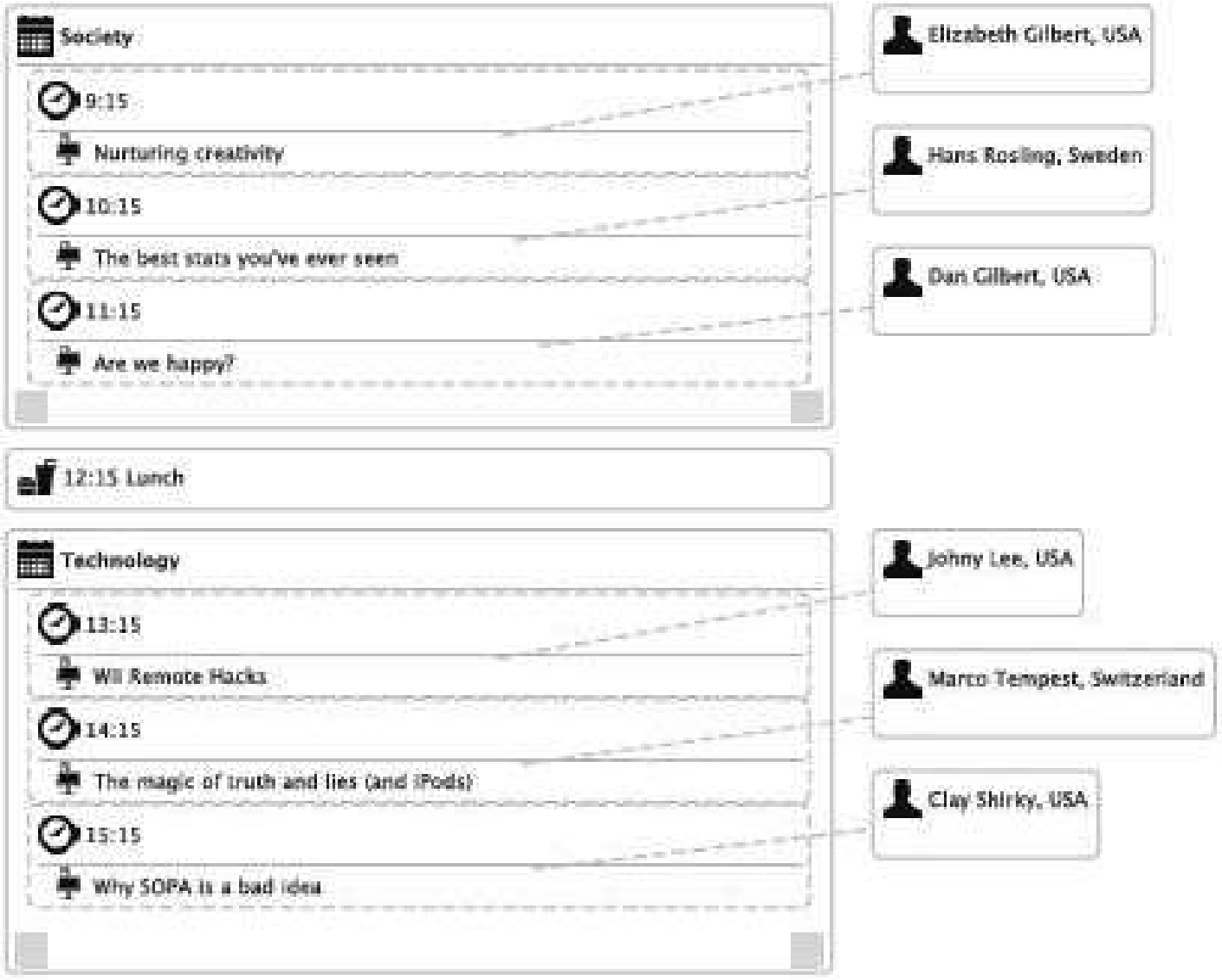
- Constraints to eliminate (as much as possible) ill-formed models.
- Example: each speaker at a conference is a presenter of a talk scheduled in a slot. (That is, we haven't missed anyone!)
- Express in **OCL (Object Constraint Language)**

```
context Conference inv:

    self.speakers->includes(

        self.elements->select(t|Track).

            slots.talk.presenter)
```

3. Concrete Syntax

- Design of concrete syntax is typically done in collaboration with end-users.
  - Identify their preferences (textual, graphical).
- What might an example concrete syntax look like?

Concrete Syntax Option 1



Concrete Syntax Option 2

CONFERENCE "TED"

TRACK "Society" :
  AT 09:15 : **TALK** "Nurturing creativity" **PRESENTED BY** "Elizabeth Gilbert"
  AT 10:15 : **TALK** "The best stats you've ever seen"
    PRESENTED BY "Hans Rosling"
  AT 11:15 : **TALK** "Are we happy?" **PRESENTED BY** "Dan Gilbert"

AT 12:15 **LUNCH**

TRACK "Technology" :
  AT 13:15 : **TALK** "Wii Remote Hacks" **PRESENTED BY** "Johny Lee"
  AT 14:15 : **TALK** "The magic of truth and lies (and iPods)"
    PRESENTED BY "Marco Tempest"
  AT 15:15 : **TALK** "Why SOPA is a bad idea" **PRESENTED BY** "Clay Shirky"

REGISTERED SPEAKERS :
  "Elizabeth Gilbert" **FROM** USA,
  "Hans Rosling" **FROM** Sweden,
  "Dan Gilbert" **FROM** USA,
  "Johny Lee" **FROM** USA,
  "Marco Tempest" **FROM** Switzerland,
  "Clay Shirky" **FROM** USA

Metamodeling: Summary

- Metamodelling is at the heart of MDE.
- Without a metamodel, it can be challenging (if not impossible) to automatically manage diverse models in systematic, repeatable, validated ways.

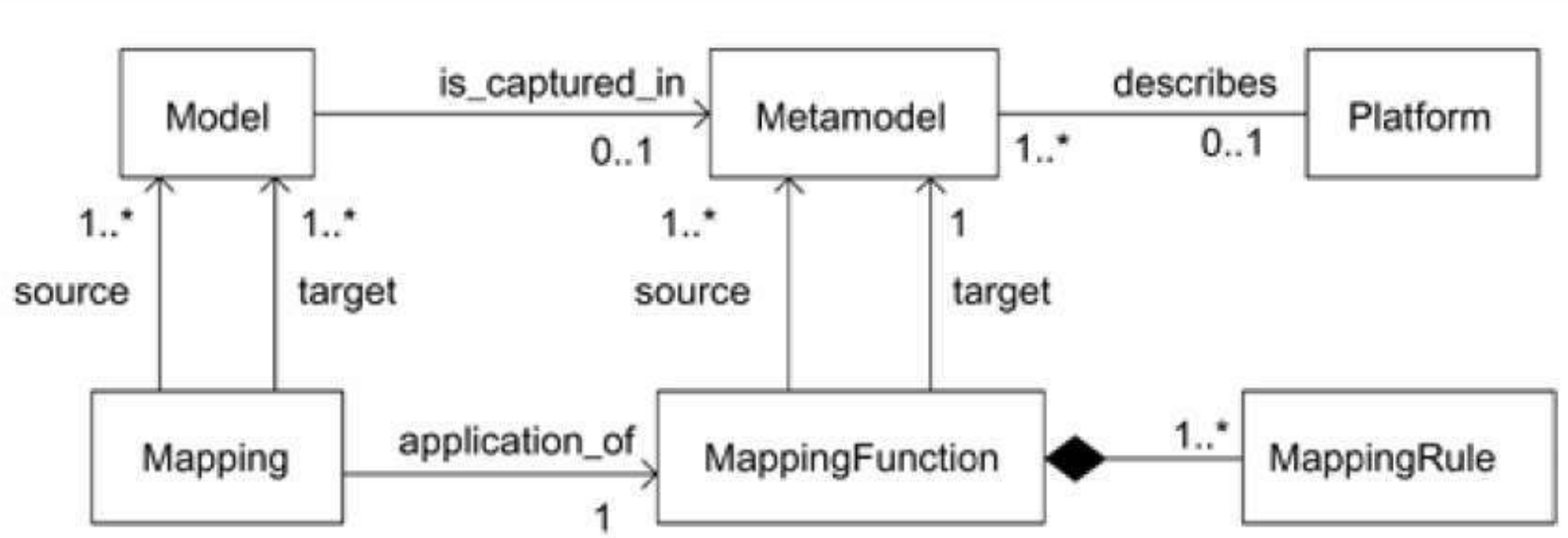
Model Driven Engineering

Third: Model Mappings

What is a Model Mapping?

- A mapping between models takes as input one or more **source** models and produces one **target** model as output.
- **Mapping rules** constrain the mapping through a **mapping function**.
- The rules on model mappings are defined at the metamodel level and apply to all sets of source models that conform to the given metamodel.

Summary:
Models, Metamodels and Mappings



Model Driven Engineering

Fourth: Modeling Languages and Notations

Modelling Languages and Notations

- In order to successfully apply MDE, modelling languages are required.
- A Modelling Language is a tool that allows designers to specify the models in their system. Specification can be: graphical, textual or both.
- In all cases, models should be formally defined and should comply with the modelling language's syntax.
- **Two groups** of modeling languages:
  - Domain Specific Languages (DSLs)
  - General Purpose Languages (GPLs)

Domain Specific Languages (DSLs)

- Languages that are designed specifically for a certain domain or company to use in modelling their systems.
- **Examples in SE:**
  - HTML for webpage development
  - SQL for databases
  - VHDL for hardware
  - MatLab for mathematics and real-time systems.

General Purpose Languages (GPLs)

- General Purpose (modelling) Languages (GPLs) represent general tools that can be applied to any domain or industrial sector (be it automotive, health, etc) for the purpose of modelling their systems.
- Examples:
  - UML
  - Petri Nets
- Choosing whether to use a GPL or a DSL is not always clear.
  - UML is better suited for mainly object-oriented software systems.
  - UML does not serve well for some domains like those that involve user interaction design, and a DSL could work better there.

Model Driven Engineering

Fifth: Model Transformations

Model Transformations

- Model transformations allow passing relevant information from one modelling formalism to another
- The “heart and soul” of model-driven software development.
- Multiple uses in MDE:
  - example: transform a UML statechart into code
  - example: transform a statechart into a formalism amenable to verification by some existing tool
- Exist in traditional software development, although implicitly.
- Model Transformation Languages Examples: ATL, QVT

Classification

- **One-to-One** (most cases) vs. **Many-to-One** (e.g., model merging)
- **Endogenous**: source and target same type vs.
  Exogenous: source and target different types
- **Out-place:** create an output model from scratch vs.
  In-place: rewrite input model
- **Horizontal**: operate on models at the same level of abstraction (e.g., model migration) vs.
  Vertical: operate on different levels of abstraction (e.g., model refinement)
- **Syntactical** vs. **Semantical**

Model to Model (M2M)

- Models do not exist in isolation in a system
- As part of the MDE process, Model-to-Model Transformations (M2M) are applied on models so they can be:
  - **merged** (e.g., to homogenize different versions of a systems)
  - **aligned** (e.g., to create a global representation of the system from different views to reason about consistency)
  - **refactored** (to improve their internal structure without changing their observable behaviour)
  - **refined** (to detail high-level models)
  - **translated** (to other languages/representations, e.g., as part of code-generation or verification/simulation processes).

Model to Text (M2T)

- Model-to-Text Transformations are used in the activity of Model Driven Code Generation.
- Also used in automating other software engineering tasks such as the generation of documentation from the models.
- Analogously, **Text-to-Model (T2M)** Transformations have a text string as input and a model as output. Such transformations are typically applied in model driven reverse-engineering.

Incremental Model Transformations

- **Batch Transformations:** transform every time the entire input model to a new output model created from scratch.
- **Incremental Model Transformations:** consider the differences between the current input model and the input model used in the last transformation run, as to minimize the changes to be done on the output model which already exists from the previous transformation.
- **Example:** If a new element is added to the input model, only the rules that are a match for the new element will be executed to update the output model.
- **Advantages**:
  - More efficient transformations
  - Changes applied to output model are preserved

Bidirectional Model Transformations

- Bidirectional Model Transformations are transformation that have a **forward** direction (from source to target), and a **backward** direction (from target to source).
- Important for ensuring model **consistency**
- Used for the purpose of model **synchronization**.
- Many bidirectional transformation languages still lack a proper understanding of the semantic implications, which hampers their use in practice.
- Some of the more successful bidirectional transformation languages out there are QVT, TGG and JTL.

Higher Order Transformations (HOTs)

- The same tools that are used for creating models, can be used for creating transformation models.
- This creates a recursive framework, meaning *transformations of transformations can be transformed themselves*.
- Just as regular models can be created, modified and augmented through transformations, we can use so called **Higher Order Transformations (HOTs)** to create, modify and augment transformation models.
- **HOTs** therefore take as input one or more model transformations and/or generate as output a model transformation.

Analysis of Model Transformations

- To be able to guarantee certain **properties** of a produced artifact, it may be very helpful, or even important, to also have knowledge of the producing transformation.
- The Analysis of Model Transformations involves looking at techniques that help ensure that model transformations produce models of sufficient quality and with desirable properties.

Verification of Model Transformations

- Important for the **quality** of the whole software development process that transformations be **correct**.
- Model transformations verification is similar to the verification of software.
- Typically done by **proving** that a given program has certain formal properties that ensure a certain level of correctness regarding that programs specification.
- **Two types** of approaches:
  - Proving that certain relations hold between the grammar of the input models and the grammar of the output model. (**Syntactic**)
  - Proving that parts of the meaning, or semantics of input models are given the correct semantics in the output model. (**Semantic**)

Model Transformation Languages and Tools

- **Declarative** transformation languages
  - generally limited to scenarios where the source and target metamodels are similar to each other in terms of **structure**
  - the transformation is a matter of **a simple mapping**
  - **sometimes** difficult to address cases where significant processing and complex mappings are involved.
- **Imperative** transformation languages
  - capable of addressing a wider range of transformation scenarios.
  - operate at a lower level of abstraction which means that users need to **manually** address issues such as tracing and resolving target elements from their source counterparts and orchestrating the transformation execution.
- **Hybrid languages**
  - ATL and QVT
  - provide both a declarative rule-based execution scheme as well as imperative features for handling complex transformation scenarios