Software Design, Modelling and Analysis in UML

Lecture 22: Meta-Modelling

2017-02-07

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Content

Inheritance

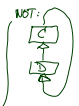
Abstract Syntax

A **signature with inheritance** is a tuple

$$\mathscr{I} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}, F, mth, \lhd)$$

where

- $(\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a signature with signals and behavioural features (F/mth are methods, analogous to V/atr attributes), and
- $\lhd \subseteq (\mathscr{C} \times \mathscr{C}) \cup (\mathscr{E} \times \mathscr{E})$ is an **acyclic generalisation** relation, i.e. $C \lhd^+ C$ for **no** $C \in \mathscr{C}$.



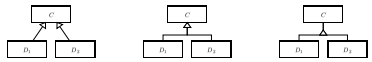
In the following (for simplicity), we assume that all attribute (method) names are of the form $C{::}v$ and $C{::}f$ for some $C \in \mathscr{C} \cup \mathscr{E}$ (**"fully qualified names"**).

Read $C \lhd D$ as...

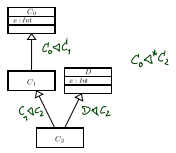
- D **inherits** from C,
- C is a **generalisation** of D,
- D is a **specialisation** of C,
- C is a **super-class** of D,
- D is a **sub-class** of C,
- ...

Inheritance: Concrete Syntax

Common graphical representations (of $\lhd = \{(C, D_1), (C, D_2)\}$):



Mapping Concrete to Abstract Syntax by Example:



Note: we can have **multiple inheritance**.

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one **expects** from **sub-types**, which is closely related to inheritance in object-oriented approaches:

The principle of **type substitutability**:
Liskov Substitution Principle (LSP) Liskov (1988); Liskov and Wing (1994).

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one **expects** from **sub-types**, which is closely related to inheritance in object-oriented approaches:

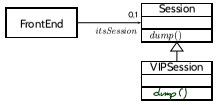
The principle of **type substitutability**:
Liskov Substitution Principle (LSP) Liskov (1988); Liskov and Wing (1994):

> "If for each object o_S of type S
> there is an object o_T of type T
> such that for all programs P defined in terms of T
> **the behavior of** P **is unchanged** when o_S is substituted for o_T
> 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**."

Static Sub-Typing



In FrontEnd's
state machine:

OK, if $T_1 \lhd T_2$

Domain Inclusion vs. Uplink Semantics

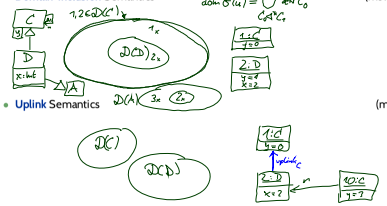
System States with Inheritance

Wanted: a formal representation of "if $C \lhd^* D$ then D 'is a' C", that is,

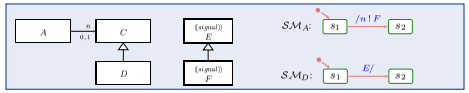
(i) D has the same attributes and behavioural features as C, and
(ii) D objects (identities) can replace C objects.

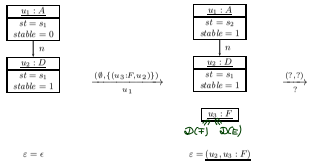
Two approaches to semantics:

- **Domain-inclusion** Semantics (more **theoretical**)
- **Uplink** Semantics (more **technical**)

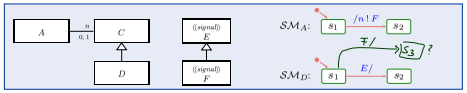


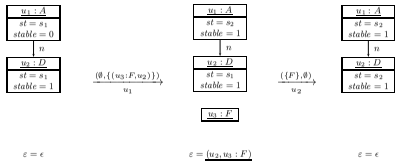
Inheritance and State-Machines: Example





Inheritance and State-Machines: Example





(ii) Dispatch

$$(\sigma, \varepsilon) \xrightarrow{(cons, Snd)}_{u} (\sigma', \varepsilon')$$

if

- $u \in \mathrm{dom}(\sigma) \cap \mathcal{P}(C) \wedge \exists u_E \in \mathcal{P}(E) : u_E \in ready(\varepsilon, u)$
- u is stable and in state machine state s, i.e. $\sigma(u)(stable) = 1$ and $\sigma(u)(st) = s$,
- a transition is **enabled**, i.e.

$$\exists\, (s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C) : F = E \wedge \Gamma[\![expr]\!](\tilde{\sigma}, u) = 1$$

where $\tilde{\sigma} = \sigma[u.params_E \mapsto u_E]$.

and

- (σ', ε') results from applying t_{act} to (σ, ε) and removing u_E from the ether, i.e.

$$(\sigma'', \varepsilon') \in t_{act}[u](\tilde{\sigma}, \varepsilon \ominus u_E),$$
$$\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset)]_{\mathcal{P}(\mathscr{C}) \setminus \{u_E\}}$$

where b **depends** (see (i))

- Consumption of u_E and the side effects of the action are observed, i.e.

$$cons = \{u_E\}, \quad Snd = Obs_{t_{act}}[u](\tilde{\sigma}, \varepsilon \ominus u_E).$$

Recall: Subtyping

There is a classical description of what one **expects** from **sub-types**, which is closely related to inheritance in object-oriented approaches:

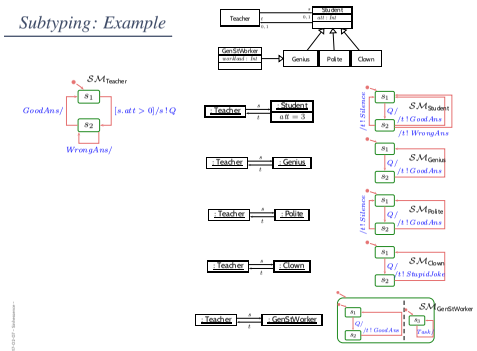
The principle of **type substitutability**:
Liskov Substitution Principle (LSP) Liskov (1988); Liskov and Wing (1994).

> "If for each object o_S of type S
> there is an object o_T of type T
> such that for all programs P defined in terms of T
> **the behavior of P is unchanged** when o_S is substituted for o_T
> 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**."

Subtyping: Example



Meta-Modelling: Idea

Meta-Modelling: Why and What

- **Meta-Modelling** is one major prerequisite for understanding
  - the standard documents OMG (2011a,b), and
  - the MDA ideas of the OMG.

- The idea is somewhat **simple**:
  - if a **modelling language** is about modelling **things**,
  - and if UML models are **things**,
  - then why not **describe** (or: **model**) the set of all UML models **using a modelling language**?

Meta-Modelling: Example

For example, let's consider a class.

- A **class** has (among others)
  - a **name**,
  - any number of **attributes**,
  - any number of **behavioural features**.

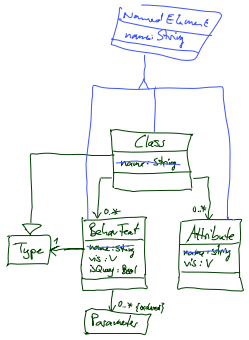
Each of the latter two has

- a **name** and
- a **visibility**.

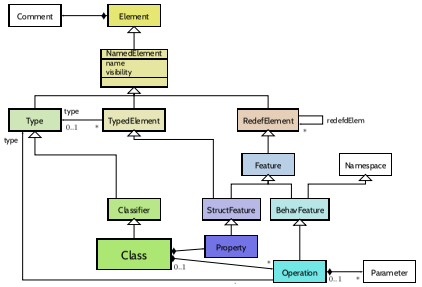
Behavioural features in addition have

- a boolean attribute **isQuery**,
- any number of parameters,
- a return type.

Can we model this (in UML, for a start)?



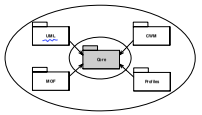
UML Meta-Model: Extract from UML 2.0 Standard



The UML 2.x Standard Revisited

UML Architecture (OMG, 2003, 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.



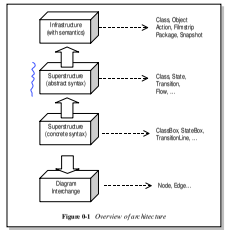


Figure 0-1 *Overview of architecture*

UML Superstructure Packages (OMG, 2007a, 15)

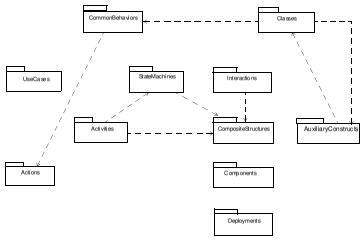
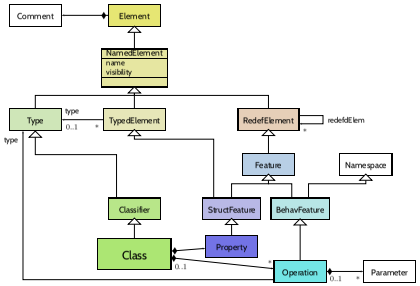


Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

Claim: Extract from UML 2.0 Standard



Classes (OMG, 2007b, 32)

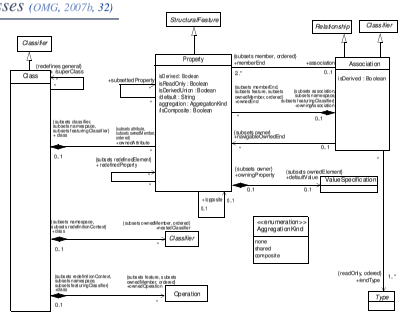


Figure 7.12 - Classes diagram of the Kernel package

Operations (OMG, 2007b, 31)

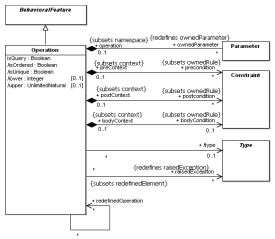


Figure 7.11 - Operations diagram of the Kernel package

Operations (OMG, 2007b, 30)

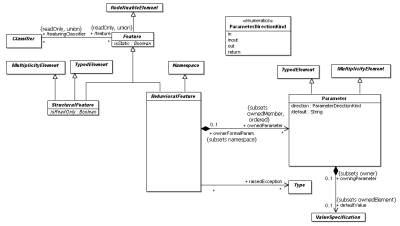
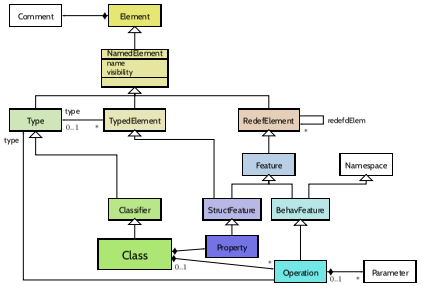


Figure 7.10 - Features diagram of the Kernel package

Claim: Extract from UML 2.0 Standard



Classifiers (OMG, 2007b, 29)

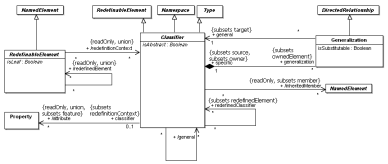


Figure 7.9 - Classifiers diagram of the Kernel package

Namespaces (OMG, 2007b, 26)

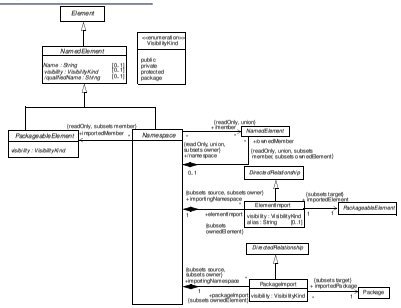


Figure 7.4 - Namespaces diagram of the Kernel package

Root Diagram (OMG, 2007b, 25)

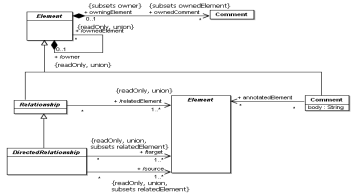


Figure 7.3 - Root diagram of the Kernel package

Table of Contents

Table of Contents

Table of Contents

Figure 7.39 - Classifier attributes and operations grouped according to visibility

7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)

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

Generalizations

- "Namespace (from Kernel)" on page 99
- "RedefinableElement (from Kernel)" on page 130
- "Type (from Kernel)" on page 136

Description

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

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 generic classifier.

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

Attributes

- isAbstract: Boolean
  If true, the Classifier does not provide a complete declaration and can typically not be instantiated. An abstract classifier is intended to be used by other classifiers (e.g., as the target of general and generalization relationships). Default value is false.

Associations

- ownedUseCase: Property [*]
  Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier. Subsets Classifier::ownedAttribute.
- /feature: Feature [*]
  Specifies each feature defined in the classifier. Subsets Namespace::member. This is a derived union.
- /general: Classifier [*]
  Specifies the immediate Classifiers for this Classifier. This is derived.

Figure 7.39 - Cl

7.3.8 Clas

A classifier is a

General

- "Namesp
- "RedEle
- "Type (f

Descripti

A classifier is a
A classifier is a
other classifier

Attribute

- isAbstra
  If true,
  Classifi

Associa

- ownedU
  Refers
  Classifi
- /feature
  Specifi
- /general
  Specifi

- generalization: Generalization[*]
  Specifies the Generalization relationships for this Classifier. These Generalizations navigate to more general classifiers in the generalization hierarchy. Subsets Element::ownedElement.
- /inheritedMember: NamedElement[*]
  Specifies all elements inherited by this classifier from the general classifiers. Subsets Namespace::member. This is derived.
- redefinedClassifier: Classifier [*]
  References the Classifiers that are redefined by this Classifier. Subsets RedefinableElement::redefinedElement.

Package Dependencies

- substitution : Substitution
  Designates the Substitutions that are owned by this Classifier. Subsets Element::ownedElement and RedefinableElement::substitution.

Package PowerTypes

- powertypeExtent : GeneralizationSet
  Designates the Generalization Set of which the associated Classifier is a power type.

Constraints

[1] The generalizations Classifiers are the classifiers referenced by the generalization relationships.

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

not self.allParents()->includes(self)

[3] A Classifier may only specialize classifiers of valid types.

self.general->forAll(c | self.oclAsType(NamedElement).

[4] The inherited visibility associations is derived by inheriting the inheritable members of the parents.

self.inheritedMember = self.allParents().ownedMember->select(p | p.isInheritableMember(self))

Package PowerTypes

[1] The Classifier that assigns its Generalization set may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that Generalization set. In other words, a power type may not be an instance of itself nor may its instances also be its subclasses.

Additional Operations

[1] The query allMembers() gives all of the features in the namespace of the classifier. In general, following mechanisms such as inheritance, this will be a larger set than features.

Classifier::allMembers(): Set(Feature)

allMembers = member->union(inheritedMember)

[2] The query parents() gives all of the immediate ancestors of a generalized Classifier.

Classifier::parents(): Set(Classifier)

parents = generalization.general

Figure 7.39 - Cl

7.3.8 Clas

A classifier is a

General

- "Namesp
- "RedEle
- "Type (f

Descripti

A classifier is a
A classifier is a
other classifier

Attribute

- isAbstra
  If true,
  Classifi

Associa

- ownedU
  Refers
  Classifi
- /feature
  Specifi
- /general
  Specifi

- generalization: Generalization[*]
  Specifies the Generalization relationships for this Classifier. These Generalizations navigate to more general classifiers in the generalization hierarchy. Subsets Element::ownedElement.
- /inheritedMember: NamedElement[*]
  Specifies all elements inherited by this classifier from the general classifiers. Subsets Namespace::member. This is derived.
- redefinedClassifier: Classifier [*]
  References the Classifiers that are redefined by this Classifier. Subsets RedefinableElement::redefinedElement.

Package Dependencies

- substitution : Substitution
  Designates the Substitutions that are owned by this Classifier. Subsets Element::ownedElement and RedefinableElement::substitution.

Package PowerTypes

- powertypeExtent : GeneralizationSet
  Designates the Generalization Set of which the associated Classifier is a power type.

Constraints

[1] The general classifiers are the classifiers referenced by the generalization relationships.

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

not self.allParents()->includes(self)

[3] A Classifier may only specialize classifiers of valid types.

self.general->forAll(c | self.oclAsType(NamedElement).

[4] The inherited visibility associations is derived by inheriting the inheritable members of the parents.

self.inheritedMember = self.allParents().ownedMember->select(p | p.isInheritableMember(self))

Package PowerTypes

[1] The Classifier that assigns its Generalization set may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that Generalization set. In other words, a power type may not be an instance of itself nor may its instances also be its subclasses.

Additional Operations

[1] The query allMembers() gives all of the features in the namespace of the classifier. In general, following mechanisms such as inheritance, this will be a larger set than features.

Classifier::allMembers(): Set(Feature)

allMembers = member->union(inheritedMember)

[2] The query parents() gives all of the immediate ancestors of a generalized Classifier.

Classifier::parents(): Set(Classifier)

parents = generalization.general

[3] The query allParents() gives all of the direct and indirect ancestors of a generalized Classifier.

Classifier::allParents(): Set(Classifier)

allParents = self.parents()->union(self.parents()->collect(p | p.allParents))

[4] The query inheritableMembers() gives all of the members of a classifier that may be inherited in one of its descendants, subject to whatever visibility restrictions apply.

Classifier::inheritableMembers(c: Classifier): Set(NamedElement)

pre: c.allParents()->includes(self)

inheritableMembers = member->select(m | c.hasVisibilityOf(m))

[5] The query hasVisibilityOf() determines whether a named element is visible in the classifier. By default all are visible. It is only called when the argument is something owned by a parent.

Classifier::hasVisibilityOf(n: NamedElement): Boolean

pre: self.allParents()->includes(n.namespace)

post: if n.visibility = #public

then

result = true

else

result = (n.visibility <> #private)

endif

[6] The query conformsTo() gives true for a classifier that defines a type that conforms to another. This is used, for example, in the specification of signature conformance for operations.

Classifier::conformsTo(other: Classifier): Boolean

conformsTo = (self = other or self.allParents()->includes(other))

[7] The query isTemplate() returns whether this classifier is actually a template. It is true if this classifier or any of its owned members are templates.

Classifier::isTemplate(): Boolean

result = (self.ownedTemplateSignature->notEmpty() or self.ownedMember->exists(m | m.isTemplate()))

[8] The query maySpecializeType() determines whether this classifier may have a generalization relationship to a classifier of the specified type. By default a classifier may specialize classifiers of the same or more general type. It is intended to be redefined by classifiers that have different specialization rules.

Classifier::maySpecializeType(c: Classifier): Boolean;

maySpecializeType = self.oclIsKindOf(c.type)

Semantics

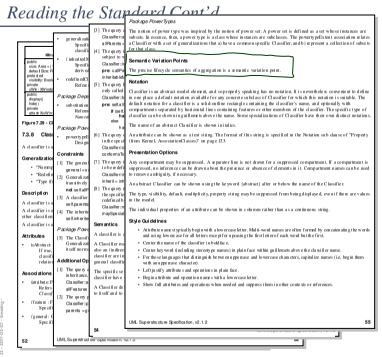
A classifier is a classification of instances according to its features.

A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is also an indirect instance of each of the general Classifiers. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraints applying to instances of the general classifier also apply to instances of the specific classifier.

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a classifier have values corresponding to the classifier's attributes.

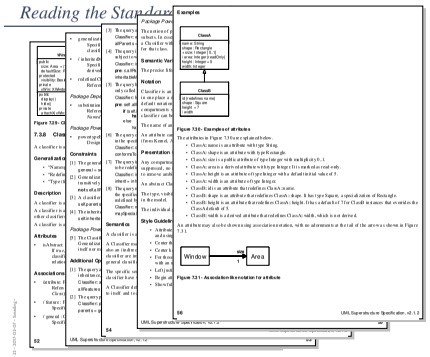
A Classifier defines a type. Type conformance between generalizable Classifiers is defined so that a Classifier conforms to itself and to all of its ancestors in the generalization hierarchy.

Reading the Standard Cont'd



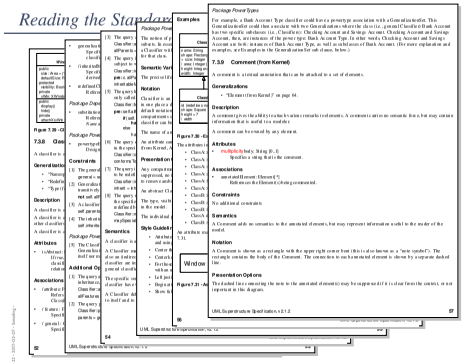
30/41

Reading the Standard



30/41

Reading the Standard



30/41

Meta Object Facility (MOF)

31/41

Open Questions...

- Now you've been "**tricked**"...
  - We didn't tell what the **modelling language** for meta-modelling 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~~

32/41

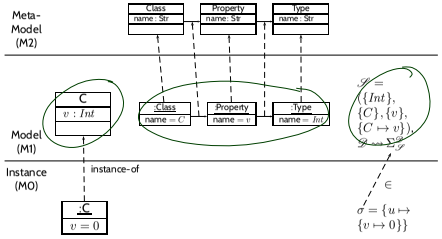
Benefits

- In particular:
  - Benefits for **Modelling Tools**.
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.

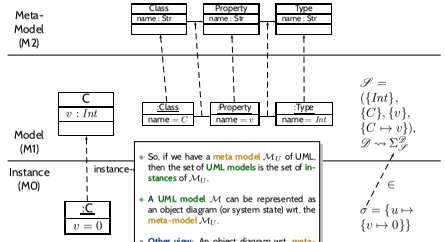
33/41

Meta-Modelling: Principle

Modelling vs. Meta-Modelling



Modelling vs. Meta-Modelling



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.

  Constraint 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.
  >
  > not self . allParents( ) ->> includes(self)" (OMG, 2007b, 53)

- The other way round:

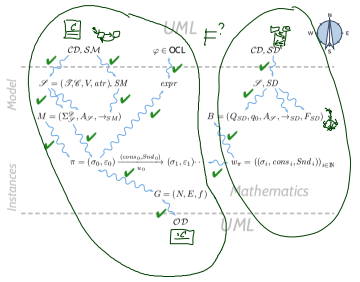
  Given a UML model $\mathcal{M}$, unfold it into an object diagram O_1 wrt. $\mathcal{M}_U$.

  If O_1 is a valid object diagram of $\mathcal{M}_U$ (i.e. satisfies all invariants from $Inv(\mathcal{M}_U)$), then $\mathcal{M}$ 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.

And That's It!

The Map



Content

- Lecture 1: Introduction

Software Design, Modelling and Analysis in UML

Lecture 1: Introduction

2016-10-18

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Content

- Lecture 1: Introduction
- Lecture 2: Semantical Model

Contents & Goals

Last Lecture:
- Introduction: Motivation, Context, Formula

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What is a signature, an object, a system state, etc.?
  - What is the purpose of signature, object, etc. in the course?
  - How do/will we Object System Signatures relate to UML class diagrams?
- Content:
  - Basic Object System Signatures
  - Structures
  - System States

Content

- Lecture 1: Introduction
- Lecture 2: Semantical Model
- Lecture 3: Object Constraint Language (OCL)

Contents & Goals

Last Lecture:
- Basic Object System Signature ($\mathcal{S}$ and Structure ($\mathcal{S}$, System State $\sigma \in \Sigma$))

This Lecture:
- Educational Objectives: Capabilities for these tasks/questions:
  - Please explain this OCL constraint.
  - Please formalise this constraint in OCL.
  - Does this OCL constraint hold in this system state?
  - Give a system state satisfying this constraint!
  - Please use alternative of ~~blackboard~~ in this OCL expression.
  - In what sense is OCL a three-valued logic? For what purpose?
  - How are $(\mathcal{S}/\top)$ and $\Sigma_{\mathcal{S}}$ related?
- Content:
  - OCL Syntax
  - OCL Semantics (over system states)

Content

- Lecture 1: Introduction
- Lecture 2: Semantical Model
- Lecture 3: Object Constraint Language (OCL)
- Lecture 4: OCL Semantics

Contents & Goals

Last Lecture:
- OCL Syntax

This Lecture:
- Educational Objectives: Capabilities for these tasks/questions:
  - Please un-abbreviate all abbreviations in this OCL expression ✓
  - Please explain this OCL constraint.
  - Please formalise this constraint in OCL.
  - Does this OCL constraint hold in this system state?
  - Give a system state satisfying this constraint!
  - In what sense is OCL a three-valued logic? For what purpose?
  - How are $(\mathcal{S}/\top)$ and $\Sigma_{\mathcal{S}}$ related?
- Content:
  - OCL Semantics
  - OCL Consistency and Satisfiability

Content

- Lecture 1: Introduction
- Lecture 2: Semantical Model
- Lecture 3: Object Constraint Language (OCL)
- Lecture 4: OCL Semantics
- Lecture 5: Object Diagrams

Contents & Goals

Last Lecture:
- OCL Semantics

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What does it mean that an OCL expression is satisfiable?
  - When is a set of OCL constraints said to be consistent?
  - What is an object diagram? What are object diagrams good for?
  - When is an object diagram called partial? What are partial ones good for?
  - When is an object diagram an object diagram (wrt. what)?
  - How are system states and object diagrams related?
  - Can you think of an object diagram which violates this OCL constraint?
- Content:
  - OCL consistency, satisfiability
  - Object Diagrams
  - Example: Object Diagrams for Documentation

Content

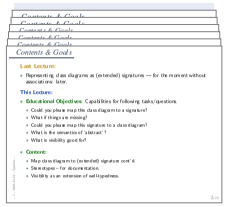
- Lecture 1: Introduction
- Lecture 2: Semantical Model
- Lecture 3: Object Constraint Language (OCL)
- Lecture 4: OCL Semantics
- Lecture 5: Object Diagrams
- Lecture 6: Class Diagrams I

Contents & Goals

Last Lecture:
- Object Diagrams
  - partial vs. complete, for a notion, for documentation, ...

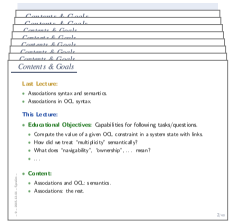
This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What is a class diagram?
  - For what purposes are class diagrams useful?
  - Could you please map this class diagram to a signature?
  - Could you please map this signature to a class diagram?
- Content:
  - Study UML syntax.
  - Prepare (extend) definition of signature.
  - Map class diagram to (extended) signature.
  - Stereotypes.

Content

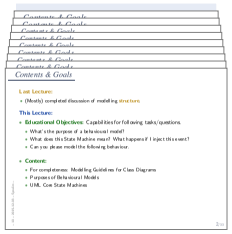


Content

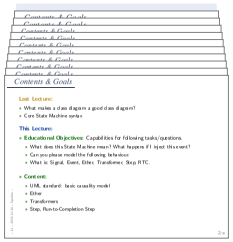
Content



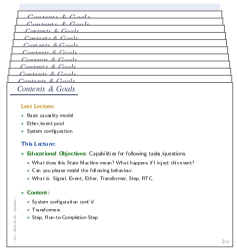
Content



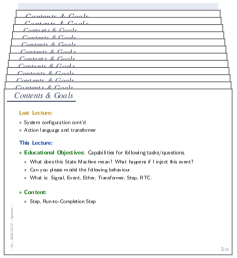
Content



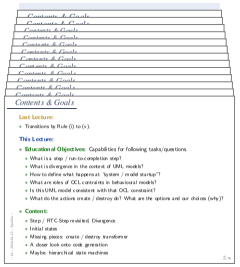
Content



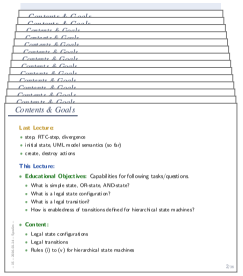
Content



Content

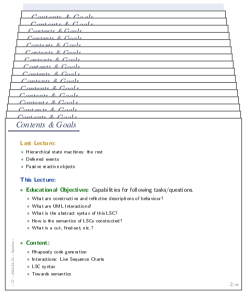


Content

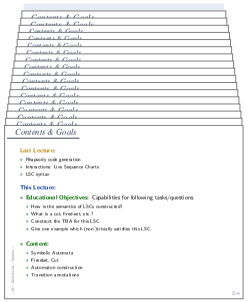


Content

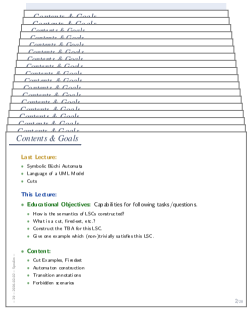
Content



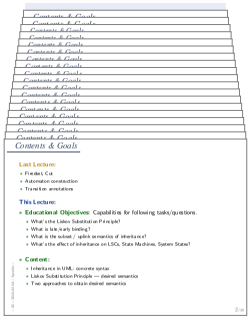
Content



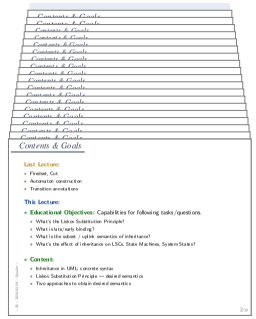
Content



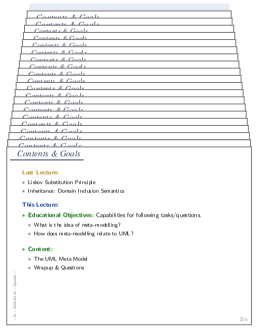
Content



Content



Content



References

References

Fischer, C. and Wehrheim, H. (2000). Behavioural subtyping relations for object-oriented formalisms. In Rus, T., editor, *AMAST*, number 1816 in Lecture Notes in Computer Science. Springer-Verlag.

Liskov, B. (1988). Data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34.

Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841.

OMG (2003). Uml 2.0 proposal of the 2U group, version 0.2, `http://www.2uworks.org/uml2submission`.

OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

OMG (2007b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

OMG (2011a). Unified modeling language: Infrastructure, version 2.4.1. Technical Report formal/2011-08-05.

OMG (2011b). Unified modeling language: Superstructure, version 2.4.1. Technical Report formal/2011-08-06.