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

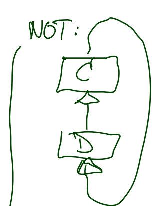
- **Inheritance**
  - **Abstract syntax**
  - **Liskov Substitution Principle**
  - Well-typedness with inheritance
  - Subset-semantics vs. uplink-semantics
- **Meta-Modelling**
  - **Idea**
  - **Experiment**: can we **model classes**?
  - **Revisit** the UML 2.x standard (vs. **experiment**)
  - **Meta Object Facility** (MOF)
  - The **principle illustrated** (once again)
- **And That's It!**
  - **The map** – in hindsight.
  - Educational objectives – **useful questions**.
- **Any open questions?**

Inheritance

Abstract Syntax

A **signature with inheritance** is a tuple

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



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** $C \in \mathscr{C}$.

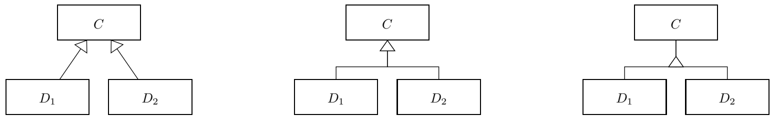
In the following (for simplicity), we assume that all attribute (method) names are of the form $C::v$ and $C::f$ for some $C \in \mathscr{C} \cup \mathscr{E}$ ("**fully qualified names**").

Read $C \lhd D$ as...

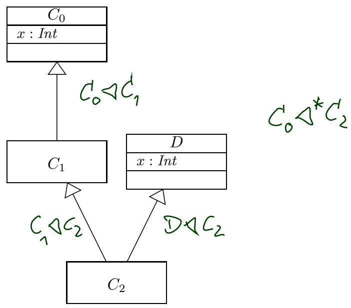
- D **inherits** from C,
- C is a **generalisation** of D,
- D is a **specialisation** of C,
- C is a **super-class** of D,
- D is a **sub-class** of C,
- ...

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

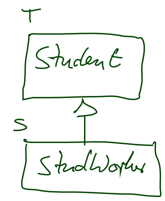
There is a classical description of what one **expects** from **sub-types**, which is closely related to inheritance in object-oriented approaches:

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).

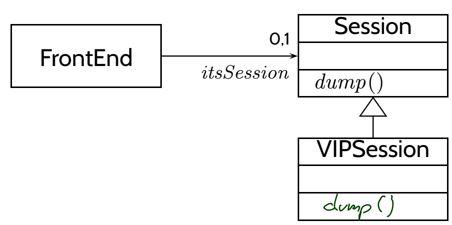


> "If for each object o_S of type S
>
> there is an object o_T of type T
>
> such that for all programs P defined in terms of T
>
> **the behavior of P is unchanged** when o_S is substituted for o_T
>
> then S is a **subtype** of T."

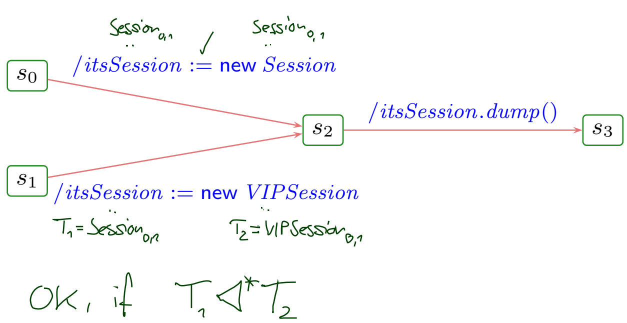
In other words: Fischer and Wehrheim (2000)

> "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:



Domain Inclusion vs. Uplink Semantics

System States with Inheritance

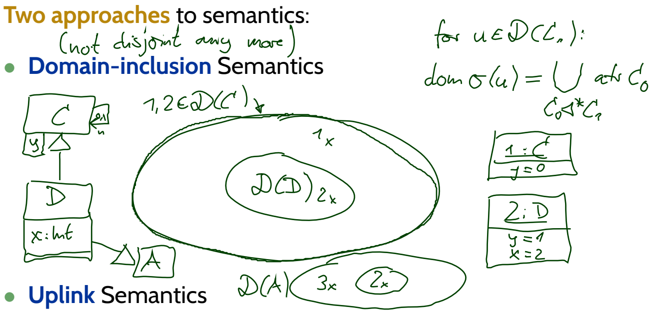
Wanted: a formal representation of "if $C \lhd^* D$ then D '**is a**' C", that is,

(i) D has the same attributes and behavioural features as C, and

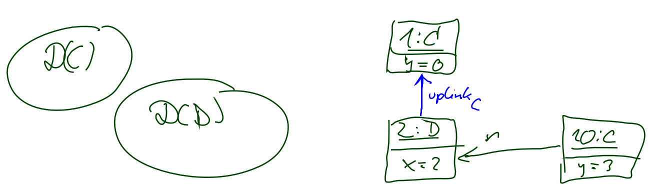
(ii) D objects (identities) can replace C objects.

Two approaches to semantics:

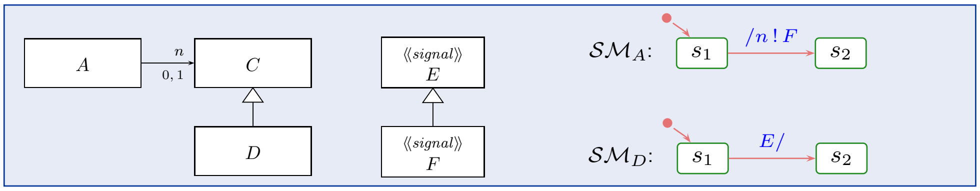
- **Domain-inclusion** Semantics (more **theoretical**)

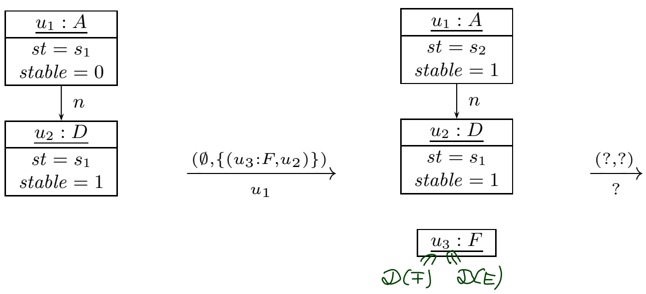


- **Uplink** Semantics (more **technical**)



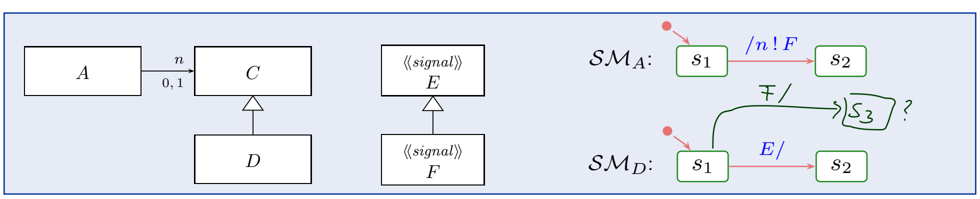
Inheritance and State-Machines: Example

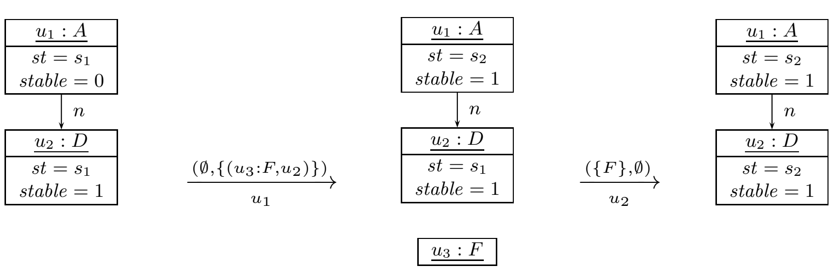




$\varepsilon = \epsilon$ $\qquad$ $\varepsilon = (u_2, u_3 : F)$

Inheritance and State-Machines: Example





$\varepsilon = \epsilon$ $\qquad$ $\varepsilon = (u_2, u_3 : F)$ $\qquad$ $\varepsilon = \epsilon$

(ii) Dispatch

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

if

- $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) \wedge \exists\, u_E \in \mathscr{D}(E) : u_E \in ready(\varepsilon, u)$
- 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.

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

where $\tilde{\sigma} = \sigma[u.params_E \mapsto u_E]$. e.g. $s_1 \xrightarrow{E[params_E.x>0]/} s_2$

and

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

$$(\sigma'', \varepsilon') \in t_{act}[u](\tilde{\sigma}, \varepsilon \ominus u_E),$$

$$\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|_{\mathscr{D}(\mathscr{C}) \setminus \{u_E\}}$$

(remove 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."

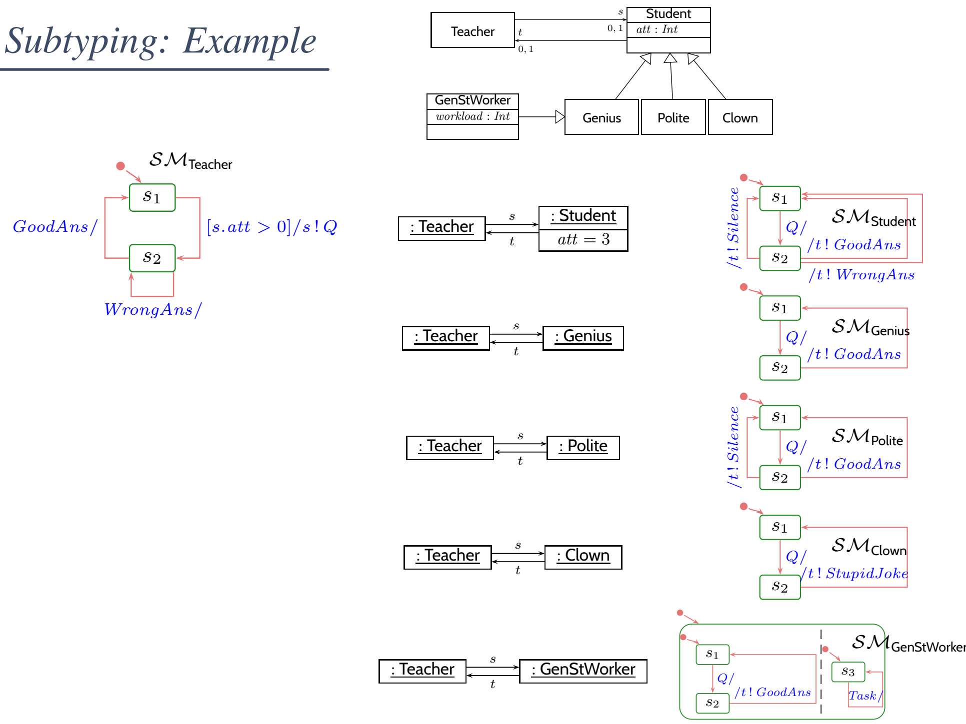
In other words: Fischer and Wehrheim (2000)

"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

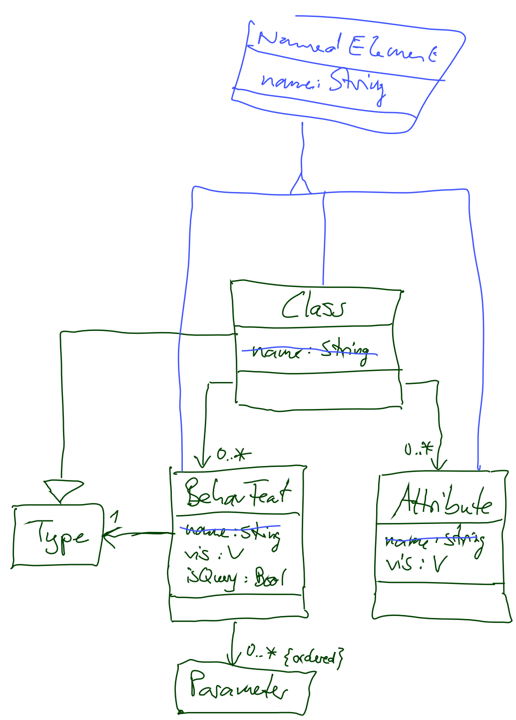
For example, let's consider a class.

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

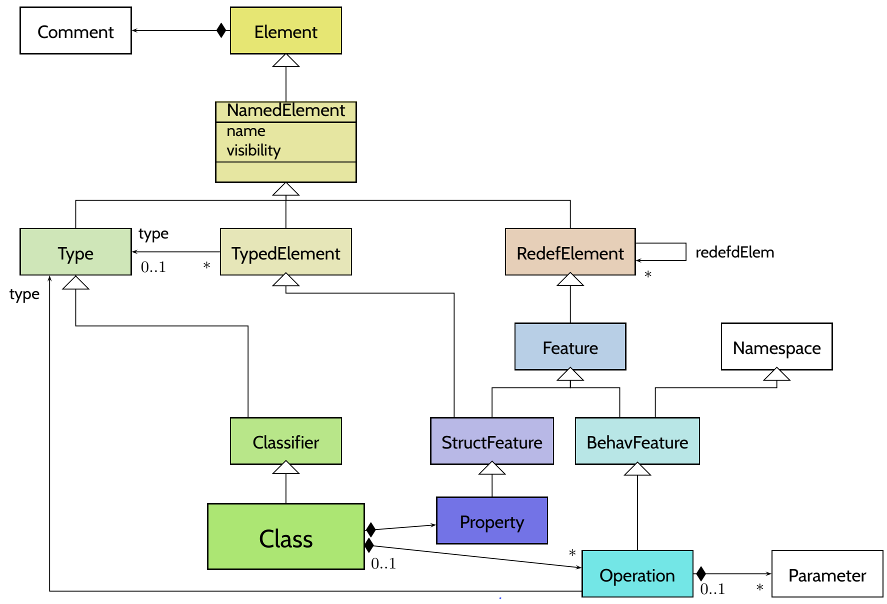
  Each of the latter two has
  - a **name** and
  - a **visibility**.

  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

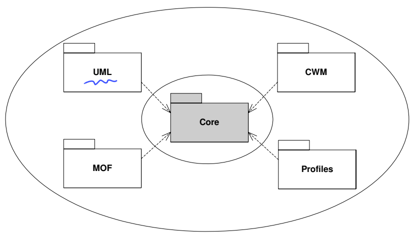


The UML 2.x Standard Revisited

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.



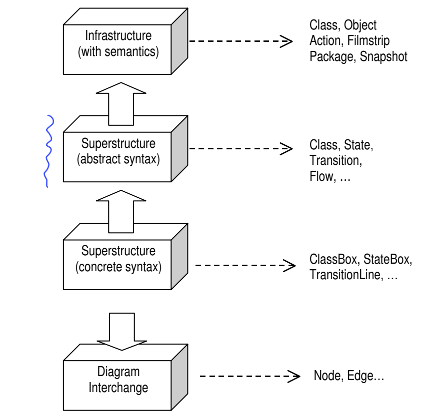


Figure 0-1 *Overview of architecture*

UML Superstructure Packages (OMG, 2007a, 15)

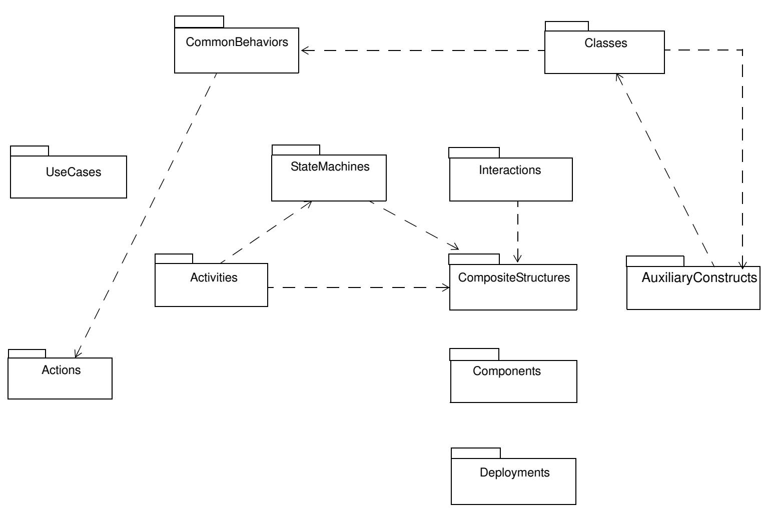
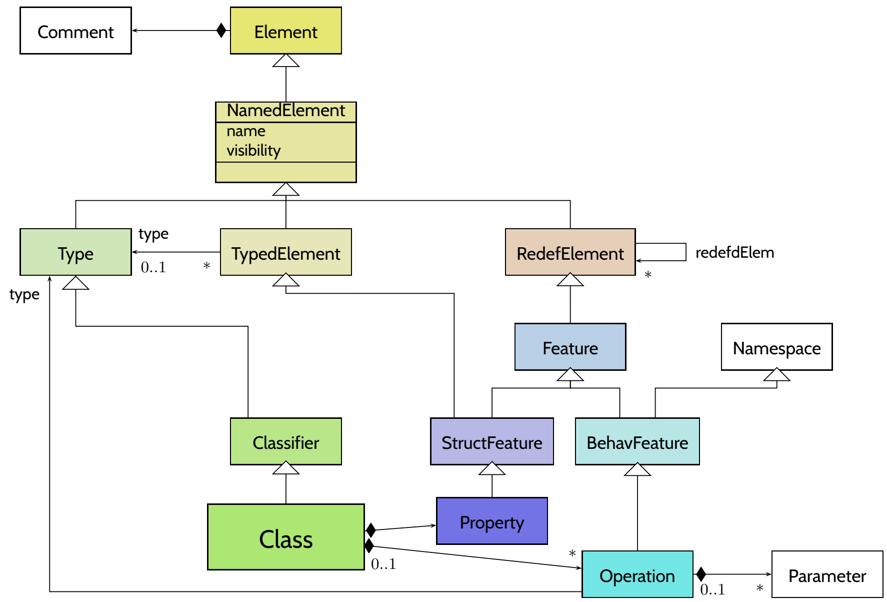


Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

Claim: Extract from UML 2.0 Standard



Classes (OMG, 2007b, 32)

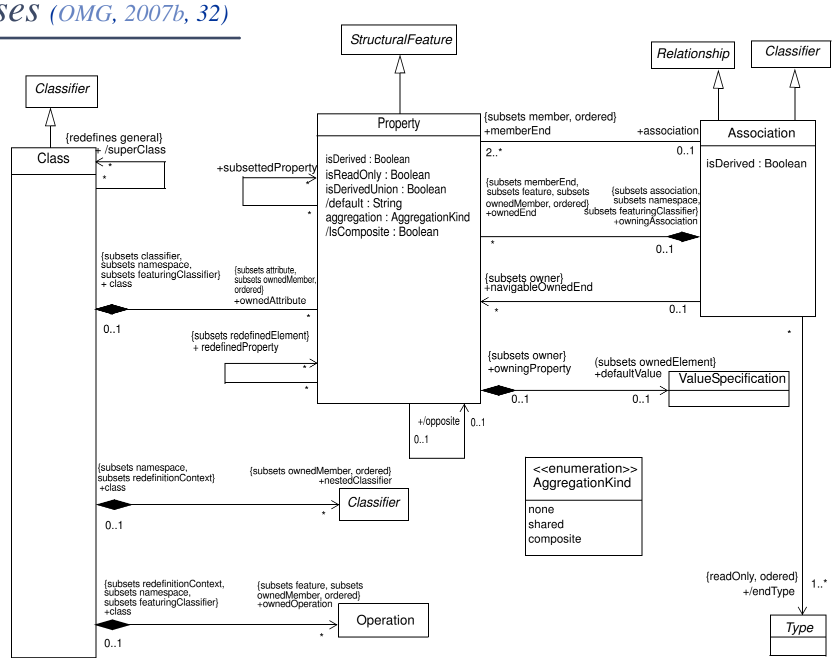


Figure 7.12 - Classes diagram of the Kernel package

Operations (OMG, 2007b, 31)

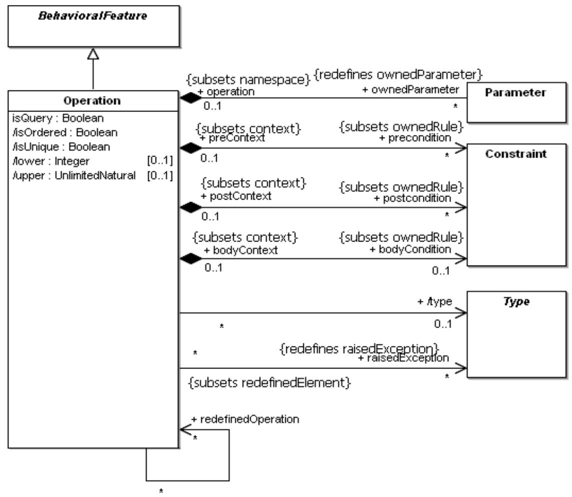


Figure 7.11 - Operations diagram of the Kernel package

Operations (OMG, 2007b, 30)

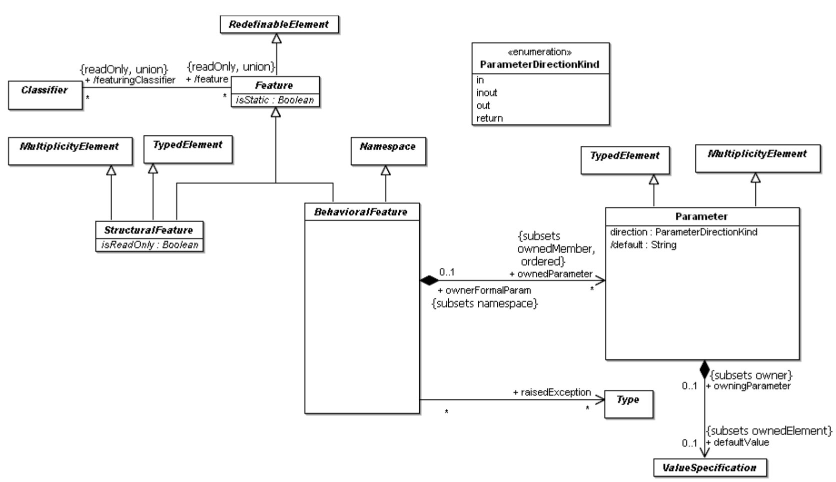
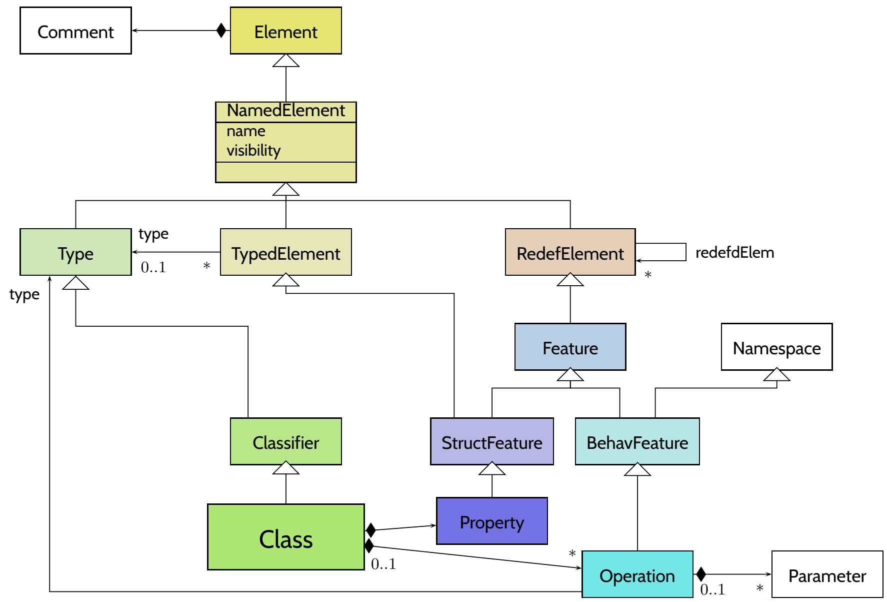


Figure 7.10 - Features diagram of the Kernel package

Claim: Extract from UML 2.0 Standard



Classifiers (OMG, 2007b, 29)

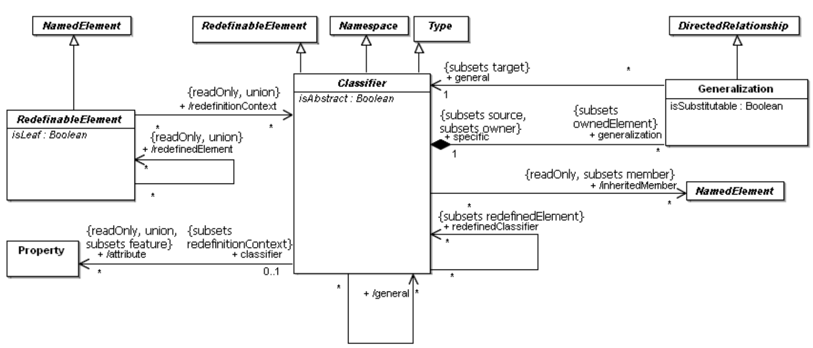


Figure 7.9 - Classifiers diagram of the Kernel package

Namespaces (OMG, 2007b, 26)

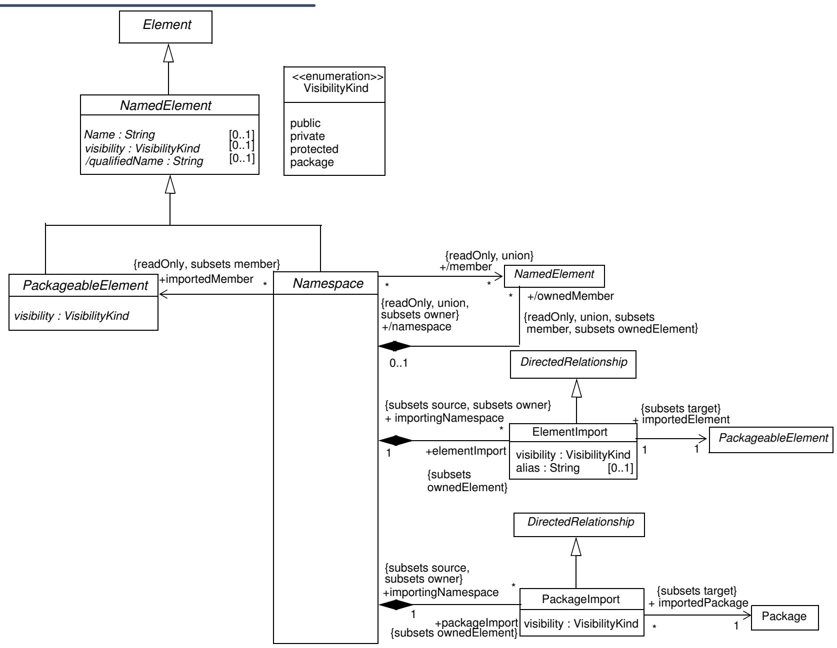


Figure 7.4 - Namespaces diagram of the Kernel package

Root Diagram (OMG, 2007b, 25)

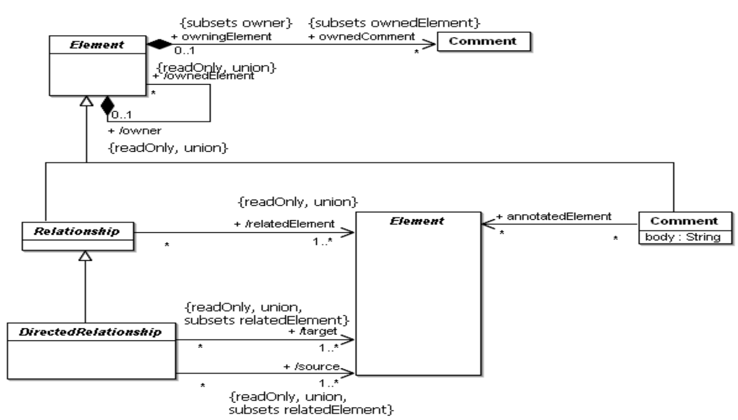


Figure 7.3 - Root diagram of the Kernel package

Reading the Standard

Table of Contents

UML Superstructure Specification, v2.1.2 i

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 Co
3. Normative References
4. Terms and Definitions
5. Symbols . . .
6. Additional Information
   - 6.1 Changes to Adopted
   - 6.2 Architectural Alignme
   - 6.3 On the Run-Time Se
     - 6.3.1 The Basic Premis
     - 6.3.2 The Semantics Ar
     - 6.3.3 The Basic Causal
     - 6.3.4 Semantics Descri
   - 6.4 The UML Metamode
     - 6.4.1 Models and What
     - 6.4.2 Semantic Levels
   - 6.5 How to Read this Sp
     - 6.5.1 Specification form
     - 6.5.2 Diagram format ..
   - 6.6 Acknowledgements

Part I - Structure . .

7. Classes . . .

UML Superstructure Specification, v2.1.2 i

ii UML Superstructure Specification, v2.1.2

Reading the Standard

Table of Contents

Reading the Standard Cont'd

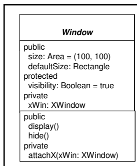


Figure 7.29 - Class notation: attributes and operations grouped according to visibility

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 135

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 general classifiers.

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 metarelationships or generalization relationships). Default value is *false*.

Associations

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

Reading the Standard Cont’d

- 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
  References the substitutions that are owned by this Classifier. Subsets *Element::ownedElement* and *NamedElement::clientDependency*.)

Package PowerTypes

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

Constraints

[1] The general classifiers are the classifiers referenced by the generalization relationships.
general = self.parents()

[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)

[3] A classifier may only specialize classifiers of a valid type.
self.parents()->forAll(c | self.maySpecializeType(c))

[4] The inheritedMember association is derived by inheriting the inheritable members of the parents.
self.inheritedMember->includesAll(self.inherit(self.parents()->collect(p | p.inheritableMembers(self)))

Package PowerTypes

[5] The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that GeneralizationSet. 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 allFeatures() gives all of the features in the namespace of the classifier. In general, through mechanisms such as inheritance, this will be a larger set than feature.
Classifier::allFeatures(): Set(Feature);
allFeatures = member->select(oclIsKindOf(Feature))

[2] The query parents() gives all of the immediate ancestors of a generalized Classifier.
Classifier::parents(): Set(Classifier);
parents = generalization.general

52

UML Superstructure Specification, v2.1.2 53

Reading the Standard Cont’d

[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()->collect(c | c.member)->includes(n)
if (self.inheritedMember->includes(n)) **then**
hasVisibilityOf = (n.visibility <> #private)
else
hasVisibilityOf = **true**

[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 inherit() defines how to inherit a set of elements. Here the operation is defined to inherit them all. It is intended to be redefined in circumstances where inheritance is affected by redefinition.
Classifier::inherit(inhs: Set(NamedElement)): Set(NamedElement);
inherit = inhs

[8] The query maySpecializeType() determines whether this classifier may have a generalization relationship to classifiers of the specified type. By default a classifier may specialize classifiers of the same or a more general type. It is intended to be redefined by classifiers that have different specialization constraints.
Classifier::maySpecializeType(c : Classifier) : Boolean;
maySpecializeType = self.oclIsKindOf(c.oclType)

Semantics

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 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 constraint applying to instances of the general classifier also applies 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.

54 UML Superstructure Specification, v2.1.2

Reading the Standard Cont'd

Package PowerTypes

The notion of power type was inspired by the notion of power set. A power set is defined as a set whose instances are subsets. In essence, then, a power type is a class whose instances are subclasses. The powertypeExtent association relates a Classifier with a set of generalizations that a) have a common specific Classifier, and b) represent a collection of subsets for that class.

Semantic Variation Points

The precise lifecycle semantics of aggregation is a semantic variation point.

Notation

Classifier is an abstract model element, and so properly speaking has no notation. It is nevertheless convenient to define in one place a default notation available for any concrete subclass of Classifier for which this notation is suitable. The default notation for a classifier is a solid-outline rectangle containing the classifier's name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier. The specific type of classifier can be shown in guillemets above the name. Some specializations of Classifier have their own distinct notations.

The name of an abstract Classifier is shown in italics.

An attribute can be shown as a text string. The format of this string is specified in the Notation sub clause of "Property (from Kernel, AssociationClasses)" on page 123.

Presentation Options

Any compartment may be suppressed. A separator line is not drawn for a suppressed compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it. Compartment names can be used to remove ambiguity, if necessary.

An abstract Classifier can be shown using the keyword {abstract} after or below the name of the Classifier.

The type, visibility, default, multiplicity, property string may be suppressed from being displayed, even if there are values in the model.

The individual properties of an attribute can be shown in columns rather than as a continuous string.

Style Guidelines

- Attribute names typically begin with a lowercase letter. Multi-word names are often formed by concatenating the words and using lowercase for all letters except for upcasing the first letter of each word but the first.
- Center the name of the classifier in boldface.
- Center keyword (including stereotype names) in plain face within guillemets above the classifier name.
- For those languages that distinguish between uppercase and lowercase characters, capitalize names (i.e, begin them with an uppercase character).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show full attributes and operations when needed and suppress them in other contexts or references.

UML Superstructure Specification, v2.1.2 55

Reading the Standard Cont'd

Examples

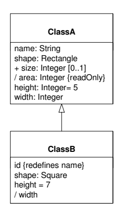


Figure 7.30 - Examples of attributes

The attributes in Figure 7.30 are explained below.

- ClassA::name is an attribute with type String.
- ClassA::shape is an attribute with type Rectangle.
- ClassA::size is a public attribute of type Integer with multiplicity 0..1.
- ClassA::area is a derived attribute with type Integer. It is marked as read-only.
- ClassA::height is an attribute of type Integer with a default initial value of 5.
- ClassA::width is an attribute of type Integer.
- ClassB::id is an attribute that redefines ClassA::name.
- ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle.
- ClassB::height is an attribute that redefines ClassA::height. It has a default of 7 for ClassB instances that overrides the ClassA default of 5.
- ClassB::width is a derived attribute that redefines ClassA::width, which is not derived.

An attribute may also be shown using association notation, with no adornments at the tail of the arrow as shown in Figure 7.31.

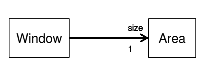
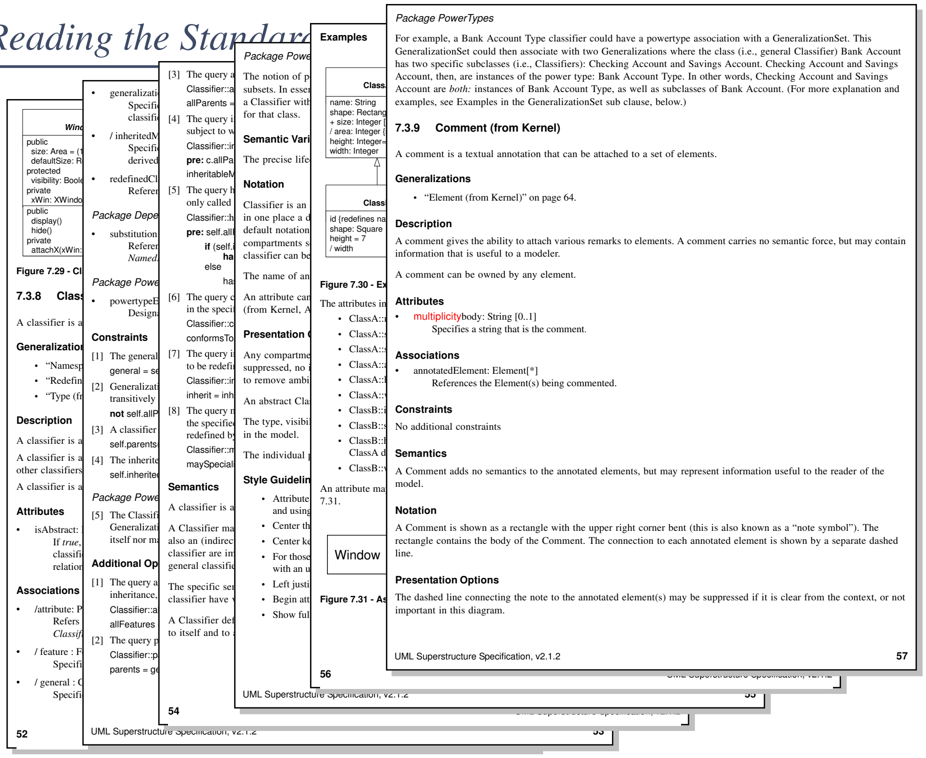


Figure 7.31 - Association-like notation for attribute

56 UML Superstructure Specification, v2.1.2

Reading the Standard



Package PowerTypes

For example, a Bank Account Type classifier could have a powertype association with a GeneralizationSet. This GeneralizationSet could then associate with two Generalizations where the class (i.e., general Classifier) Bank Account has two specific subclasses (i.e., Classifiers): Checking Account and Savings Account. Checking Account and Savings Account, then, are instances of the power type: Bank Account Type. In other words, Checking Account and Savings Account are *both:* instances of Bank Account Type, as well as subclasses of Bank Account. (For more explanation and examples, see Examples in the GeneralizationSet sub clause, below.)

7.3.9 Comment (from Kernel)

A comment is a textual annotation that can be attached to a set of elements.

Generalizations

- "Element (from Kernel)" on page 64.

Description

A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

A comment can be owned by any element.

Attributes

- multiplicitybody: String [0..1]
  Specifies a string that is the comment.

Associations

- annotatedElement: Element[*]
  References the Element(s) being commented.

Constraints

No additional constraints

Semantics

A Comment adds no semantics to the annotated elements, 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 to each annotated element is shown by a separate dashed line.

Presentation Options

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

UML Superstructure Specification, v2.1.2 57

Meta Object Facility (MOF)

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).

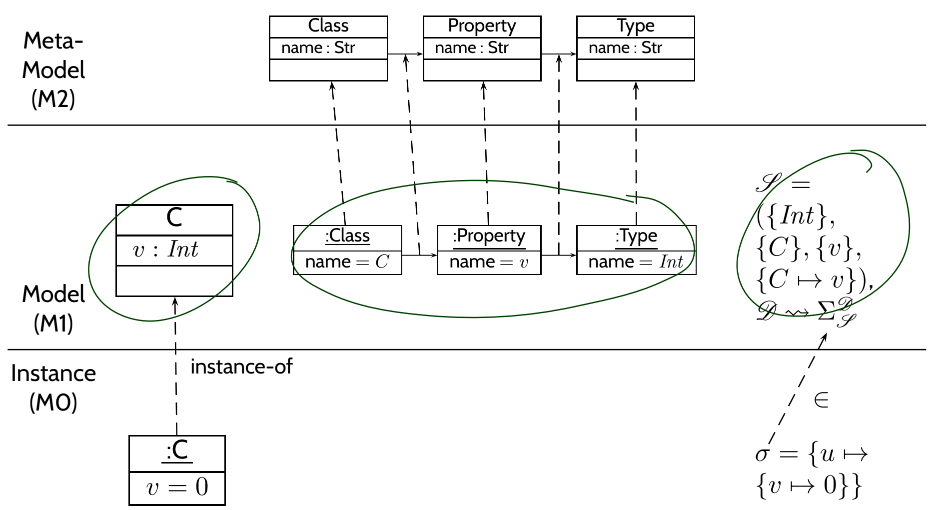
- So: things on meta level
  - M0 are object diagrams/system states
  - M1 are **words of the language UML**
  - M2 are **words of the language MOF**
  - M3 are **words of the language** ...MOF

Benefits

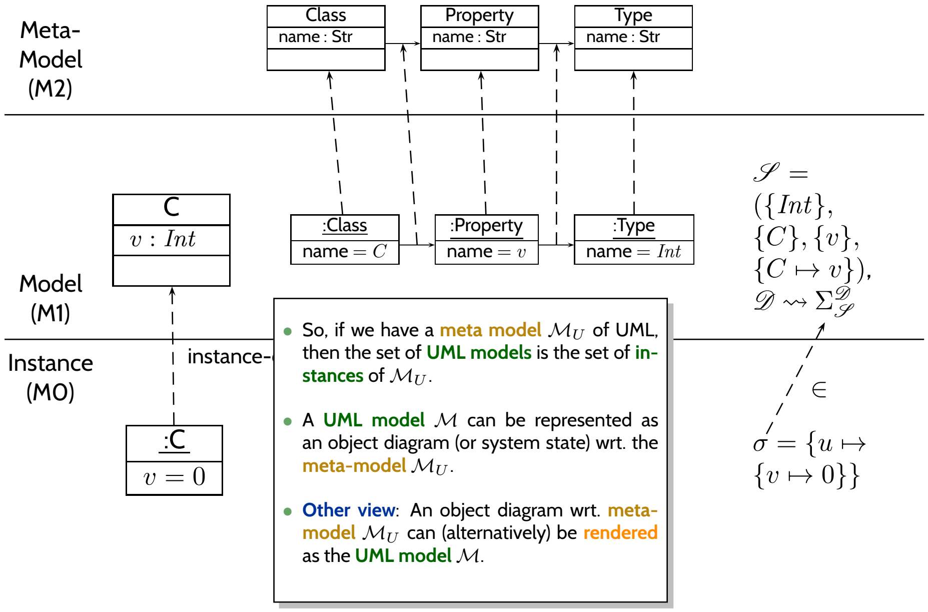
- In particular:
  - Benefits for **Modelling Tools**.
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.

Meta-Modelling: Principle

Modelling vs. Meta-Modelling



Modelling vs. Meta-Modelling



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

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**.

  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:

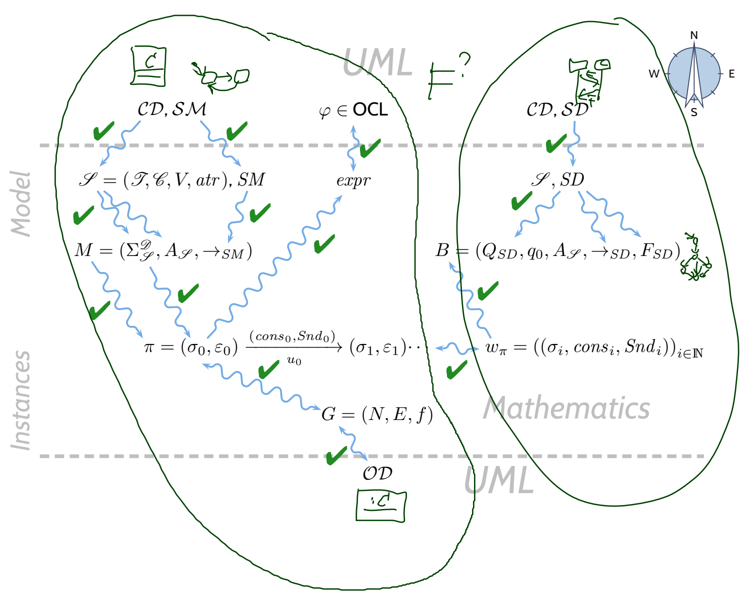
  Given a **UML model** $\mathcal{M}$, unfold it into an object diagram O_1 wrt. $\mathcal{M}_U$.

  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.

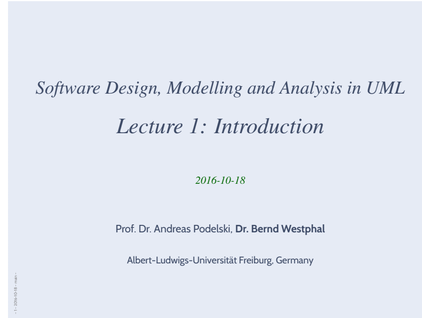
And That's It!

The Map



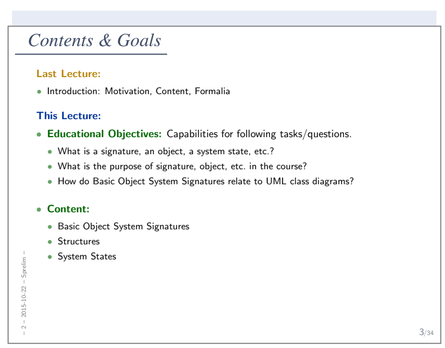
Content

- **Lecture 1**: Introduction



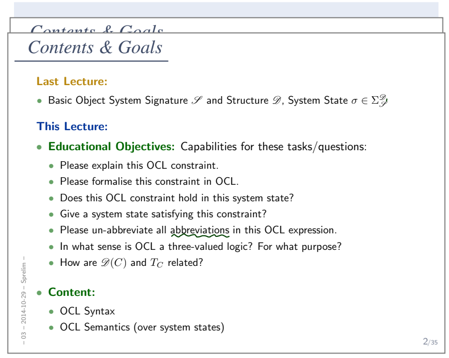
Content

- **Lecture 1**: Introduction
- **Lecture 2**: Semantical Model



Content

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



Contents & Goals

Last Lecture:

- Basic Object System Signature $\mathscr{S}$ and Structure $\mathscr{D}$, System State $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$

This Lecture:

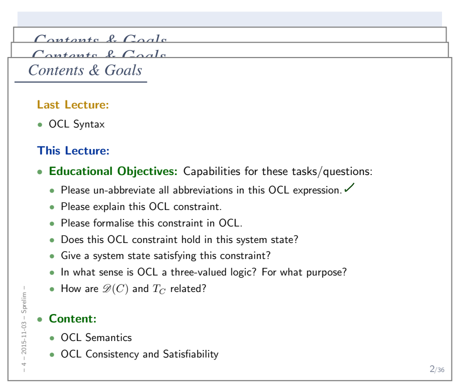
- **Educational Objectives:** Capabilities for these tasks/questions:
  - Please explain this OCL constraint.
  - Please formalise this constraint in OCL.
  - Does this OCL constraint hold in this system state?
  - Give a system state satisfying this constraint?
  - Please un-abbreviate all abbreviations in this OCL expression.
  - In what sense is OCL a three-valued logic? For what purpose?
  - How are $\mathscr{D}(C)$ and T_C related?
- **Content:**
  - OCL Syntax
  - OCL Semantics (over system states)

– 03 – 2014-10-29 – Sprelim –

2/35

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:

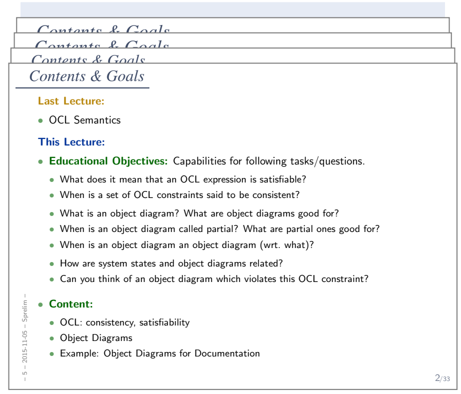
- **Educational Objectives:** Capabilities for these tasks/questions:
  - Please un-abbreviate all abbreviations in this OCL expression. ✓
  - Please explain this OCL constraint.
  - Please formalise this constraint in OCL.
  - Does this OCL constraint hold in this system state?
  - Give a system state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

– 4 – 2015-11-03 – Sprelim –

2/36

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:

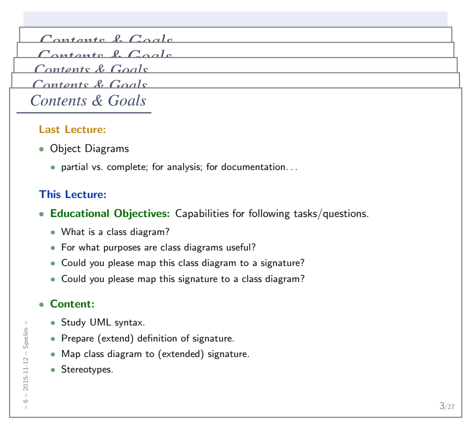
- **Educational Objectives:** Capabilities for following tasks/questions.
  - What does it mean that an OCL expression is satisfiable?
  - When is a set of OCL constraints said to be consistent?
  - What is an object diagram? What are object diagrams good for?
  - When is an object diagram called partial? 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

– 5 – 2015-11-05 – Sprelim –

2/33

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 analysis; for documentation. . .

This Lecture:

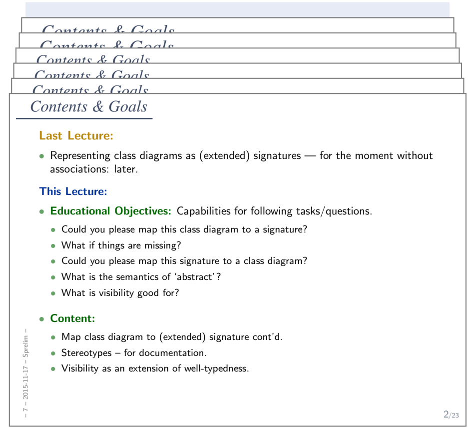
- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is a class diagram?
  - For what purposes are class diagrams useful?
  - Could you please map this class diagram to a signature?
  - Could you please map this signature to a class diagram?
- **Content:**
  - Study UML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

– 6 – 2015-11-12 – Sprelim –

3/27

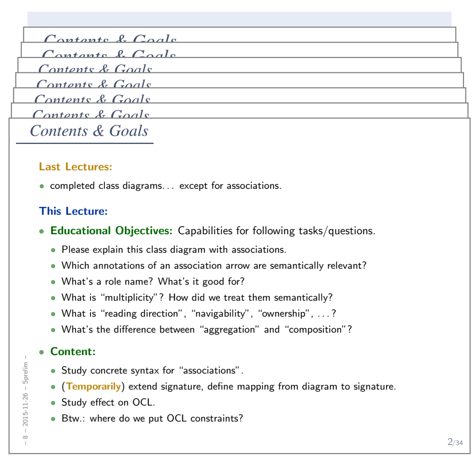
Content

- **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
- **Lecture 7**: Class Diagrams II



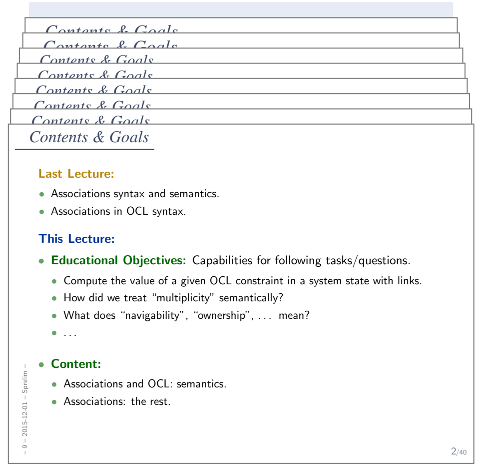
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III



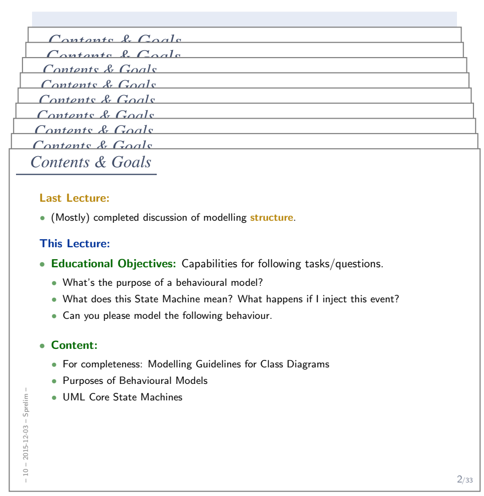
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV



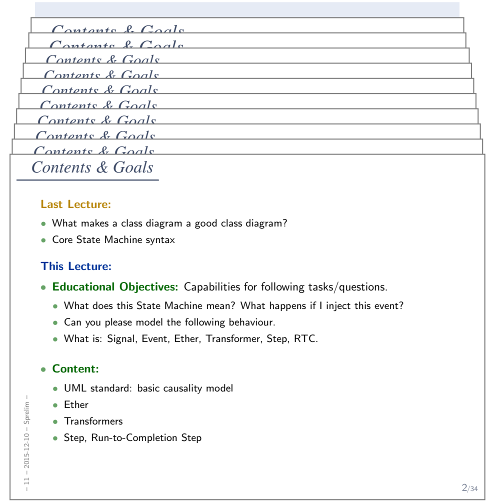
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview



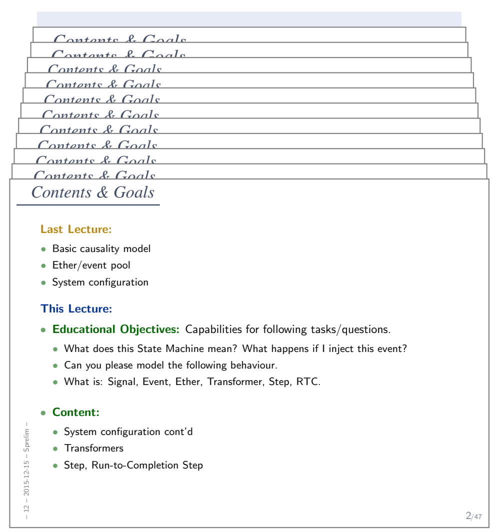
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I



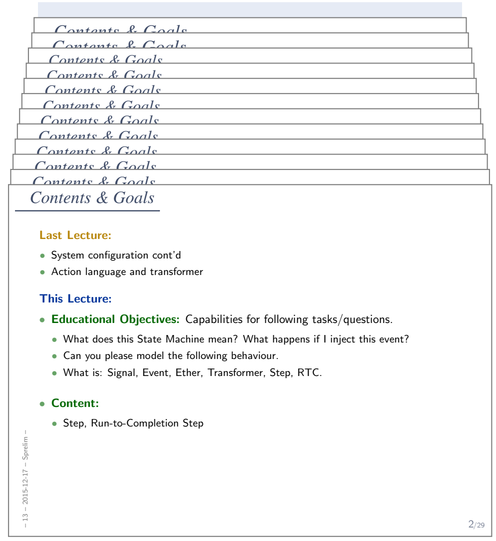
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II



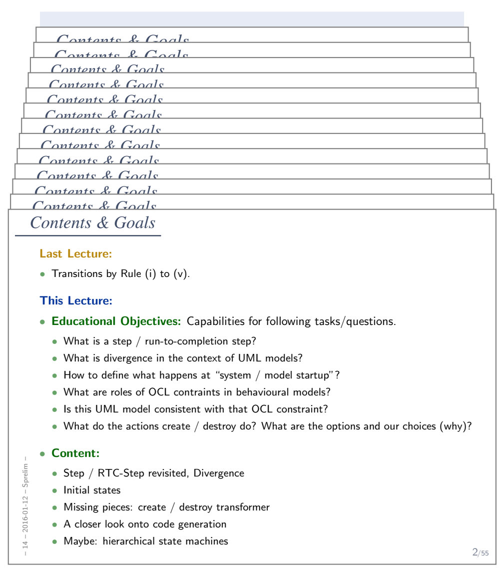
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III



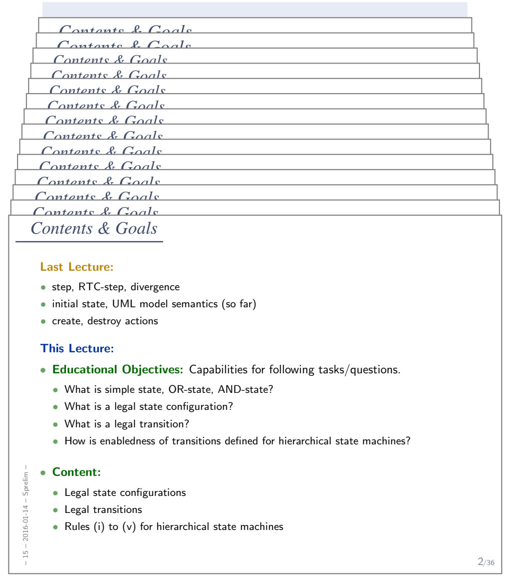
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III
- **Lecture 14**: Hierarchical State Machines I



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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III
- **Lecture 14**: Hierarchical State Machines I
- **Lecture 15**: Hierarchical State Machines II



Contents & Goals

Last Lecture:
- step, RTC-step, divergence
- initial state, UML model semantics (so far)
- create, destroy actions

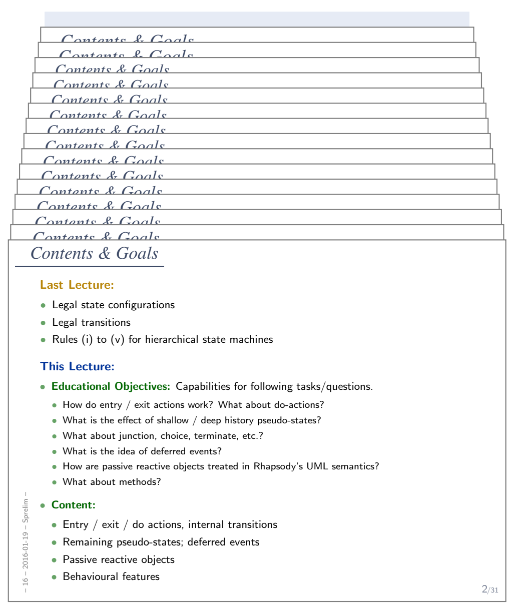
This Lecture:
- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is simple state, OR-state, AND-state?
  - What is a legal state configuration?
  - What is a legal transition?
  - How is enabledness of transitions defined for hierarchical state machines?
- **Content:**
  - Legal state configurations
  - Legal transitions
  - Rules (i) to (v) for hierarchical state machines

– 15 – 2016-01-14 – Sprelim –

2/36

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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III
- **Lecture 14**: Hierarchical State Machines I
- **Lecture 15**: Hierarchical State Machines II
- **Lecture 16**: Hierarchical State Machines III



Contents & Goals

Last Lecture:
- Legal state configurations
- Legal transitions
- Rules (i) to (v) for hierarchical state machines

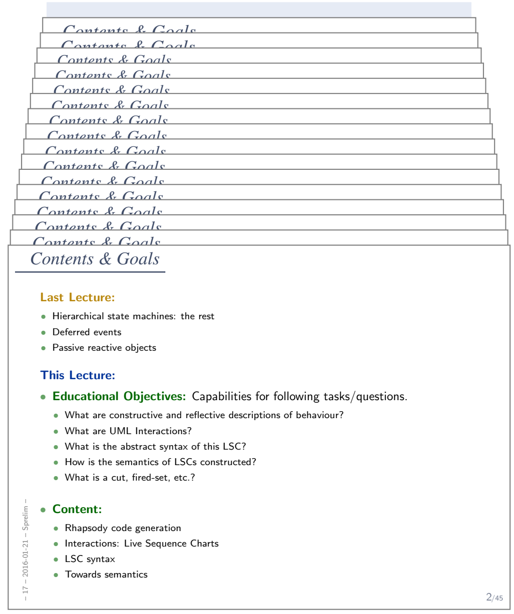
This Lecture:
- **Educational Objectives:** Capabilities for following tasks/questions.
  - How do entry / exit actions work? What about do-actions?
  - What is the effect of shallow / deep history pseudo-states?
  - What about junction, choice, terminate, etc.?
  - What is the idea of deferred events?
  - How are passive reactive objects treated in Rhapsody's UML semantics?
  - What about methods?
- **Content:**
  - Entry / exit / do actions, internal transitions
  - Remaining pseudo-states; deferred events
  - Passive reactive objects
  - Behavioural features

– 16 – 2016-01-19 – Sprelim –

2/31

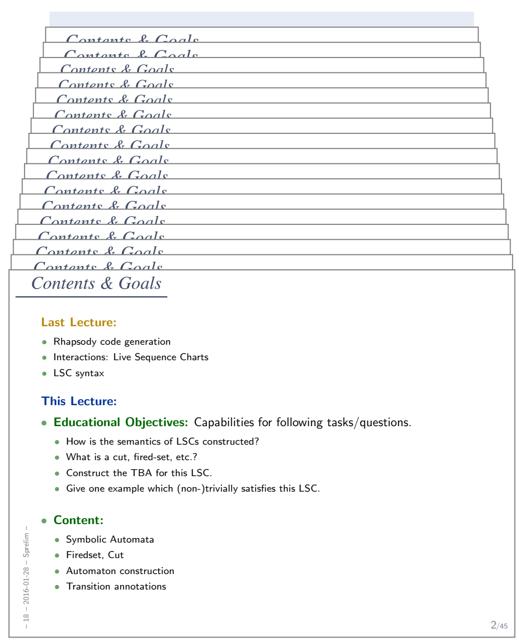
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III
- **Lecture 14**: Hierarchical State Machines I
- **Lecture 15**: Hierarchical State Machines II
- **Lecture 16**: Hierarchical State Machines III
- **Lecture 17**: Live Sequence Charts I

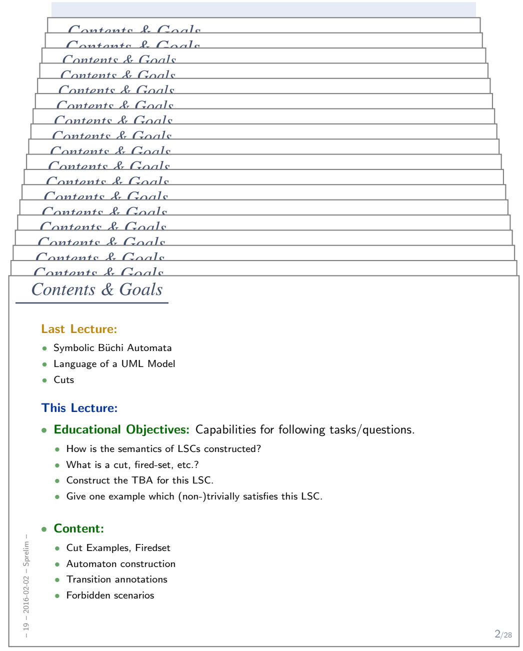


Content

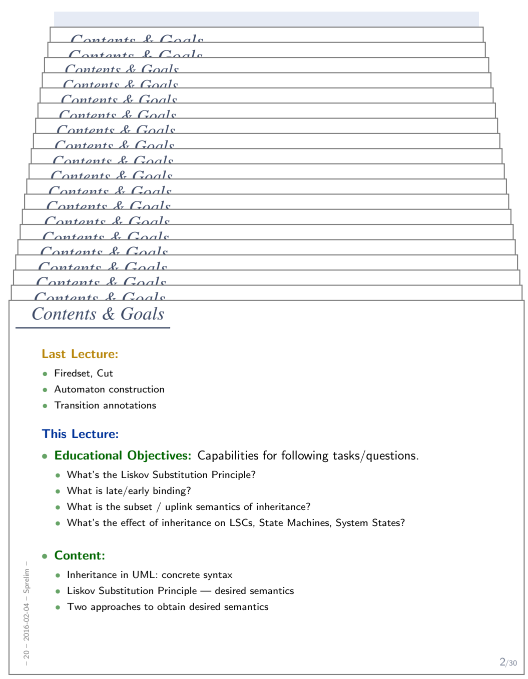
- **Lecture 1**: Introduction
- **Lecture 2**: Semantical Model
- **Lecture 3**: Object Constraint Language (OCL)
- **Lecture 4**: OCL Semantics
- **Lecture 5**: Object Diagrams
- **Lecture 6**: Class Diagrams I
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III
- **Lecture 14**: Hierarchical State Machines I
- **Lecture 15**: Hierarchical State Machines II
- **Lecture 16**: Hierarchical State Machines III
- **Lecture 17**: Live Sequence Charts I
- **Lecture 18**: Live Sequence Charts II



Content

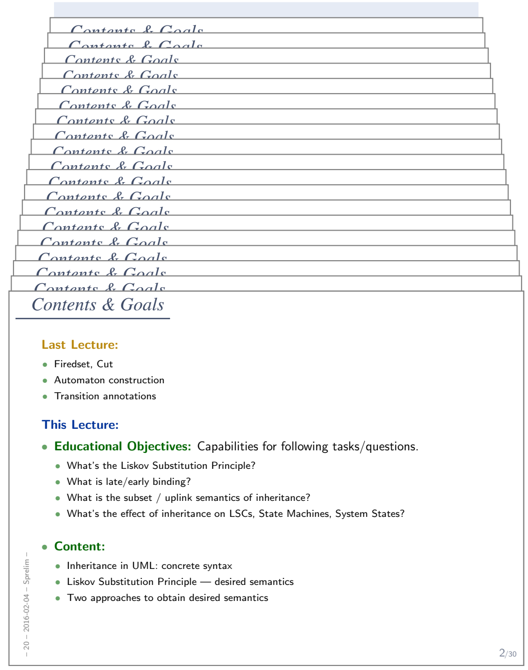


Content



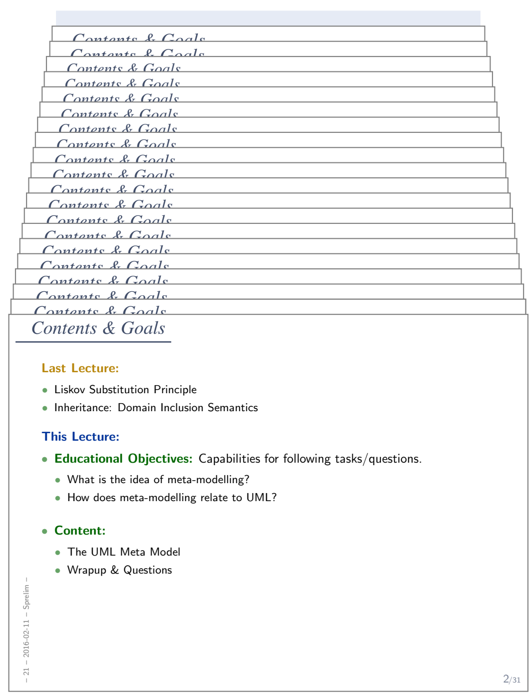
Content

- **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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III
- **Lecture 14**: Hierarchical State Machines I
- **Lecture 15**: Hierarchical State Machines II
- **Lecture 16**: Hierarchical State Machines III
- **Lecture 17**: Live Sequence Charts I
- **Lecture 18**: Live Sequence Charts II
- **Lecture 19**: Live Sequence Charts III
- **Lecture 20**: Live Sequence Charts IV
- **Lecture 21**: MBSE & Inheritance



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
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview
- **Lecture 11**: Core State Machines I
- **Lecture 12**: Core State Machines II
- **Lecture 13**: Core State Machines III
- **Lecture 14**: Hierarchical State Machines I
- **Lecture 15**: Hierarchical State Machines II
- **Lecture 16**: Hierarchical State Machines III
- **Lecture 17**: Live Sequence Charts I
- **Lecture 18**: Live Sequence Charts II
- **Lecture 19**: Live Sequence Charts III
- **Lecture 20**: Live Sequence Charts IV
- **Lecture 21**: MBSE & Inheritance
- **Lecture 22**: Meta-Modelling



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.