Software Design, Modelling and Analysis in UML

Lecture 19: Inheritance II, Meta-Modelling

2012-02-08

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:
- Live Sequence Charts Semantics

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What’s the Liskov Substitution Principle?
  - What is late/early binding?
  - What is the subset, what the uplink semantics of inheritance?
  - What’s the effect of inheritance on LSCs, State Machines, System States?
  - What’s the idea of Meta-Modelling?

- Content:
  - Inheritance in UML: concrete syntax
  - Liskov Substitution Principle — desired semantics
  - Two approaches to obtain desired semantics
Desired Semantics of Specialisation: Subtyping

There is a classical description of what one expects from sub-types, which in the OO domain is closely related to inheritance:

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994]. (Liskov Substitution Principle (LSP).)

“If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \),

- the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \),

then \( S \) is a subtype of \( T \)."

\[
S \text{ sub-type of } T \iff \forall o_1 \in S \exists o_2 \in T : \forall P \cdot [P]_T(o_1) = [P]_T(o_2)
\]
Desired Semantics of Specialisation: Subtyping

There is a classical description of what one expects from sub-types, which in the OO domain is closely related to inheritance:

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994].

(Liskov Substitution Principle (LSP).)

“If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \), then \( S \) is a subtype of \( T \).”

In other words: [Fischer and Wehrheim, 2000]

“An instance of the sub-type shall be usable whenever an instance of the supertype was expected, without a client being able to tell the difference.”
**Desired Semantics of Specialisation: Subtyping**

There is a classical description of what one *expects* from sub-*types*, which in the OO domain is closely related to inheritance:

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994].

(Liskov Substitution Principle (LSP).)

“If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \), then \( S \) is a **subtype** of \( T \).”

In other words: [Fischer and Wehrheim, 2000]

“An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected, without a client being able to tell the difference.”

So, what’s “**usable**”? Who’s a “**client**”? And what’s a “**difference**”? 
What Does [Fischer and Wehrheim, 2000] Mean for UML?

"An instance of the sub-type shall be usable whenever an instance of the supertype was expected, without a client being able to tell the difference."

- Wanted: sub-typing for UML.
- With

```
  C
  |
  v

  C
  |
  v

  D_1
  |
  v

  D_1
  |
  v

  D_2
```

we don’t even have usability.

- It would be nice, if the well-formedness rules and semantics of

```
  D_1
  |
  v

  D_2
```

would ensure \( D_1 \) is a sub-type of \( C \):

- that \( D_1 \) objects can be used interchangeably by everyone who is using \( C \)'s,
- is not able to tell the difference (i.e. see unexpected behaviour).

"...shall be usable..." for UML
Easy: Static Typing

Given:

\[
C_1 \xrightarrow{\text{itsC1}} (x: \text{Int}) \rightarrow \text{Int}
\]

\[
D_1 \xrightarrow{\text{itsD1}} (x: \text{Int}) \rightarrow \text{Int}
\]

\[
C_2 \xrightarrow{\text{itsC2}} (x: \text{Int}) \rightarrow \text{Int}
\]

\[
D_2 \xrightarrow{\text{itsD2}} (x: \text{Bool}) \rightarrow \text{Int}
\]

\[
\langle \langle \text{signal} \rangle \rangle E \xrightarrow{\text{itsE}} \langle \langle \text{signal} \rangle \rangle F
\]

Wanted:

• \(x > 0\) also well-typed for \(D_1\)

• assignment \(\text{itsC1} := \text{itsD1}\) being well-typed

• \(\text{itsD1}.x = 0, \text{itsD1}.f(0), \text{itsD1} \not\rightarrow F\)

being well-typed (and doing the right thing).

Approach:

• Simply define it as being well-typed,

• adjust system state definition to do the right thing.

\[
\text{e.g. } D_2 \xrightarrow{\text{can't be \(D_1\) in: \(x > 0\)}}
\]

\[
\text{e.g. } x \not\rightarrow 0
\]

\[
\text{Static Typing Cont'd}
\]

Notions (from category theory):

• invariance,

• covariance,

• contravariance.

We could call, e.g. a method, sub-type preserving, if and only if it

• accepts more general types as input (contravariant),

• provides a more specialised type as output (covariant).

This is a notion used by many programming languages — and easily type-checked.
**Excursus: Late Binding of Behavioural Features**

**Late Binding**

What transformer applies in what situation?

(Early (compile time) binding.)

| Type of object determines which implementation is used (not caring for what the object really is) | f not overridden in D | f overridden in D | Value of `someC`/
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>someC</code> -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
<td>u1: C</td>
</tr>
<tr>
<td><code>someD</code> -&gt; f()</td>
<td>C::f()</td>
<td>D::f()</td>
<td>u2: D</td>
</tr>
<tr>
<td><code>someC</code> -&gt; f()</td>
<td>D::f()</td>
<td>C::f()</td>
<td>v1: D</td>
</tr>
</tbody>
</table>

What one could want is something different: (Late binding.)

| Type of object determines which implementation is used | f not overridden in D | f overridden in D | Value of `someC`/
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>someC</code> -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
<td>u1: C</td>
</tr>
<tr>
<td><code>someD</code> -&gt; f()</td>
<td>C::f()</td>
<td>D::f()</td>
<td>u2: D</td>
</tr>
<tr>
<td><code>someC</code> -&gt; f()</td>
<td>D::f()</td>
<td>D::f()</td>
<td>v1: D</td>
</tr>
</tbody>
</table>
Late Binding in the Standard and Programming Lang.

- In the standard, Section 11.3.10, “CallOperationAction”:
  
  "Semantic Variation Points
  The mechanism for determining the method to be invoked as a result of a call operation is unspecified." [OMG, 2007b, 247]

- In C++,
  - methods are by default “(early) compile time binding”,
  - can be declared to be “late binding” by keyword “virtual”,
  - the declaration applies to all inheriting classes.

- In Java,
  - methods are “late binding”;
  - there are patterns to imitate the effect of “early binding”

Exercise: What could have driven the designers of C++ to take that approach?

Note: late binding typically applies only to methods, not to attributes.
(But: getter/setter methods have been invented recently.)

Back to the Main Track: “...tell the difference...” for UML
**With Only Early Binding...**

- ...we’re **done** (if we realise it correctly in the framework).
- Then
  - if we’re calling method \( f \) of an object \( u \),
  - which is an instance of \( D \) with \( C \triangleleft D \)
  - via a \( C \)-link, \( \text{ \( C \)-part will be used} \)
  - then we (by definition) only see and change the \( C \)-part.
  - We cannot tell whether \( u \) is a \( C \) or an \( D \) instance.

So we immediately also have behavioural/dynamic subtyping.

---

**Difficult: Dynamic Subtyping**

- \( C::f \) and \( D::f \) are **type compatible**, but \( D \) is **not necessarily** a **sub-type** of \( C \).

- **Examples: (C++)**

```
int C::f(int) {
    return 0;
}
```

vs.

```
int D::f(int) {
    return 1;
}
```
Sub-Typing Principles Cont’d

- In the standard, Section 7.3.36, “Operation”:
  “Semantic Variation Points
  [...] When operations are redefined in a specialization, rules regarding
  invariance, covariance, or contravariance of types and preconditions
determine whether the specialized classifier is substitutable for its more
general parent. Such rules constitute semantic variation points with
respect to redefinition of operations.” [OMG, 2007a, 106]

- So, better: call a method sub-type preserving, if and only if it
  (i) accepts more input values (contravariant),
  (ii) on the old values, has fewer behaviour (covariant).

  Note: This (ii) is no longer a matter of simple type-checking!

- And not necessarily the end of the story:
  - One could, e.g. want to consider execution time.
  - Or, like [Fischer and Wehrheim, 2000], relax to “fewer observable
    behaviour”, thus admitting the sub-type to do more work on inputs.

  Note: “testing” differences depends on the granularity of the semantics.

- Related: “has a weaker pre-condition,” (contravariant),
  “has a stronger post-condition.” (covariant).

Ensuring Sub-Typing for State Machines

- In the CASE tool we consider, multiple classes
  in an inheritance hierarchy can have state machines.

- But the state machine of a sub-class cannot be drawn from scratch.

- Instead, the state machine of a sub-class can only be obtained by
  applying actions from a restricted set to a copy of the original one.

  Roughly (cf. User Guide, p. 760, for details),
  - add things into (hierarchical) states,
  - add more states,
  - attach a transition to a different target (limited).

- They ensure, that the sub-class is a behavioural sub-type of the super
  class. (But method implementations can still destroy that property.)

- Technically, the idea is that (by late binding) only the state machine of the most
  specialised classes are running.

  By knowledge of the framework, the (code for) state machines of super-classes is still
  accessible — but using it is hardly a good idea...
Towards System States

Wanted: a formal representation of “if $C \subseteq D$ then $D$ `is a' $C$”, that is,

(i) $D$ has the same attributes and behavioural features as $C$, and
(ii) $D$ objects (identities) can replace $C$ objects.

We’ll discuss two approaches to semantics:

- **Domain-inclusion Semantics**
- **Uplink Semantics**

Meta-Modelling: Idea and Example
Meta-Modelling: Why and What

- **Meta-Modelling** is one major prerequisite for understanding
  - the standard documents [OMG, 2007a, OMG, 2007b], and
  - the MDA ideas of the OMG.

- The idea is simple:
  - if a modelling language is about modelling things,
  - and if UML models are and comprise things,
  - then why not model those in a modelling language?

- In other words:
  Why not have a model $M_U$ such that
  - the set of legal instances of $M_U$
    is
  - the set of well-formed (!) UML models.

Meta-Modelling: Example

- For example, let’s consider a class.

- A class has (on a superficial level)
  - a name,
  - any number of attributes,
  - any number of behavioural features.

  Each of the latter two has
  - a name and
  - a visibility.

  Behavioural features in addition have
  - a boolean attribute isQuery,
  - any number of parameters,
  - a return type.

- Can we model this (in UML, for a start)?
UML Meta-Model: Extract

Classes [OMG, 2007b, 32]

Figure 7.12 - Classes diagram of the Kernel package
Classifiers [OMG, 2007b, 29]

![Classifier diagram of the Kernel package](image)

Namespaces [OMG, 2007b, 26]

![Namespaces diagram of the Kernel package](image)

Figure 7.3 - Root diagram of the Kernel package

Interesting: Declaration/Definition  [OMG, 2007b, 424]

Figure 13.6 - Common Behavior
UML Architecture [7, 8]

- Meta-modelling has already been used for UML 1.x.

- For UML 2.0, the request for proposals (RFP) asked for a separation of concerns: Infrastructure and Superstructure.

- One reason: sharing with MOF (see later) and, e.g., CWM.

UML Superstructure Packages [OMG, 2007a, 15]

Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure
Meta-Modelling: Principle

**Modelling vs. Meta-Modelling**

Meta-Model (M2)

- Class
  - name: Str
- Property
  - name: Str
- Type
  - name: Str

Model (M1)

- Class
  - name = C
- Property
  - name = v
- Type
  - name = Z

Instance (M0)

- C
  - v = 0

\[ \mathcal{C} = \{ \{ Z \}, \{ C \}, \{ v \}, \{ C \mapsto v \} \}, \mathcal{D} 
\overset{\sigma}{\sim} \mathcal{\Sigma}_{\mathcal{D}} \]
Modelling vs. Meta-Modelling

Meta-Model (M2)

Class
name : Str

Property
name : Str

Type
name : Str

Model (M1)

\[ C \]

\[ \nu : \mathbb{Z} \]

\[ \text{Class} \]
name = C

\[ \text{Property} \]
name = \nu

\[ \text{Type} \]
name = \mathbb{Z}

\[ \mathcal{S} = (\{\mathbb{Z}\}, \{C\}, \{\nu\}, \{C \mapsto \nu\}), \mathcal{P} \rightsquigarrow \Sigma \]

Instance (M0)

- So, if we have a meta model \( M_U \) of UML, then the set of UML models is the set of instances of \( M_U \).
- A UML model \( M \) can be represented as an object diagram (or system state) wrt. the meta-model \( M_U \).
- Other view: An object diagram wrt. meta-model \( M_U \) can (alternatively) be rendered as the UML model \( M \).

Well-Formedness as Constraints in the Meta-Model

- The set of well-formed UML models can be defined as the set of object diagrams satisfying all constraints of the meta-model.

For example,

"[2] Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.

\[ \text{not self . allParents()} \rightarrow \text{includes(self)} \] [OMG, 2007b, 53]"

- The other way round:
  Given a UML model \( M \), unfold it into an object diagram \( O_1 \) wrt. \( M_U \).
  If \( O_1 \) is a valid object diagram of \( M_U \) (i.e. satisfies all invariants from \( \text{Inv}(M_U) \)), then \( M \) is a well-formed UML model.

That is, if we have an object diagram validity checker for of the meta-modelling language, then we have a well-formedness checker for UML models.
# Reading the Standard

## Table of Contents

1. Scope ........................................... 1  
2. Conformance .................................... 1  
   2.1 Language Units ................................ 2  
   2.2 Compliance Levels .............................. 2  
   2.3 Meaning and Types of Compliance .......... 6  
   2.4 Compliance Level Contents .................. 8  
3. Normative References .......................... 10  
4. Terms and Definitions ........................... 10  
5. Symbols .......................................... 10  
6. Additional Information .......................... 10  
   6.1 Changes to Adopted OMG Specifications ... 10  
   6.2 ArchitecturalAlignment and MDA Support .... 10  
   6.3 On the Run-Time Semantics of UML ......... 11  
   6.4 The UML Metamodel ......................... 13  
   6.5 How to Read this Specification ............. 15  
   6.6 Acknowledgements ............................ 19  
7. Classes ......................................... 23  
8. Part I - Structure .................................. 21  
9. Part II - Semantics ................................ 23

## Reading the Standard

| 1. Scope │  |
|----------|---|
| 2. Conformance │ 1 |
| 3. Normative References │ 10 |
| 4. Terms and Definitions │ 10 |
| 5. Symbols │ 10 |
| 6. Additional Information │ 10 |
| 7. Classes │ 23 |
| 8. Part I - Structure │ 21 |
| 9. Part II - Semantics │ 23 |

**Table of Contents**

1. Scope .......................... 1  
2. Conformance .......................... 1  
   2.1 Language Units .......................... 2  
   2.2 Compliance Levels ......................... 2  
   2.3 Meaning and Types of Compliance ......... 6  
   2.4 Compliance Level Contents ................. 8  
3. Normative References ............... 10  
4. Terms and Definitions ............... 10  
5. Symbols ..................................... 10  
6. Additional Information ............... 10  
   6.1 Changes to Adopted OMG Specifications .... 10  
   6.2 Architectural Alignment and MDA Support .... 10  
   6.3 On the Run-Time Semantics of UML ......... 11  
   6.4 The UML Metamodel ......................... 13  
   6.5 How to Read this Specification ............. 15  
   6.6 Acknowledgements .......................... 19  
7. Classes ..................................... 23  
8. Part I - Structure .......................... 21  
9. Part II - Semantics .......................... 23

| 7.1 Overview │ 23 |
| 7.2 Abstract Syntax │ 24 |
| 7.3 Class Descriptions │ 38 |
| 7.3.1 Abstraction (from Dependencies) │ 38 |
| 7.3.2 Aggregation (from Kernel) │ 38 |
| 7.3.3 Association (from Kernel) │ 38 |
| 7.3.4 AssociationEnd (from AssociationClasses) │ 38 |
| 7.3.5 AssociationEnd (from Kernel) │ 38 |
| 7.3.6 BehavedClassifier (from Interfaces) │ 49 |
| 7.3.7 BehavedClassifier (from Kernel) │ 49 |
| 7.3.8 Classifier (from Kernel, Dependencies, PowerTypes) │ 52 |
| 7.3.9 Constraint (from Kernel) │ 57 |
| 7.3.10 Constraint (from AssociationClasses) │ 57 |
| 7.3.11 ClassType (from Kernel) │ 58 |
| 7.3.12 Dependency (from Dependencies) │ 60 |
| 7.3.13 Dependency (from Kernel) │ 60 |
| 7.3.14 Element (from Kernel) │ 64 |
| 7.3.15 ElementReference (from Kernel) │ 64 |
| 7.3.16 Expression (from Kernel) │ 68 |
| 7.3.17 Feature (from Kernel) │ 72 |
| 7.3.18 Feature (from Kernel, PowerTypes) │ 71 |
| 7.3.19 Generalization (from Kernel, PowerTypes) │ 71 |
| 7.3.20 Generalization (from Kernel) │ 75 |
| 7.3.21 Interface (from Kernel) │ 78 |
| 7.3.22 Interface (from Interfaces) │ 78 |
| 7.3.23 InterfaceRealization (from Interfaces) │ 78 |
| 7.3.24 InterfaceRealization (from Kernel) │ 78 |
| 7.3.25 Multiplicity (from Kernel) │ 94 |
| 7.3.26 NamedElement (from Kernel, Dependencies) │ 97 |
| 7.3.27 NamedElement (from Kernel) │ 97 |
| 7.3.28 OperatorExpression (from Kernel) │ 101 |
| 7.3.29 Operation (from Kernel, Interfaces) │ 103 |
| 7.3.30 Package (from Kernel) │ 107 |
| 7.3.31 Package (from Kernel, PowerTypes) │ 107 |
| 7.3.32 Package (from Kernel) │ 110 |
| 7.3.33 Package (from Kernel, Dependencies) │ 110 |
| 7.3.34 Package (from Kernel) │ 116 |
| 7.3.35 Package (from Kernel, Dependencies) │ 116 |
| 7.3.36 Package (from Kernel) │ 120 |
| 7.3.37 Package (from Kernel, Dependencies) │ 120 |
| 7.3.38 Package (from Kernel) │ 123 |
| 7.3.39 Package (from Kernel, Dependencies) │ 123 |
| 7.3.40 Package (from Kernel) │ 125 |
| 7.3.41 Parameter (from Kernel) │ 129 |
| 7.3.42 Parameter (from Kernel, Dependencies) │ 129 |

**Table of Contents**

1. Scope .......................... 1  
2. Conformance .......................... 1  
   2.1 Language Units .......................... 2  
   2.2 Compliance Levels ......................... 2  
   2.3 Meaning and Types of Compliance ......... 6  
   2.4 Compliance Level Contents ................. 8  
3. Normative References ............... 10  
4. Terms and Definitions ............... 10  
5. Symbols ..................................... 10  
6. Additional Information ............... 10  
   6.1 Changes to Adopted OMG Specifications .... 10  
   6.2 Architectural Alignment and MDA Support .... 10  
   6.3 On the Run-Time Semantics of UML ......... 11  
   6.4 The UML Metamodel ......................... 13  
   6.5 How to Read this Specification ............. 15  
   6.6 Acknowledgements .......................... 19  
7. Classes ..................................... 23  
8. Part I - Structure .......................... 21  
9. Part II - Semantics .......................... 23

| 7.1 Overview │ 23 |
| 7.2 Abstract Syntax │ 24 |
| 7.3 Class Descriptions │ 38 |
| 7.3.1 Abstraction (from Dependencies) │ 38 |
| 7.3.2 Aggregation (from Kernel) │ 38 |
| 7.3.3 Association (from Kernel) │ 38 |
| 7.3.4 AssociationEnd (from AssociationClasses) │ 38 |
| 7.3.5 AssociationEnd (from Kernel) │ 38 |
| 7.3.6 BehavedClassifier (from Interfaces) │ 49 |
| 7.3.7 BehavedClassifier (from Kernel) │ 49 |
| 7.3.8 Classifier (from Kernel, Dependencies, PowerTypes) │ 52 |
| 7.3.9 Constraint (from Kernel) │ 57 |
| 7.3.10 Constraint (from AssociationClasses) │ 57 |
| 7.3.11 ClassType (from Kernel) │ 58 |
| 7.3.12 Dependency (from Dependencies) │ 60 |
| 7.3.13 Dependency (from Kernel) │ 60 |
| 7.3.14 Element (from Kernel) │ 64 |
| 7.3.15 ElementReference (from Kernel) │ 64 |
| 7.3.16 Expression (from Kernel) │ 68 |
| 7.3.17 Feature (from Kernel) │ 72 |
| 7.3.18 Feature (from Kernel, PowerTypes) │ 71 |
| 7.3.19 Generalization (from Kernel, PowerTypes) │ 71 |
| 7.3.20 Generalization (from Kernel) │ 75 |
| 7.3.21 Interface (from Kernel) │ 78 |
| 7.3.22 Interface (from Interfaces) │ 78 |
| 7.3.23 InterfaceRealization (from Interfaces) │ 78 |
| 7.3.24 InterfaceRealization (from Kernel) │ 78 |
| 7.3.25 Multiplicity (from Kernel) │ 94 |
| 7.3.26 NamedElement (from Kernel, Dependencies) │ 97 |
| 7.3.27 NamedElement (from Kernel) │ 97 |
| 7.3.28 OperatorExpression (from Kernel) │ 101 |
| 7.3.29 Operation (from Kernel, Interfaces) │ 103 |
| 7.3.30 Package (from Kernel) │ 107 |
| 7.3.31 Package (from Kernel, PowerTypes) │ 107 |
| 7.3.32 Package (from Kernel) │ 110 |
| 7.3.33 Package (from Kernel, Dependencies) │ 110 |
| 7.3.34 Package (from Kernel) │ 116 |
| 7.3.35 Package (from Kernel, Dependencies) │ 116 |
| 7.3.36 Package (from Kernel) │ 120 |
| 7.3.37 Package (from Kernel, Dependencies) │ 120 |
| 7.3.38 Package (from Kernel) │ 123 |
| 7.3.39 Package (from Kernel, Dependencies) │ 123 |
| 7.3.40 Package (from Kernel) │ 125 |
| 7.3.41 Parameter (from Kernel) │ 129 |
| 7.3.42 Parameter (from Kernel, Dependencies) │ 129 |
### Reading the Standard

#### Table of Contents

1. **Scope**
2. **Conformance**
   - 2.1 Language Units
   - 2.2 Compliance Levels
   - 2.3 Meaning and Types
   - 2.4 Compliance Level Conditions
3. **Normative References**
4. **Terms and Definitions**
5. **Symbols**
6. **Additional Information**
   - 6.1 Changes to Adopted
   - 6.2 Architectural Alignments
   - 6.3 On the Run-Time Set
   - 6.4 The UML Metamodel
   - 6.5 How to Read this Spec
   - 6.6 Diagrams
6. **Part I - Structure**
7. **Classes**
8. **Components**
9. **Composite Structures**
10. **Deployments**
11. **6. Additional Information**
12. **6.1 Changes to Adopted**
13. **6.2 Architectural Alignments**
14. **6.3 On the Run-Time Set**
15. **6.4 The UML Metamodel**
16. **6.5 How to Read this Spec**
17. **6.6 Diagrams**

#### 7. Classes

#### 7.1 Overview

#### 7.2 Abstract Syntax

#### 7.3 Class Descriptions

- 7.3.1 Abstraction (from Kernel)
- 7.3.2 Aggregation (from Kernel)
- 7.3.3 Association (from Kernel)
- 7.3.4 AssociationEnd (from Kernel)
- 7.3.5 AssociationEndKind (from Kernel)
- 7.3.6 Attribute (from Kernel)
- 7.3.7 BehavioredClass (from Kernel)
- 7.3.8 Classifier (from Kernel)
- 7.3.9 ClassifierLibrary (from Kernel)
- 7.3.10 Class (from Kernel)
- 7.3.11 Constraint (from Kernel)
- 7.3.12 Construct (from Kernel)
- 7.3.13 Dependency (from Kernel)
- 7.3.14 Element (from Kernel)
- 7.3.15 ElementImport (from Kernel)
- 7.3.16 Feature (from Kernel)
- 7.3.17 FeatureExpression (from Kernel)
- 7.3.18 FeatureGeneralization (from Kernel)
- 7.3.19 FeatureInheritance (from Kernel)
- 7.3.20 FeatureOperation (from Kernel)
- 7.3.21 FeatureReference (from Kernel)
- 7.3.22 FeatureSpecification (from Kernel)
- 7.3.23 FeatureValue (from Kernel)
- 7.3.24 Interface (from Interfaces)
- 7.3.25 ValueSpecification (from Kernel)
- 7.3.26 ValueSpecificationKind (from Kernel)
- 7.3.27 LiteralInteger (from Kernel)
- 7.3.28 LiteralString (from Kernel)
- 7.3.29 LiteralURI (from Kernel)
- 7.3.30 PackageableElement (from Kernel)
- 7.3.31 Package (from Kernel)
- 7.3.32 PackageContent (from Kernel)
- 7.3.33 Namespace (from Kernel)
- 7.3.34 NamespacePackage (from Kernel)
- 7.3.35 Parameter (from Kernel)
- 7.3.36 ParameterDirectionKind (from Kernel)
- 7.3.37 ParameterSimpleType (from Kernel)
- 7.3.38 ParameterizedType (from Kernel)
- 7.3.39 ParameterizedTypeBase (from Kernel)
- 7.3.40 ParameterizedTypeKind (from Kernel)
- 7.3.41 ParameterizedTypeParameter (from Kernel)
- 7.3.42 ParameterizedTypeParameterKind (from Kernel)
- 7.3.43 PrimitiveType (from Kernel)
- 7.3.44 PrimitiveTypeFactory (from Kernel)
- 7.3.45 PrimitiveTypeKind (from Kernel)
- 7.3.46 PrimitiveTypeSource (from Kernel)
- 7.3.47 Relationship (from Kernel)
- 7.3.48 Role (from Kernel)
- 7.3.49 StructuralFeature (from Kernel)
- 7.3.50 Substitution (from Kernel)
- 7.3.51 Type (from Kernel)
- 7.3.52 TypeParameter (from Kernel)
- 7.3.53 Usage (from Kernel)
- 7.3.54 VisibilitySpecification (from Kernel)
- 7.3.55 VisibilityKind (from Kernel)
- 7.3.56 VisibilityKindValue (from Kernel)

#### 7.4 Diagrams

#### 8. Components

8.1 **Overview**
8.2 **Abstract syntax**
8.3 **Class Descriptions**
8.4 **Diagrams**

#### 9. Composite Structures

9.1 **Overview**
9.2 **Abstract syntax**
9.3 **Class Descriptions**
9.4 **Diagrams**

#### 10. Deployments

### 2.4 Compliance Level Contents

- Compliance Level 1
- Compliance Level 2
- Compliance Level 3
- Compliance Level 4
- Compliance Level 5

### Reading the Standard Cont’d

#### Figure 7.29 - Class notation: attributes and operations grouped according to visibility

### 7.3.4 Classifier (from Kernel, Dependencies, PowerTypes)

A classifier is a generalization of instance; it describes a set of instances that have features in common.

#### Generalizations

- "Superclass (from Kernel)" on page 57
- "Role (from Kernel)" on page 130

#### Description

A classifier is a metaclass whose members can include features. Classifier is an abstract metaclass.

A classifier is a type and can use generalizations, finally making possible to define generalization relationships in other classifiers. A classifier can specify a generalization hierarchy by referencing its general classifiers.

A classifier is a nonterminal element, meaning that it is possible to model nested classifiers.

#### Attributes

- "Architectural Behavior" on page 152
- "Role (from Kernel)" on page 130

#### Associations

- "Feature (from Kernel)" on page 191
- "Parameter (from Kernel)" on page 183

#### Figures

- "Brokerable Element (from Kernel)" on page 174
- "Feature (from Kernel)" on page 171
- "Parameter (from Kernel)" on page 168
A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to attributes.

**Description**

**Generalizations**

- A classifier is a namespace whose members can include features. A classifier is an abstract meta-class.

**Attributes**

- **Visibility**: Boolean = true
- **Public**: If the classifier is intended to be used by other classifiers (e.g., as the target of generalization relationships). Default value is false.
- **RedefinedClassifier**: Classifier[*]

**Methods**

- **Classifier::feature**: Set(Feature)
- **Classifier::parent**: Set(Classifier)

**UML Superstructure Specification, v2.1.2**
A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers. A classifier can specify a generalization hierarchy by referencing its general classifiers.

### Generalizations

- **generalization**: Generalization[*]
  - Specifies the general classifiers for this classifier. This is derived.

### Associations

- **feature**: Feature[*]
  - Proteced
  - Displays
- **redefinedClassifier**: Classifier[*]
  - Specifies each feature defined in the classifier. Subsets
- **parent**: Classifier
  - Returns all the parents of this classifier.

### Semantics

- **conformsTo**
  - The query conformsTo() gives true for a classifier that defines a type that conforms to another. This is used, for example, to determine which types are compatible with a specific type of model element.

### Examples

- **Class A**
  - **name**: String
  - **size**: Integer
  - **area**: Derived attribute with type Integer. It is marked as read-only.

---

### Package Power Types

Power types are special types of classifiers that define a set of related classifiers. They are useful for defining a hierarchy of models that share common characteristics. Power types are defined by a set of classifiers that are all derived from the same classifier, and they can be used to restrict the types of classifiers that can be used in a model.

- **hasVisibilityOf**
  - The hasVisibilityOf() query is used to determine the visibility of a classifier. It returns true if the visibility of a classifier is not private.

---

### Notation

- **id**
  - The id property is used to uniquely identify a classifier.

---

**Figure 7.31 - Association-like notation for attribute**

The notation used for attributes in classifiers is similar to that used in association diagrams. Attributes are represented as lines connecting classifiers to other classifiers or model elements. The direction of the line indicates whether the attribute is owned by one classifier or whether it is shared between multiple classifiers.

- **value**
  - The value property is used to store the value of an attribute.

---

### Reading list

- **UML Superstructure Specification, v2.1.2**
  - This reference provides a comprehensive overview of the UML standard, including the syntax and semantics of classifiers and other model elements.
• isAbstract: Boolean
A classifier is a namespace whose members can include features. Classifier is an abstract meta-class.

Other classifiers. A classifier can specify a generalization hierarchy by referencing its general classifiers.

A classifier is a redefinable element, meaning that it is possible to redefine nested classifiers.

Generalizations

Classifiers in a one-place to defined notions (components) classifier contains the name of all classifiers.

The classifier is an abstract class (from Kernel).

Notation
Classifiers in the model.

Generalizations
Any component superclassifier to its immediate classifiers.

Associations
Any classifier is a collection of classifiers.

Constraints
Derived.

Stereotypes

Style Guidelines
- Arrows and text.
- Center it.
- For those with no text.
- Left and right.
- Begin and end.

Presentation Options

The dashed line connecting the node to the annotated element(s) may be suppressed if it is clear from the context, or not important in this diagnosis.

UML Superstructure Specification, v2.1.2

7.3.8 Comment (from Kernel)
A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain annotations that are useful in a model.

A comment can be owned by any element.

Attributes
- multiple/empty: String
  Specifies a string that is in the comment.

Associations
- unnamedElement: Element
  References the Element(s) being commented.

Constraints
No additional constraints.

Semantics
A Comment adds no semantics to the annotated element, but may represent information useful to the reader of the model.

Notation
A Comment is shown as a rectangle with the upper right corner bent (this is also known as a “note symbol”). The rectangle contains the body of the Comment. The connection of such an annotated element is drawn by a separate dashed line.

Presentation Options
The dashed line connecting the node to the annotated element(s) may be suppressed if it is clear from the context, or not important in this diagnosis.

UML Superstructure Specification, v2.1.2

55

Package PowerTypes

For example, a Bank Account Type classifier could have a prototype association with a GeneralizationSet. This GeneralizationSet could then associate with two Generalizations where the class, i.e., prototype Classifier: Bank Account has two specific subtypes, i.e., (Classifier: Checking Account and Savings Account). Checking Account and Savings Account are both instances of Bank Account Type, as well as subtypes of Bank Account. (For more explanation and examples, see Examples in the GeneralizationSet sub-clause, below.)

7.3.9 Package PowerTypes

A classifier is a classification of instances according to their features. A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is a Classifier with a set of generalizations that a) have a common specific Classifier, and b) represent a collection of subsets.

Classifiers in a one-place to defined notions (components) classifier contains the name of all classifiers.

The classifier is an abstract class (from Kernel).

Notation
Classifiers in the model.

Generalizations
Any component superclassifier to its immediate classifiers.

Associations
Any classifier is a collection of classifiers.

Constraints
Derived.

Stereotypes

Style Guidelines
- Arrows and text.
- Center it.
- For those with no text.
- Left and right.
- Begin and end.

Presentation Options

The dashed line connecting the node to the annotated element(s) may be suppressed if it is clear from the context, or not important in this diagnosis.

UML Superstructure Specification, v2.1.2

42/57

Meta Object Facility (MOF)
Open Questions...

- Now you’ve been “tricked” again. Twice.
  - We didn’t tell what the **modelling language** for meta-modelling is.
  - We didn’t tell what the **is-instance-of** relation of this language is.
- **Idea:** have a **minimal object-oriented core** comprising the notions of **class**, **association**, **inheritance**, etc. with “self-explaining” semantics.

- This is **Meta Object Facility** (MOF), which (more or less) coincides with UML Infrastructure [OMG, 2007a].

- So: things on meta level
  - M0 are object diagrams/system states
  - M1 are **words of the language UML**
  - M2 are **words of the language MOF**
  - M3 are **words of the language** . . .

**MOF Semantics**

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn’t have a notion of Signal, our signature has.)
**MOF Semantics**

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn’t have a notion of Signal, our signature has.)

- Other approach:
  - Define a **generic, graph based** “is-instance-of” relation.
  - Object diagrams (that are graphs) then are the system states — not only graphical representations of system states.

- If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.
MOF Semantics

- One approach:
  - Treat it with our signature-based theory
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn’t have a notion of Signal, our signature has.)

- Other approach:
  - Define a generic, graph based “is-instance-of” relation.
  - Object diagrams (that are graphs) then are the system states — not only graphical representations of system states.
  - If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.
  - Most interesting: also do generic definition of behaviour within a closed modelling setting, but this is clearly still research, e.g. [?]
Benefits: Overview

- We’ll (superficially) look at three aspects:
  - Benefits for **Modelling Tools**.
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.

Benefits for Modelling Tools

- The meta-model $M_U$ of UML immediately provides a data-structure representation for the abstract syntax ($\sim$ for our signatures).

  If we have code generation for UML models, e.g. into Java, then we can immediately represent UML models in memory for Java.

  (Because each MOF model is in particular a UML model.)

- There exist tools and libraries called **MOF-repositories**, which can generically represent instances of MOF instances (in particular UML models).

  And which can often generate specific code to manipulate instances of MOF instances in terms of the MOF instance.
And not only in memory, if we can represent MOF instances in files, we obtain a canonical representation of UML models in files, e.g., in XML.

→ XML Metadata Interchange (XMI)

**Note:** A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) → OMG Diagram Interchange.
Benefits for Modelling Tools Cont’d

• And not only in memory, if we can represent MOF instances in files, we obtain a canonical representation of UML models in files, e.g. in XML.
  → XML Metadata Interchange (XMI)

• Note: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) → OMG Diagram Interchange.

• Note: There are slight ambiguities in the XMI standard.
  And different tools by different vendors often seem to lie at opposite ends on the scale of interpretation. Which is surely a coincidence.
  In some cases, it’s possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given.
  Plus XMI compatibility doesn’t necessarily refer to Diagram Interchange.

• To re-iterate: this is generic for all MOF-based modelling languages such as UML, CWM, etc.
  And also for Domain Specific Languages which don’t even exit yet.
Benefits for Language Design

• Recall: we said that code-generators are possible “readers” of stereotypes.

• For example, (heavily simplifying) we could
  • introduce the stereotypes Button, Toolbar, ...
  • for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  • instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. corresponding to the stereotype.

One mechanism to define DSLs (based on UML, and “within” UML): Profiles.

Et voilà: we can model Gtk-GUIs and generate code for them.
Benefits for Language Design

- Recall: we said that code-generators are possible “readers” of stereotypes.
- For example, (heavily simplifying) we could
  - introduce the stereotypes Button, Toolbar, ...
  - for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  - instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. corresponding to the stereotype.

Et voilà: we can model Gtk-GUIs and generate code for them.
- Another view:
  - UML with these stereotypes is a new modelling language: Gtk-UML.
  - Which lives on the same meta-level as UML (M2).
  - It’s a Domain Specific Modelling Language (DSL).

One mechanism to define DSLs (based on UML, and “within” UML): Profiles.

Benefits for Language Design Cont’d

- For each DSL defined by a Profile, we immediately have
  - in memory representations,
  - modelling tools,
  - file representations.

- **Note**: here, the semantics of the stereotypes (and thus the language of Gtk-UML) lies in the code-generator.

  That’s the first “reader” that understands these special stereotypes.
  (And that’s what’s meant in the standard when they’re talking about giving stereotypes semantics).

- One can also impose additional well-formedness rules, for instance that certain components shall all implement a certain interface (and thus have certain methods available). (Cf. [Stahl and Völter, 2005].)
Benefits for Language Design Cont’d

- One step further:
  - Nobody hinders us to obtain a model of UML (written in MOF),
  - throw out parts unnecessary for our purposes,
  - add (= integrate into the existing hierarchy) more adequate new constructs, for instance, contracts or something more close to hardware as interrupt or sensor or driver,
  - and maybe also stereotypes.

→ a new language standing next to UML, CWM, etc.

- Drawback: the resulting language is not necessarily UML any more, so we can’t use proven UML modelling tools.

- But we can use all tools for MOF (or MOF-like things). For instance, Eclipse EMF/GMF/GEF.

Benefits for Model (to Model) Transformation

- There are manifold applications for model-to-model transformations:
  - For instance, tool support for re-factorings, like moving common attributes upwards the inheritance hierarchy.

This can now be defined as graph-rewriting rules on the level of MOF.
The graph to be rewritten is the UML model.
Benefits for Model (to Model) Transformation

- There are manifold applications for model-to-model transformations:
  - For instance, tool support for re-factorings, like moving common attributes upwards the inheritance hierarchy. This can now be defined as graph-rewriting rules on the level of MOF. The graph to be rewritten is the UML model.
  - Similarly, one could transform a Gtk-UML model into a UML model, where the inheritance from classes like Gtk::Button is made explicit. The transformation would add this class Gtk::Button and the inheritance relation and remove the stereotype.

- Similarly, one could have a GUI-UML model transformed into a Gtk-UML model, or a Qt-UML model. The former a PIM (Platform Independent Model), the latter a PSM (Platform Specific Model) — cf. MDA.
Special Case: Code Generation

- Recall that we said that, e.g. Java code, can also be seen as a model. So code-generation is a special case of model-to-model transformation; only the destination looks quite different.

- Note: Code generation needn’t be as expensive as buying a modelling tool with full fledged code generation.

  - If we have the UML model (or the DSL model) given as an XML file, code generation can be as simple as an XSLT script.

    “Can be” in the sense of

    “There may be situation where a graphical and abstract representation of something is desired which has a clear and direct mapping to some textual representation.”

In general, code generation can (in colloquial terms) become arbitrarily difficult.
Example: Model and XMI

<?xml version='1.0' encoding='UTF-8' ?>
<XMI xmi.version='1.2' xmlns:UML='org.omg.xmi.namespace.UML' timestamp='Mon Feb 02 18:23:12 CET 2009'>
  <XMI.content>
    <UML:Model xmi.id='...'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id='...' name='SensorA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='pt100'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id='...' name='ControllerA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='65C02'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id='...' name='UsbA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='NET2270'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
      </UML:Namespace.ownedElement>
      <UML:Association xmi.id='...' name='in'>...</UML:Association>
      <UML:Association xmi.id='...' name='out'>...</UML:Association>
    </UML:Model>
  </XMI.content>
</XMI>

References
References


