Model Driven Architecture
Metamodeling — Applications

Prof. Dr. Peter Thiemann

Universität Freiburg

31.05.2006
Feature models are a tool for domain analysis
  - Provide a hierarchical view of features and their dependencies
  - Establish an ontology for categorization

Visualized by feature diagrams


Popularized for Generative Programming by Czarnecki and Eisenäcker

Also for analyzing other domains
Hierarchical, but **not** is-a relation (as in a class diagram)

Features may be qualified as required, optional, alternative, or *n-of-m* (selection)
Feature Modeling
Feature Model in Abstract Syntax

additionalFeatureSFG
FM::SubfeatureGroup
kind="optional"
AdditionalFeature
FM::Feature
name="AdditionalFeatures"
addFeatureTwoSFG
FM::SubfeatureGroup
kind="nOfM"
threadFeature
FM::Feature
name="ThreadSafety"

stackFeature:
FM::Concept

optimizationDFG
FM::SubfeatureGroup
kind="optional"
Optimization
FM::Feature
name="Optimization"

optimizationFeature
FM::Feature
name="Optimization"
optimizationKindSFG
FM::SubfeatureGroup
kind="alternative"

typeFeature
FM::Feature
name="TypeCheck"
speedFeature
FM::Feature
name="Speed"

boundsFeature
FM::Feature
name="BoundsCheck"

memoryFeature
FM::Feature
name="MemoryUsage"
New feature ⇒
- new attribute in metamodel
- new slot in model
- extension of concrete syntax
Macros for combinations of features

Multiplicity of subfeatures

Priority of features; stakeholders

Open / closed

Additional constraints (without graphical notation)

- *requires*: feature dependency
- *excludes*: feature anti-dependency
- *recommends*: soft dependency
- *discourages*: soft anti-dependency

Example: Optimization for speed *discourages* thread safety, *discourages* bounds check
Domain specific modeling language for small and embedded systems

Main abstraction: component

A component may

- provide services via interfaces
- require services via interfaces
- have configuration parameters
- be an application (does not provide services)
Pitfalls in Metamodeling

How to avoid
- confusion with UML notation
- mixing metalevels

Central question
- what is the mapping to a programming language?
Interfaces
Every instance of Entity should implement SomeInterface

- wrong approach

- book solution use OCL or subsetting of metaassociation
Every instance of `Entity` should implement `SomeInterface`.

- **Correct solution** use OCL

  ```
  realization
  -->select(hasStereotype("interface"))
  -->select(name="SomeInterface")
  -->size() = 1
  ```

- **Solution with metaassociation and OCL** is also possible
Problem: A Component may depend from multiple Interfaces because the Component may invoke operations of the Interfaces.

wrong approach “metaclass Component depends on metaclass Interface”

correct solution a metaassociation “uses”
An **Entity** must have an identifying attribute with name **ID** and type **String**. **Entity** is a subclass of **UML::Class**.

- wrong approach

<table>
<thead>
<tr>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID : String</td>
</tr>
</tbody>
</table>

defines a tagged value **IF** for all **Entity** instances in the model
**Identifying Attribute**

- **correct solution**

  - There must be exactly one attribute with name `ID`
  - All attributes named `ID` must have type `String`
there must be exactly one attribute with name ID
all attributes named ID must have type String

incorrect attempt

context Entity inv:
 Attribute
    ->select (Name="ID" and Type.Name="String")
    ->size() = 1
Each instance of **Entity** must have exactly one attribute of type **EntityPK**, where **EntityPK** is a subclass of **Attribute**.

- wrong approach
- correct solution
Objects are instances of classes

Links are instances of associations
The **Auto** and **Person** classes are instances of the MOF metaclass **UML::Class**

- The objects **me:** and **myFather:** are instances of the MOF metaclass **UML::Object**
Metalevels and Instanceof
A Look at the Metamodel

- two different instanceof relations