Software Design, Modelling and Analysis in UML

Lecture 21: Meta-Modelling

2016-02-11

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- Liskov Substitution Principle
- Inheritance: Domain Inclusion Semantics

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is the idea of meta-modelling?
  - How does meta-modelling relate to UML?

- **Content:**
  - The UML Meta Model
  - Wrapup & Questions

Meta-Modelling: Idea

- **Meta-Modelling** is one major prerequisite for understanding
  - the standard documents OMG (2007a,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 **model** UML models using a modelling language?

- In other words:

  Why not have a model $\mathcal{M}_U$ such that

  - the set of legal instances of $\mathcal{M}_U$

  is

  - the set of well-formed (!) UML models.

Meta-Modelling: Example

$\mathcal{D}(V) = \{+, -, \#\}$

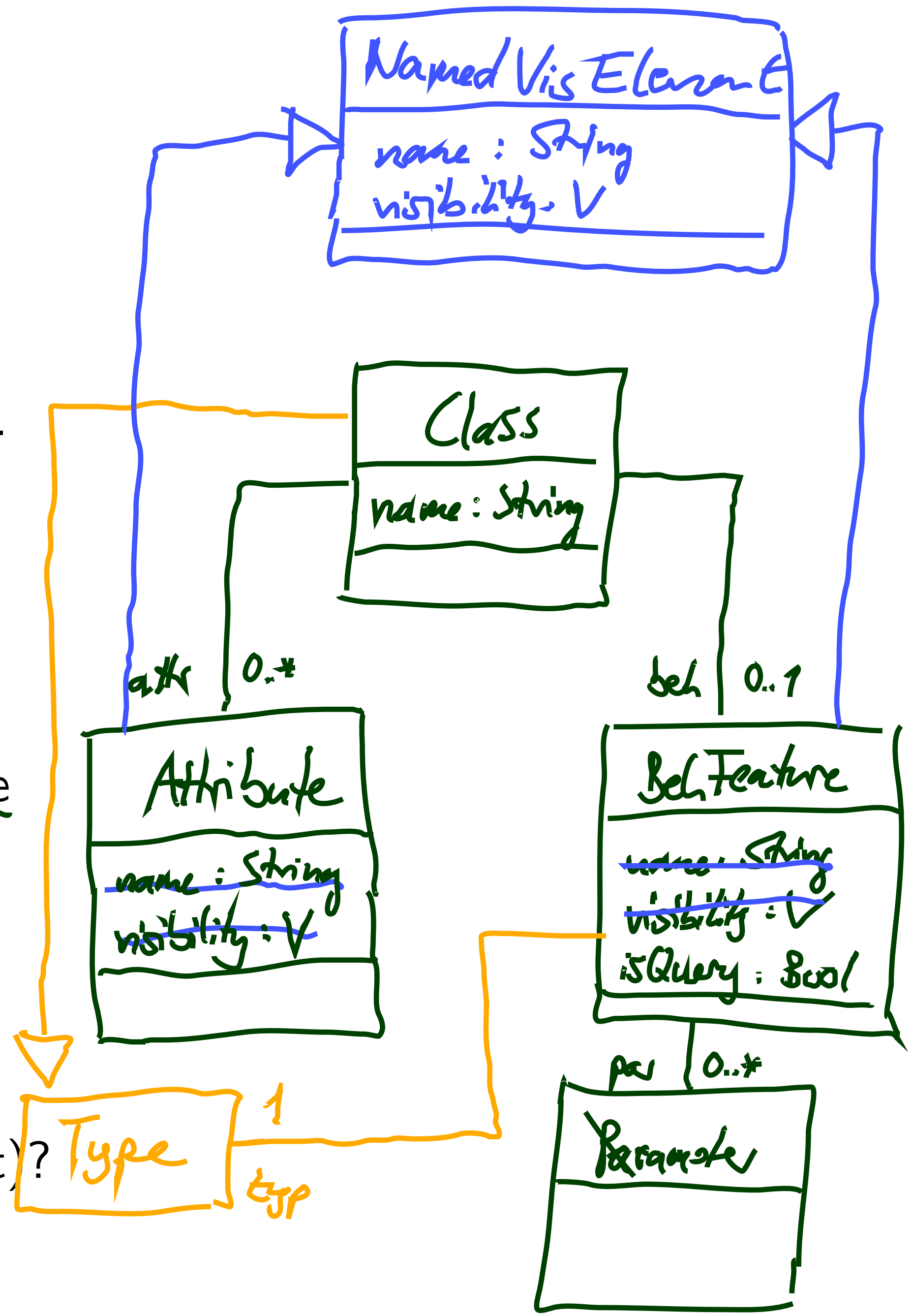
For example, let's consider a class.

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

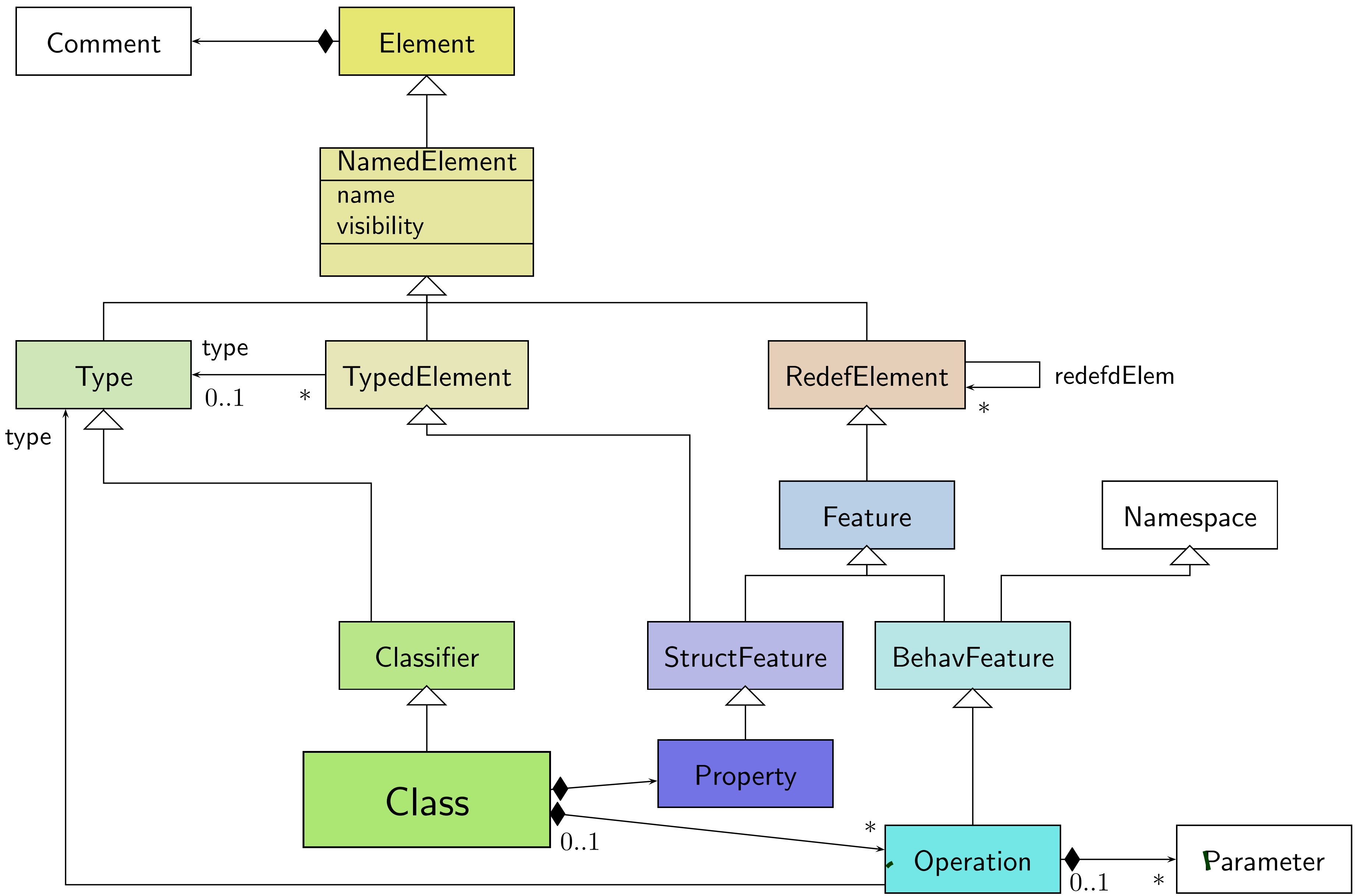
  Each of the latter two has
  - a **name** and
  - a **visibility**.

  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)?

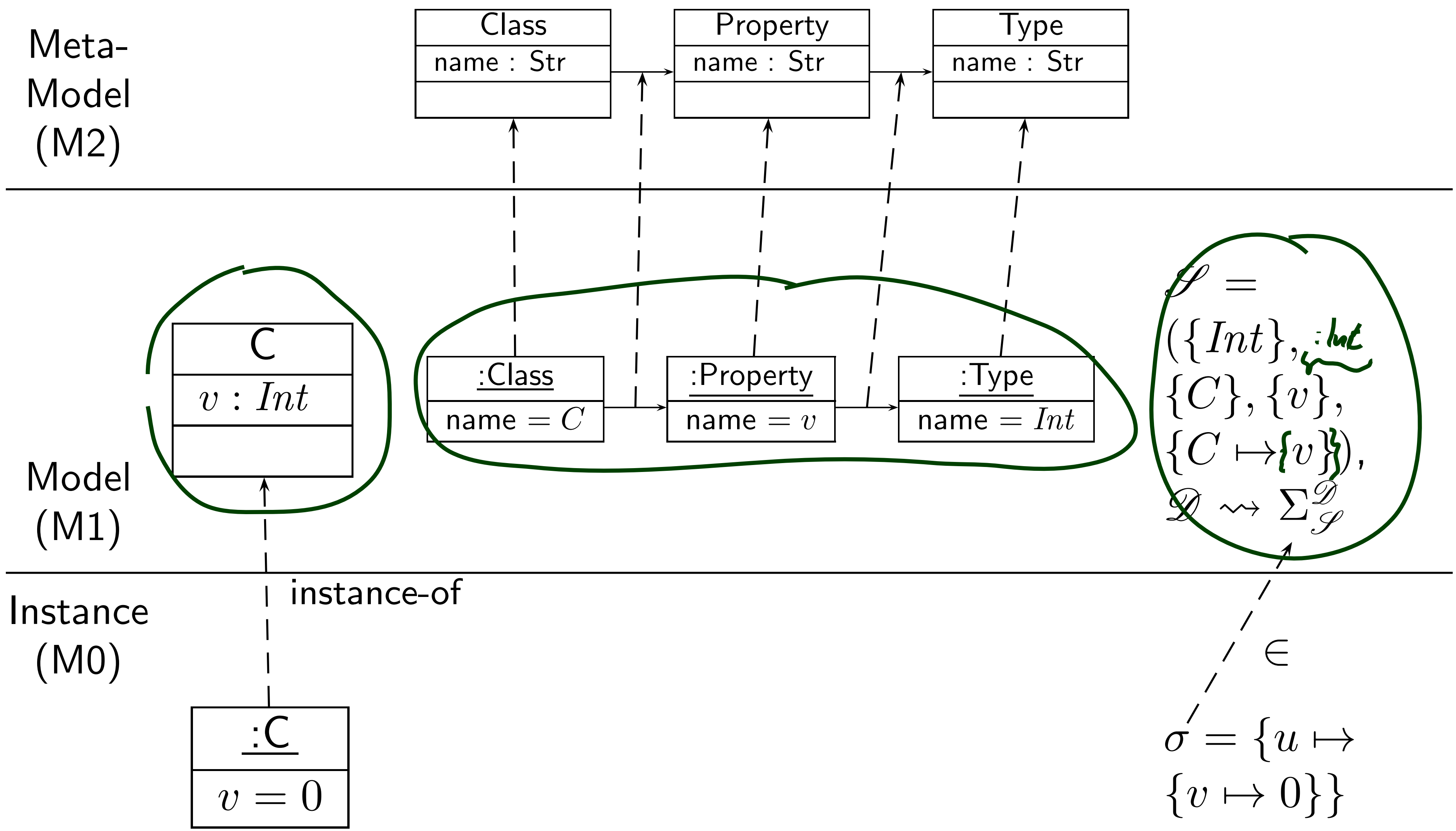


UML Meta-Model: Extract from UML 2.0 Standard

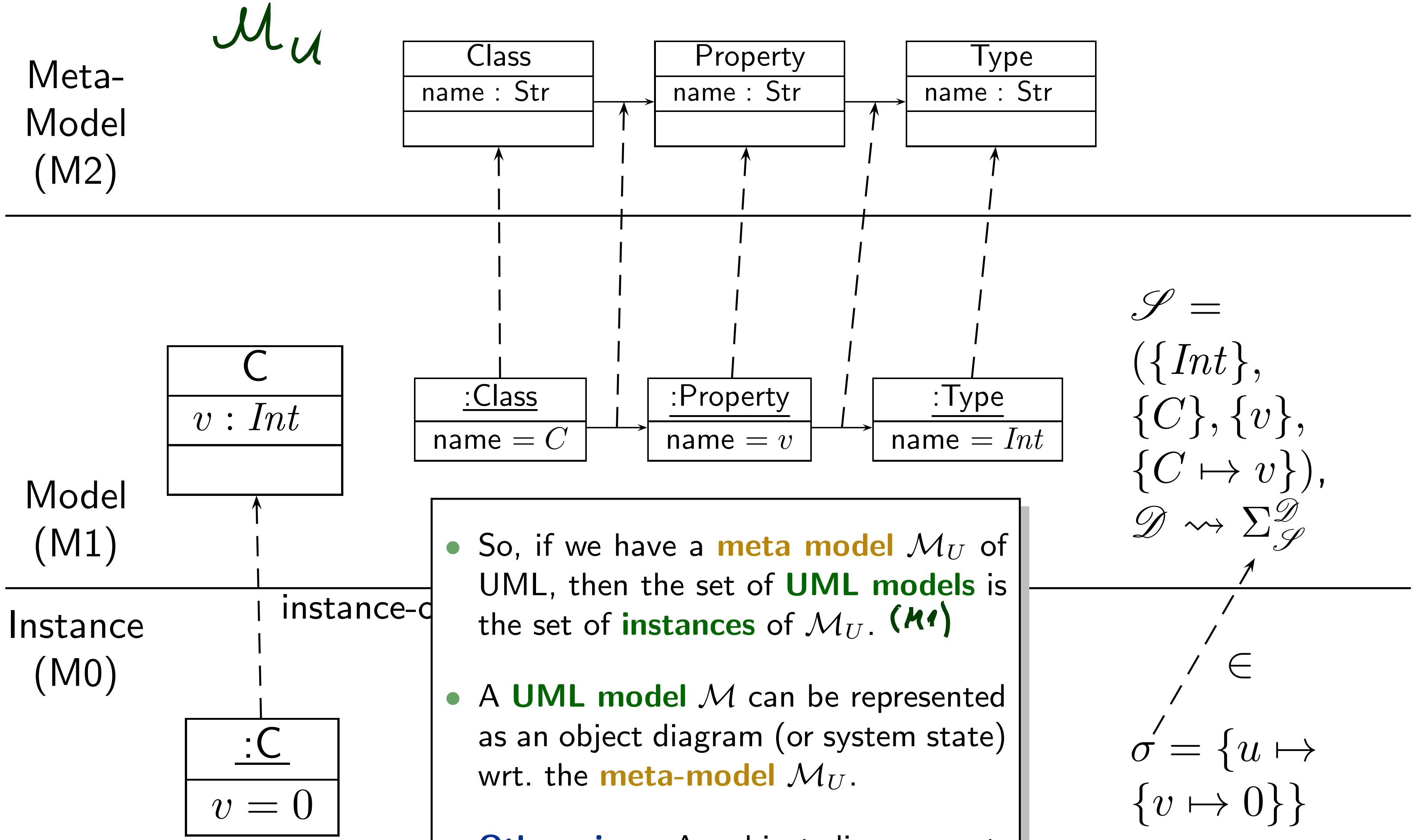


Meta-Modelling: Principle

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 $\mathcal{M}_U$. (M1)
- 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:

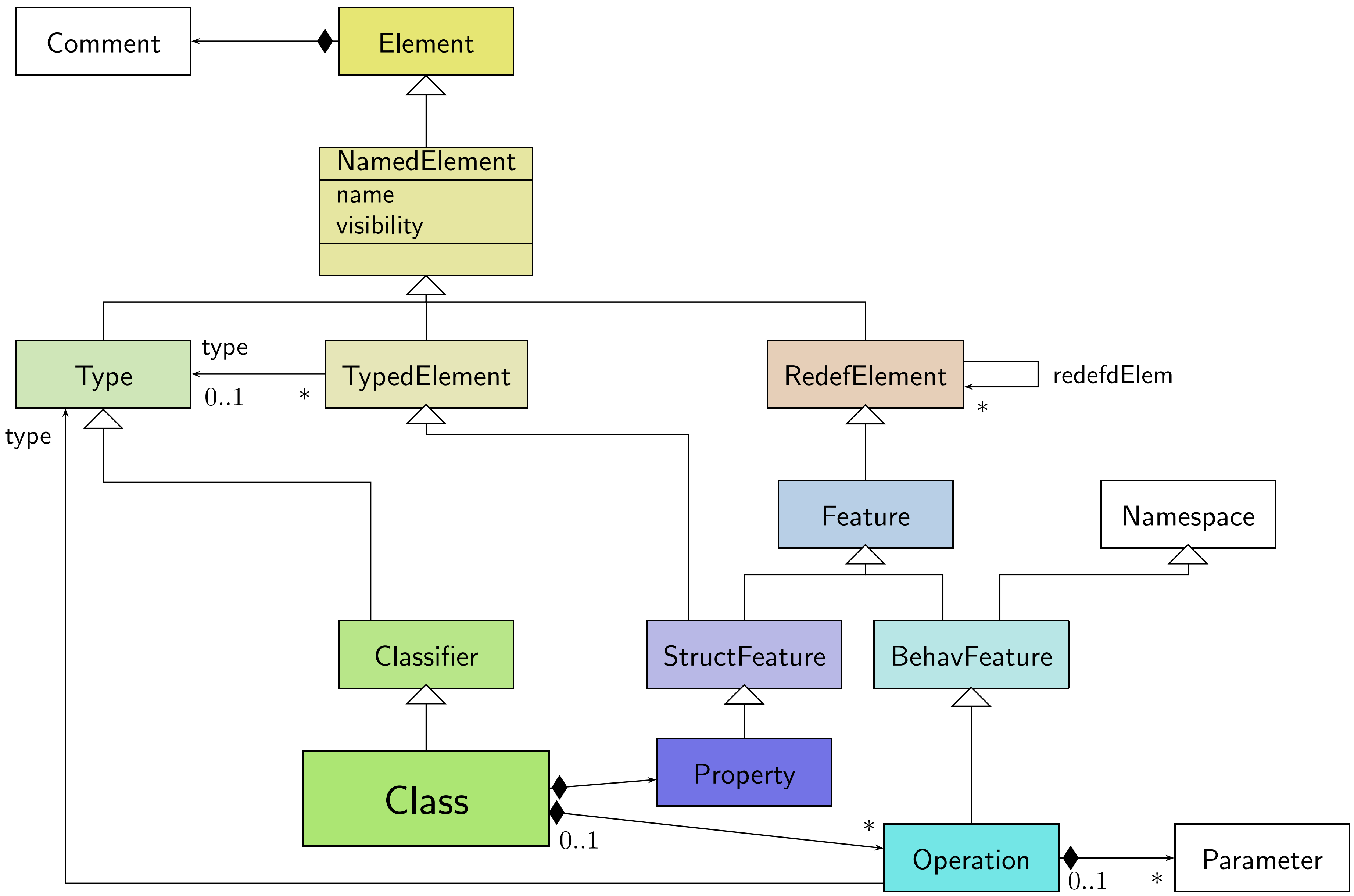
  Given a **UML model** $\mathcal{M}$, unfold it into an object diagram O_1 wrt. $\mathcal{M}_U$.

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

The UML 2.x Standard Revisited

Claim: Extract from UML 2.0 Standard



Classes (OMG, 2007b, 32)

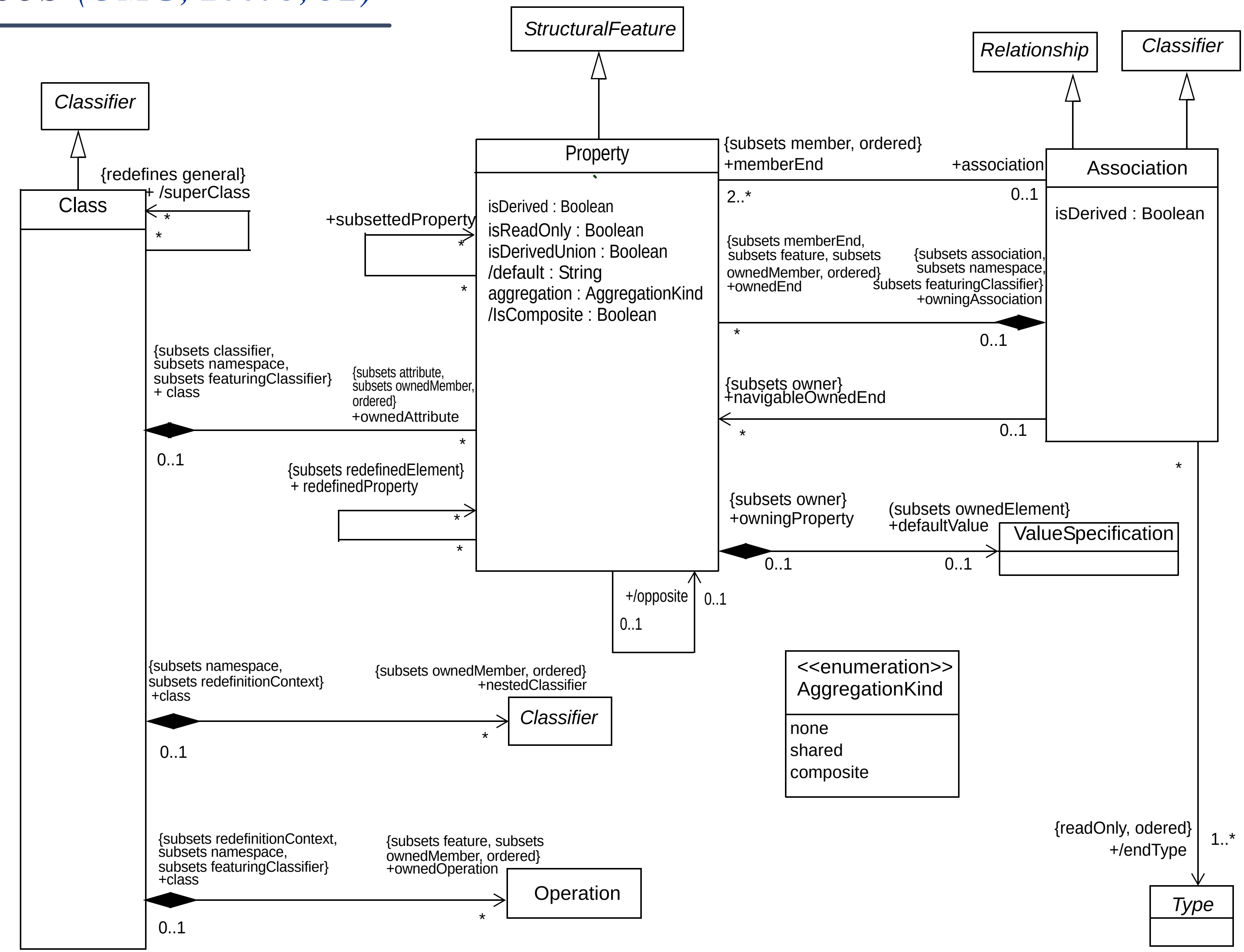


Figure 7.12 - Classes diagram of the Kernel package

– 21 – 2016-02-11 – Sumlmm –

Operations (OMG, 2007b, 31)

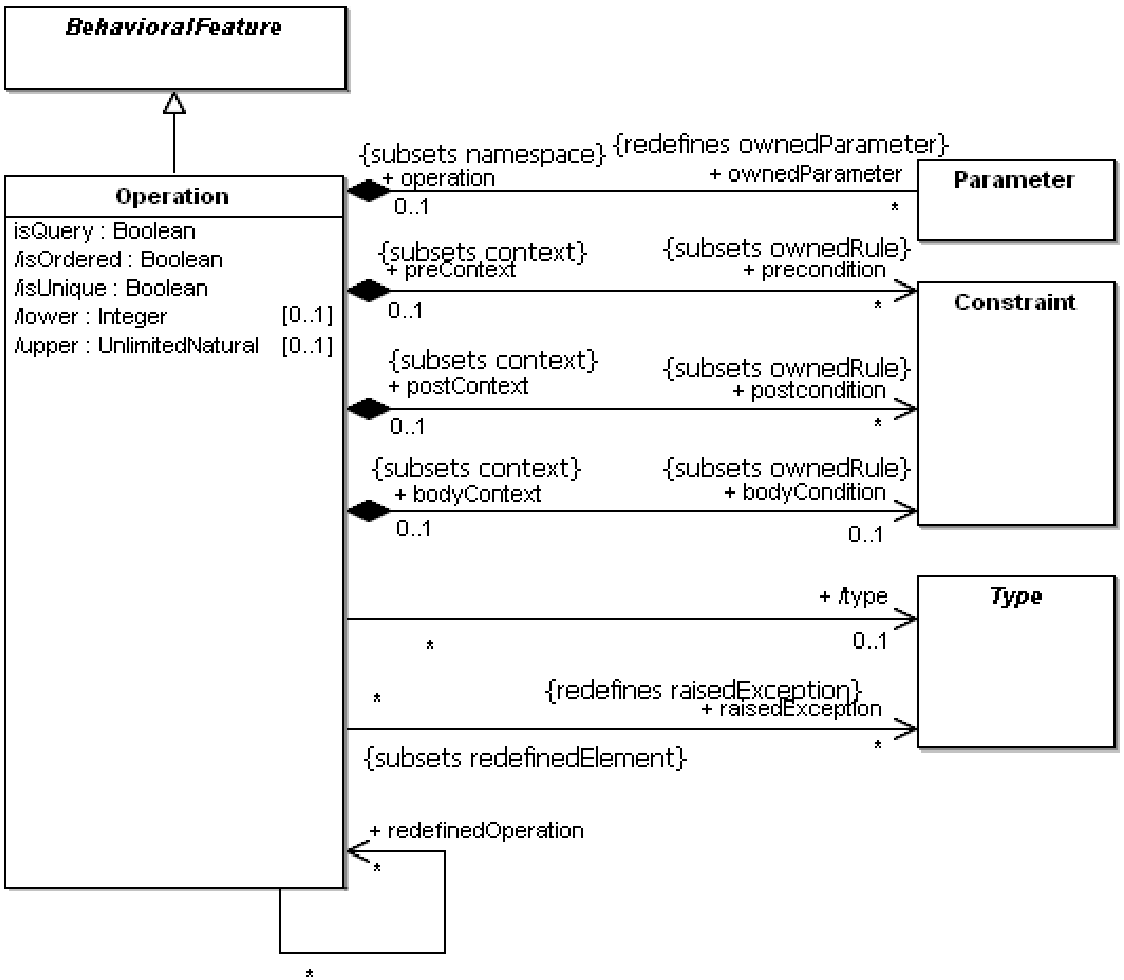


Figure 7.11 - Operations diagram of the Kernel package

Operations (OMG, 2007b, 30)

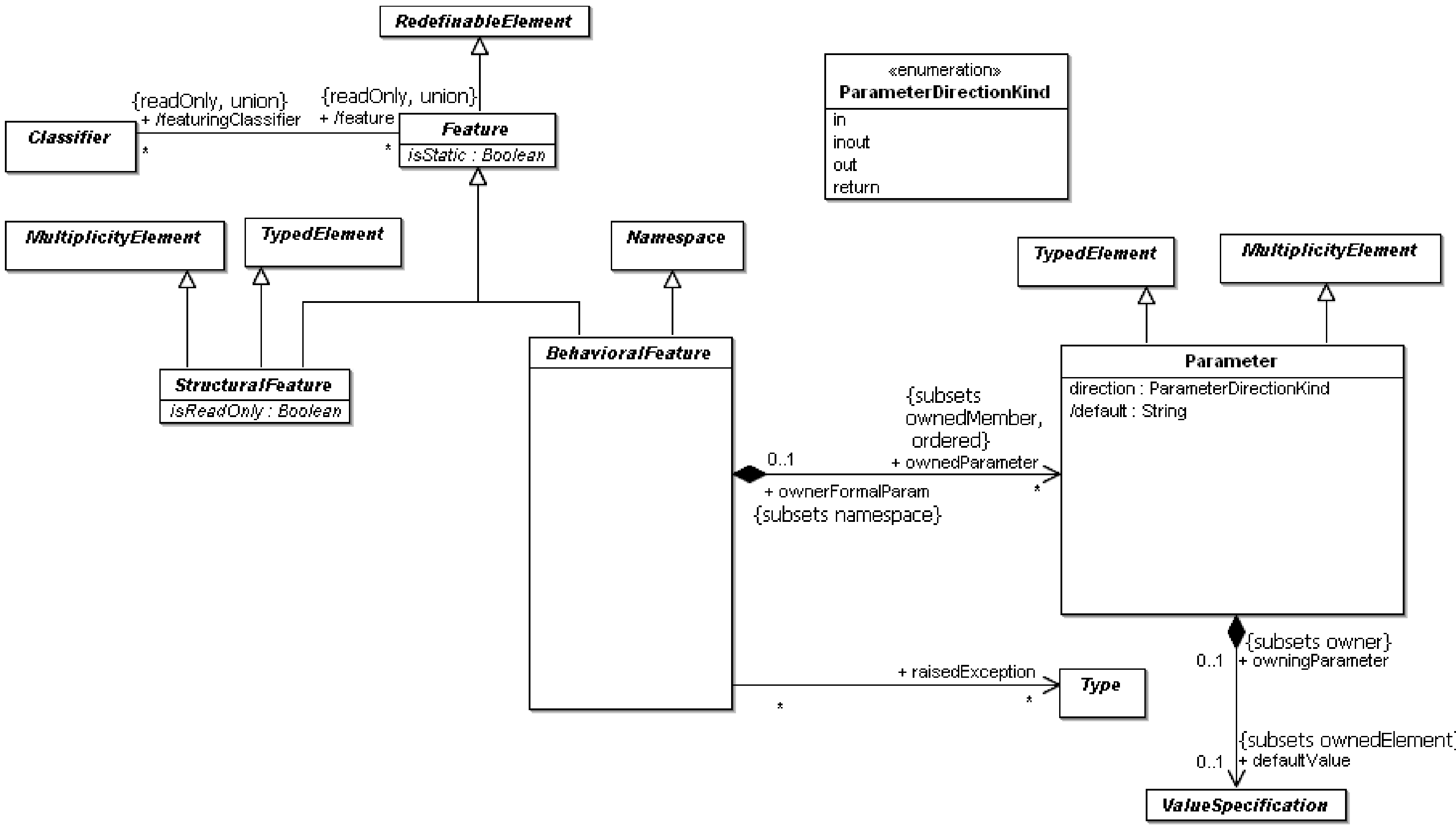


Figure 7.10 - Features diagram of the Kernel package

Classifiers (OMG, 2007b, 29)

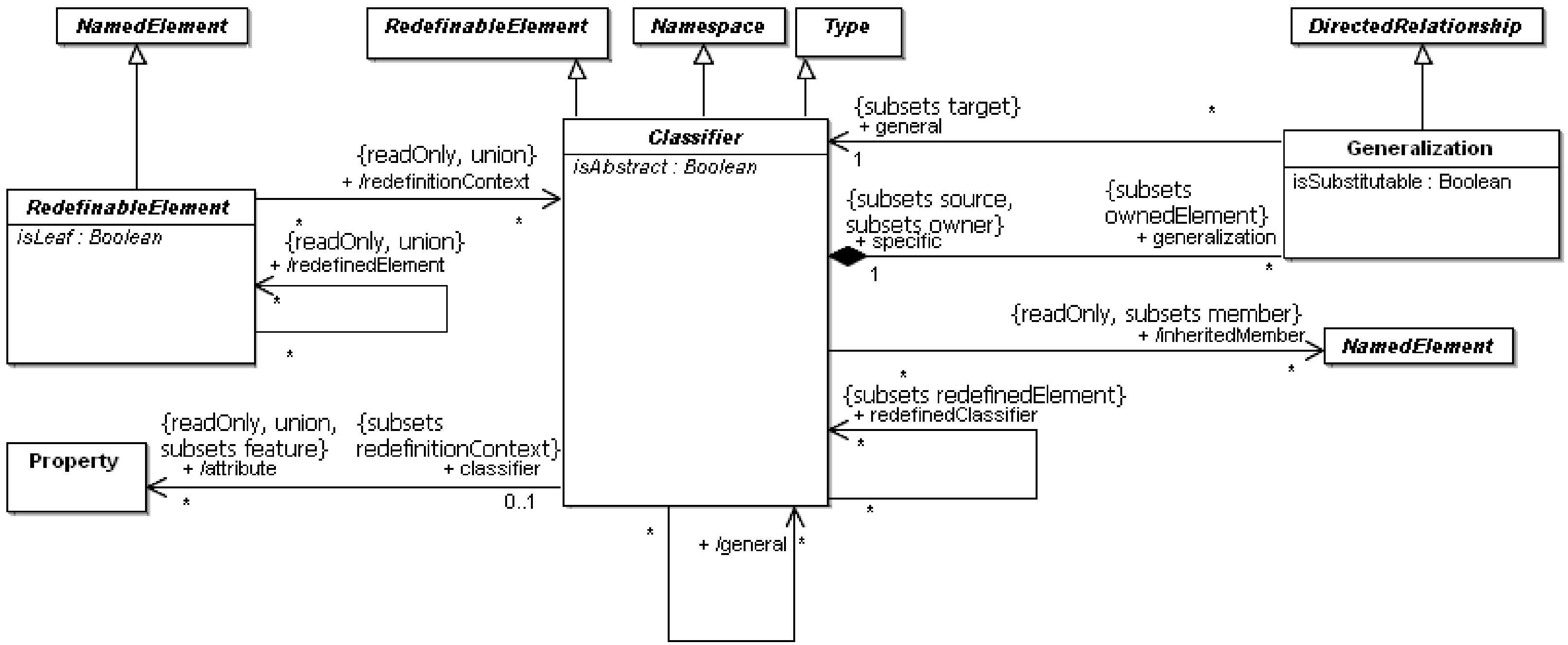


Figure 7.9 - Classifiers diagram of the Kernel package

Namespaces (OMG, 2007b, 26)

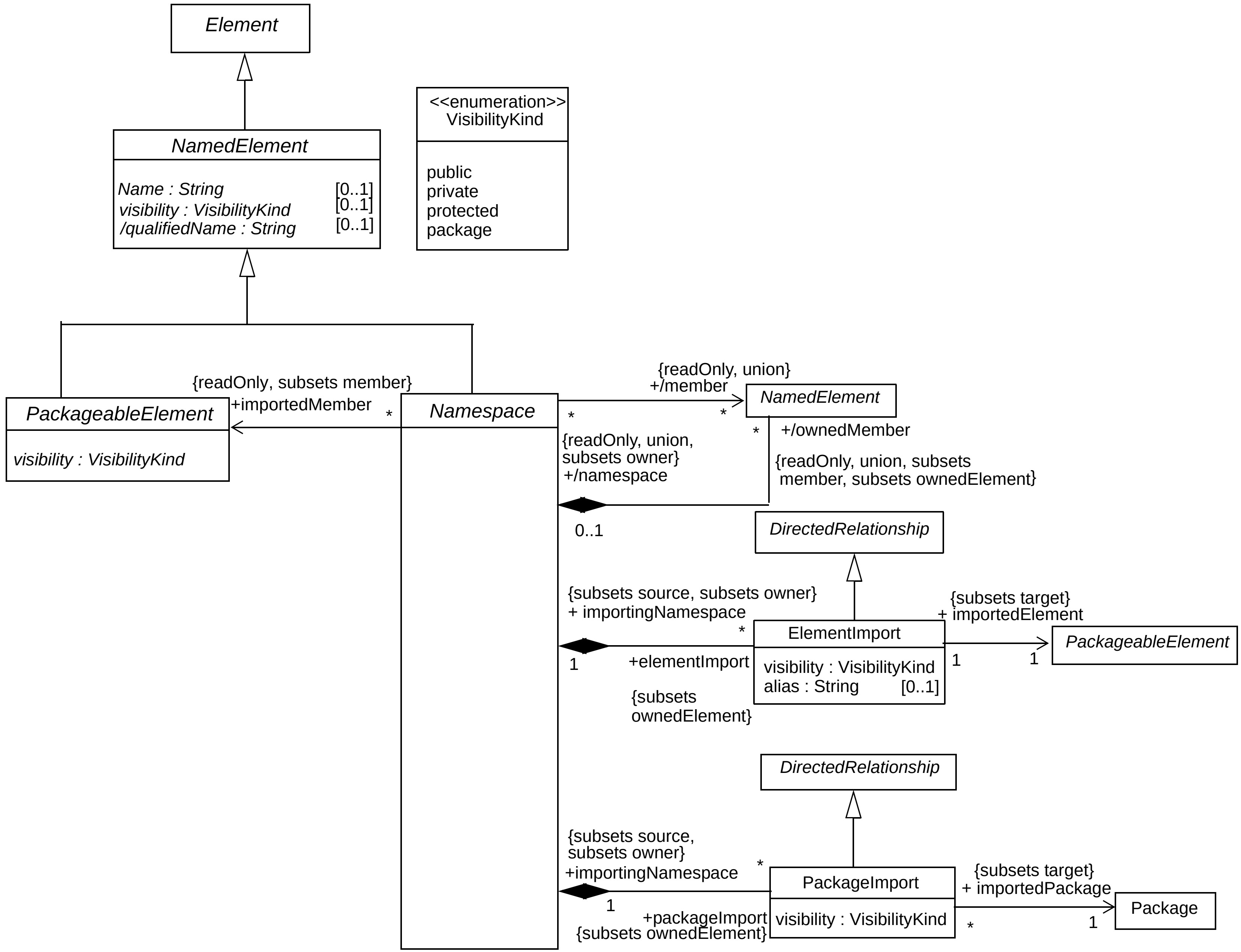


Figure 7.4 - Namespaces diagram of the Kernel package

Root Diagram (OMG, 2007b, 25)

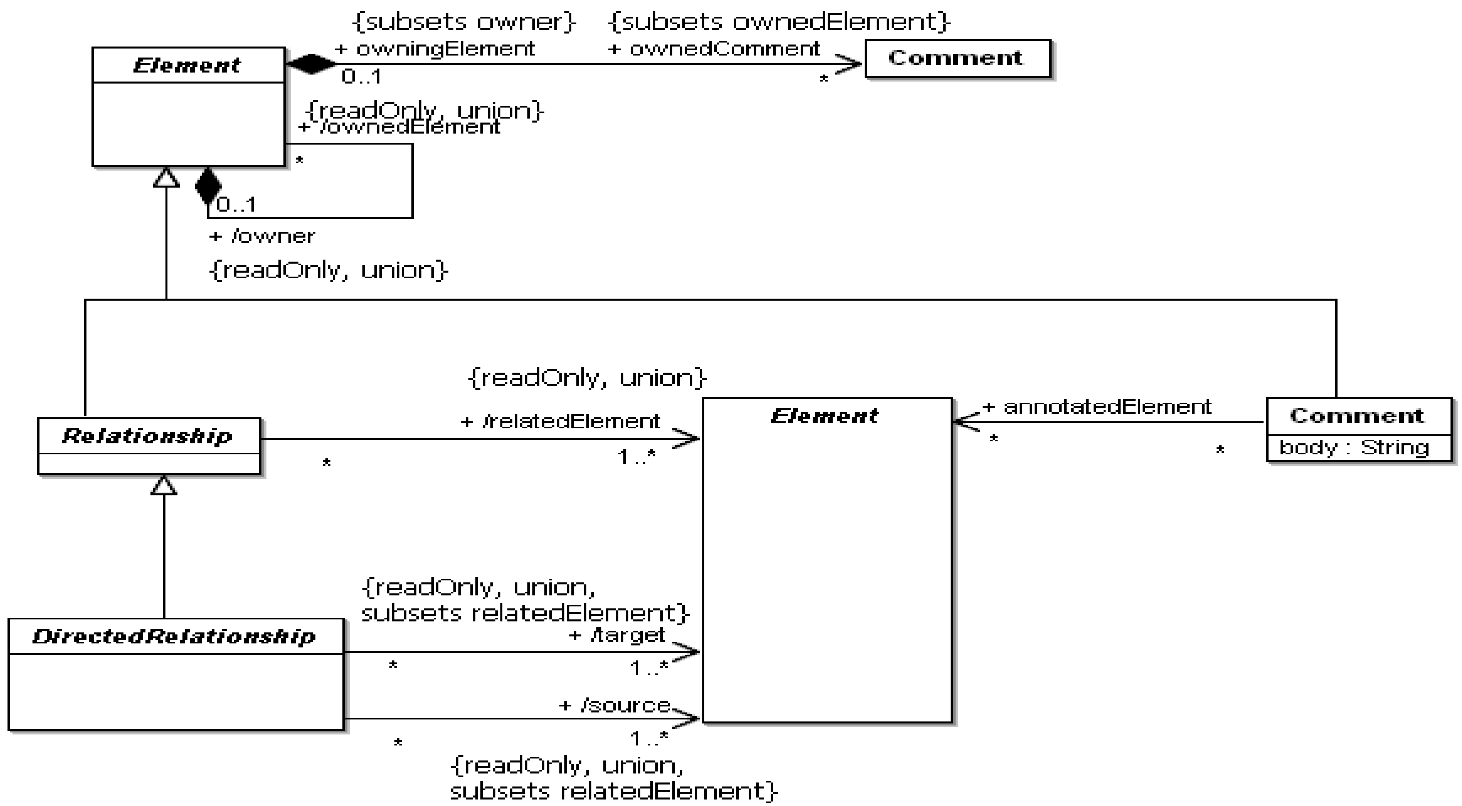


Figure 7.3 - Root diagram of the Kernel package

Interesting: Declaration/Definition (OMG, 2007b, 424)

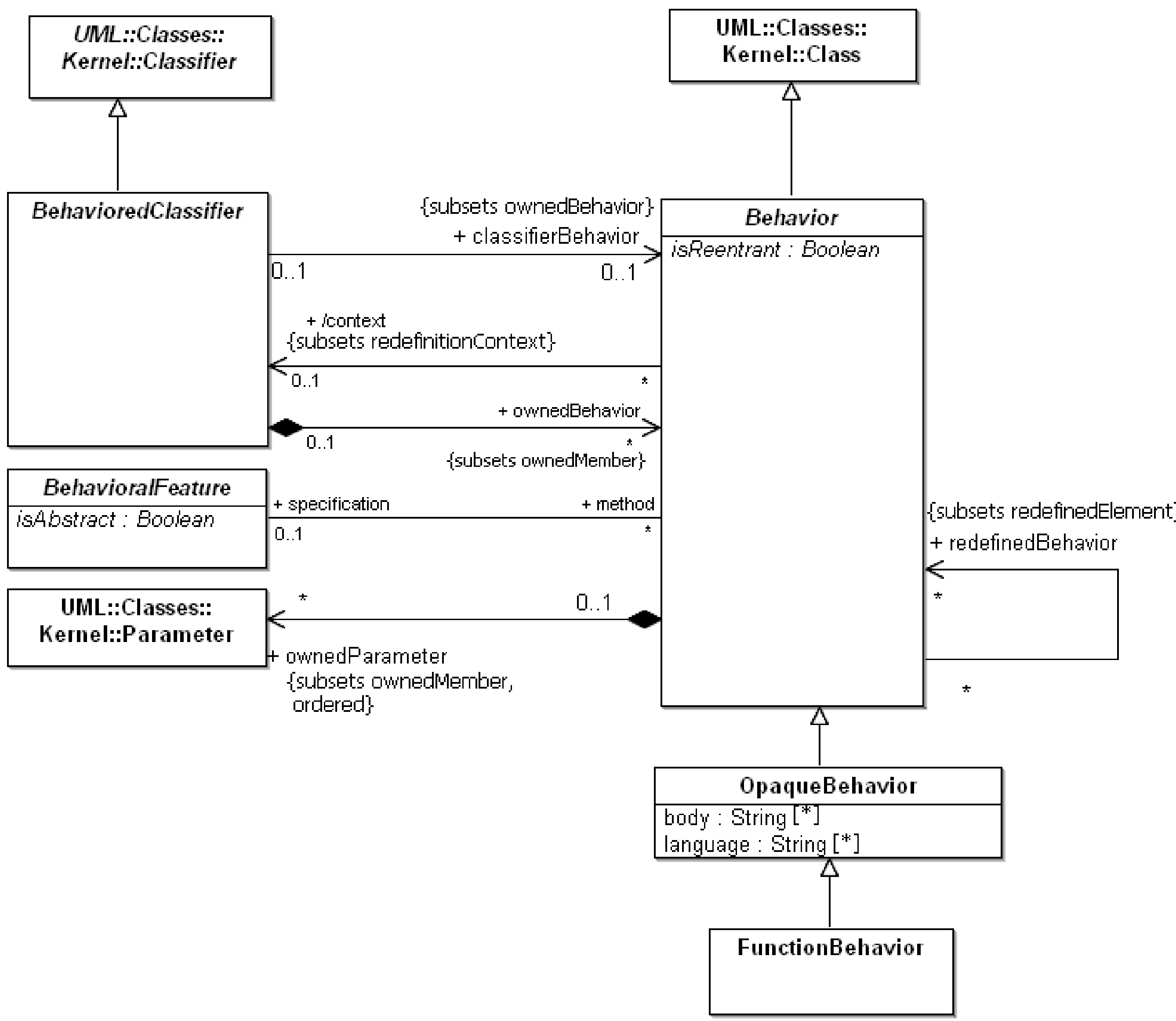
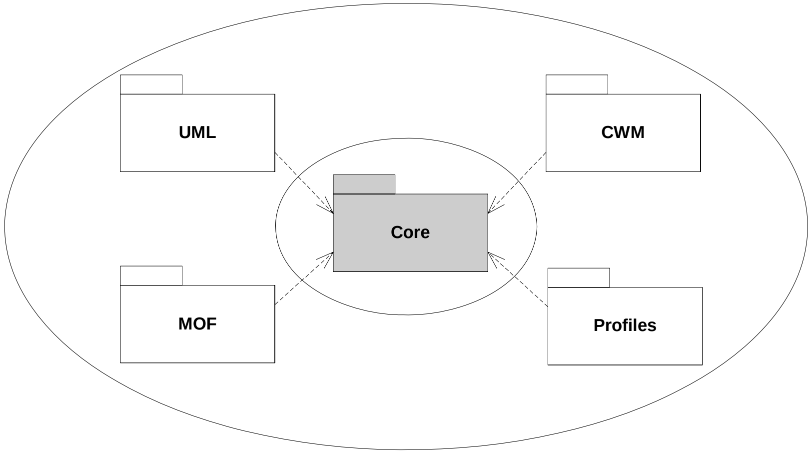


Figure 13.6 - Common Behavior

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.



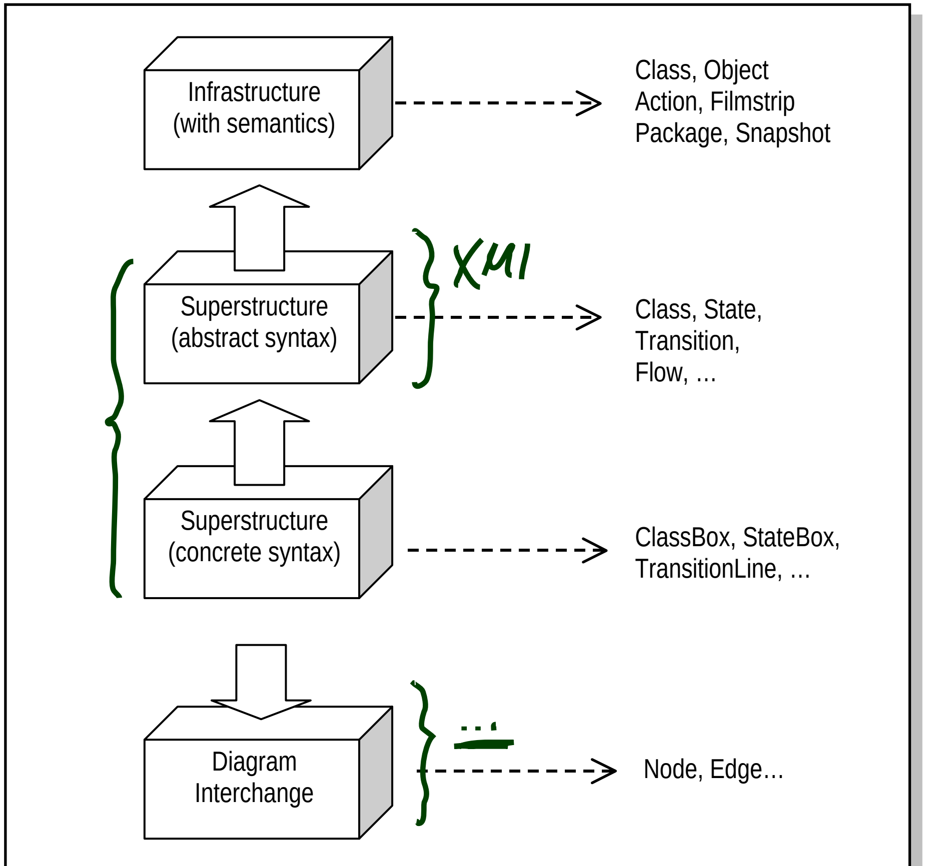


Figure 0-1 *Overview of architecture*

UML Superstructure Packages (OMG, 2007a, 15)

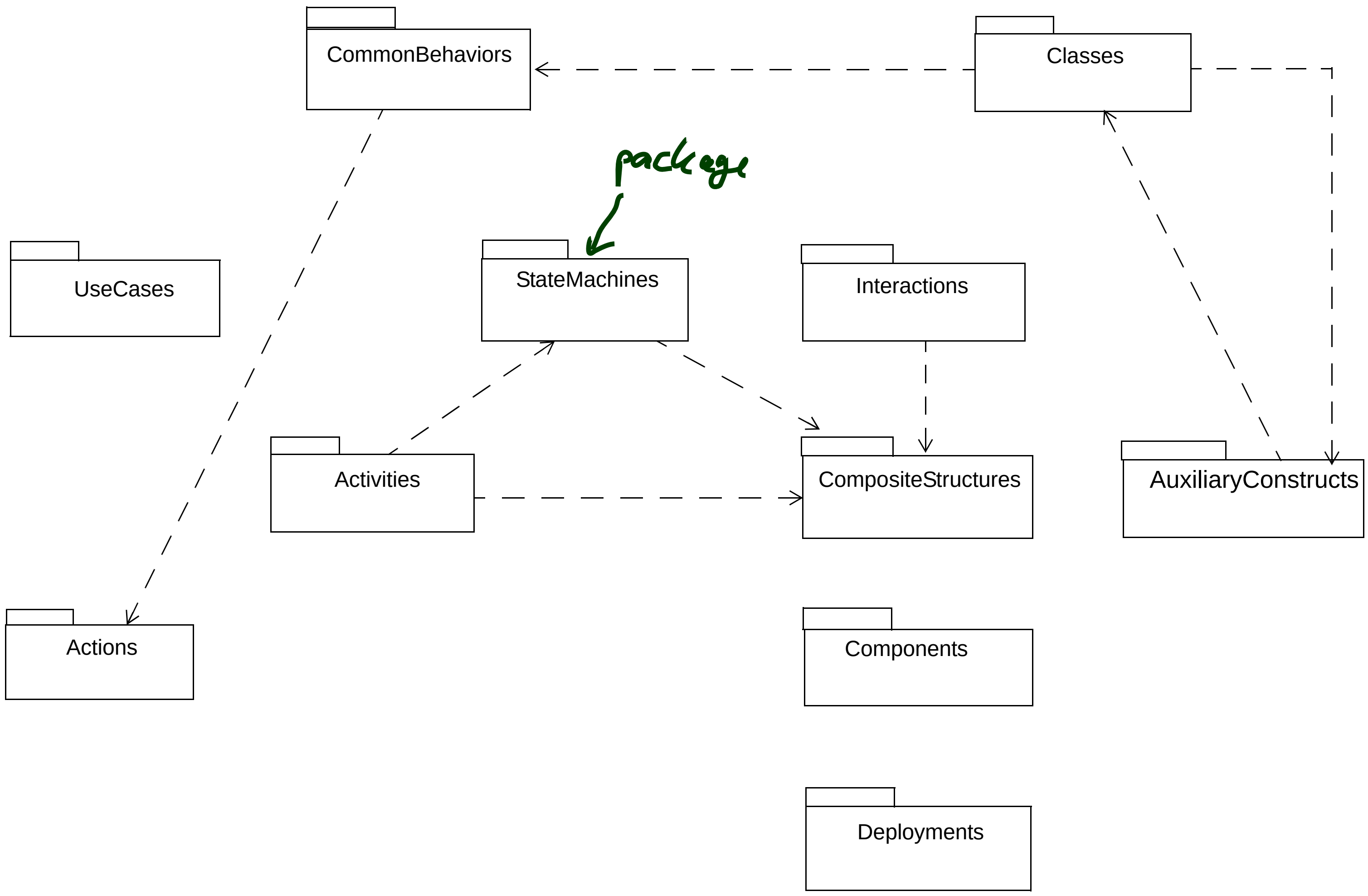


Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

Reading the Standard

Table of Contents

Reading the Standard

Table of Contents

Reading the Standard

Table of Contents

Reading the Standard Cont’d

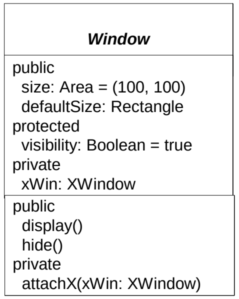


Figure 7.29 - Class notation: attributes and operations grouped according to visibility

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

Wind
public
size: Area = (1
defaultSize: R
protected
visibility: Boole
private
xWin: XWindo
public
display()
hide()
private
attachX(xWin:

Figure 7.29 - Cl

7.3.8 Class

A classifier is a

Generalizatio

- “Namesp
- “Redefin
- “Type (fr

Description

A classifier is a

A classifier is a
other classifiers

A classifier is a

Attributes

- isAbstract: I
  If *true*,
  classifi
  relation

Associations

- /attribute: P
  Refers
  Classif
- / feature : F
  Specifi
- / general : C
  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
  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

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

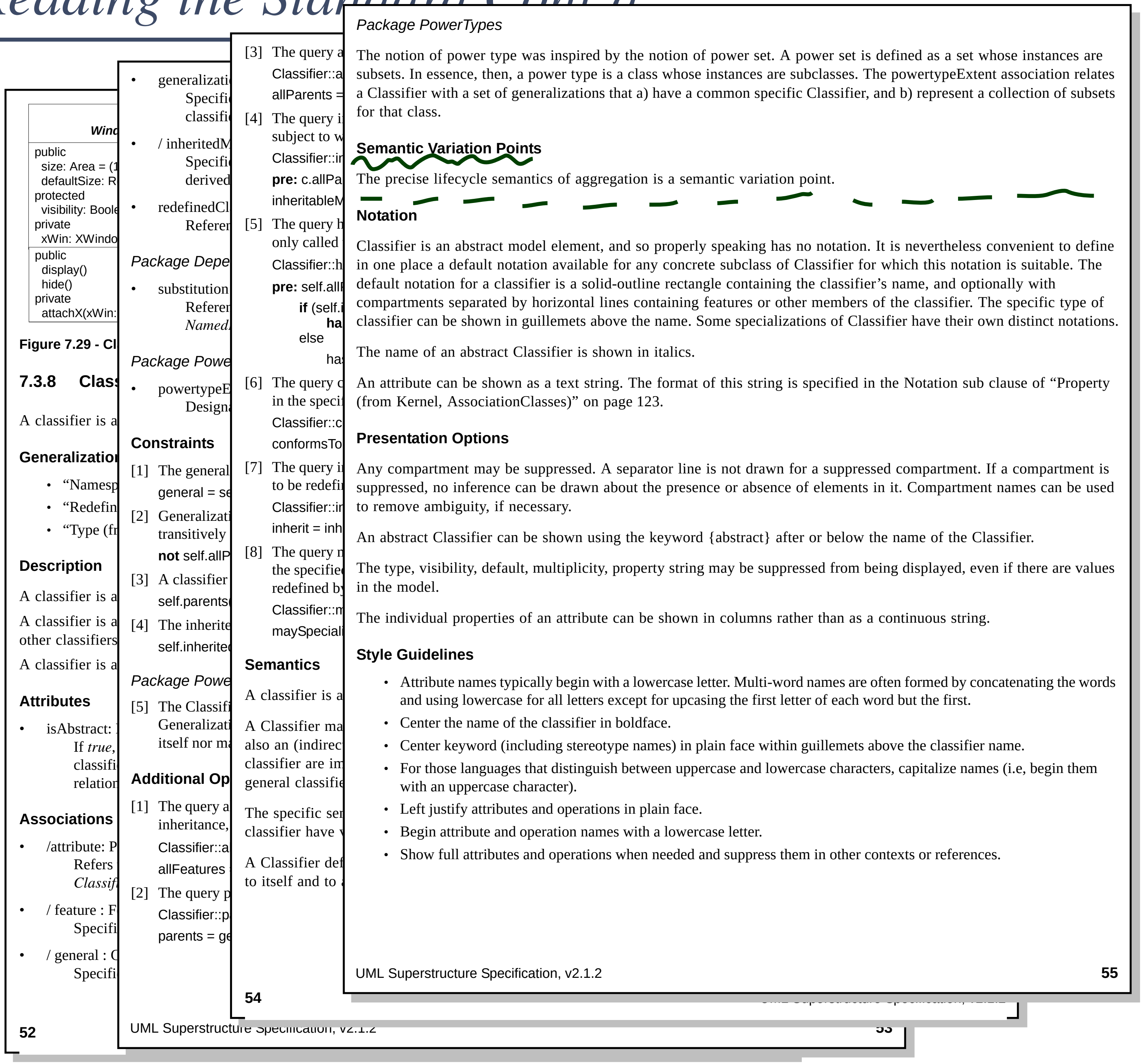
A classifier is a classification of instances according to their features.

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.

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.

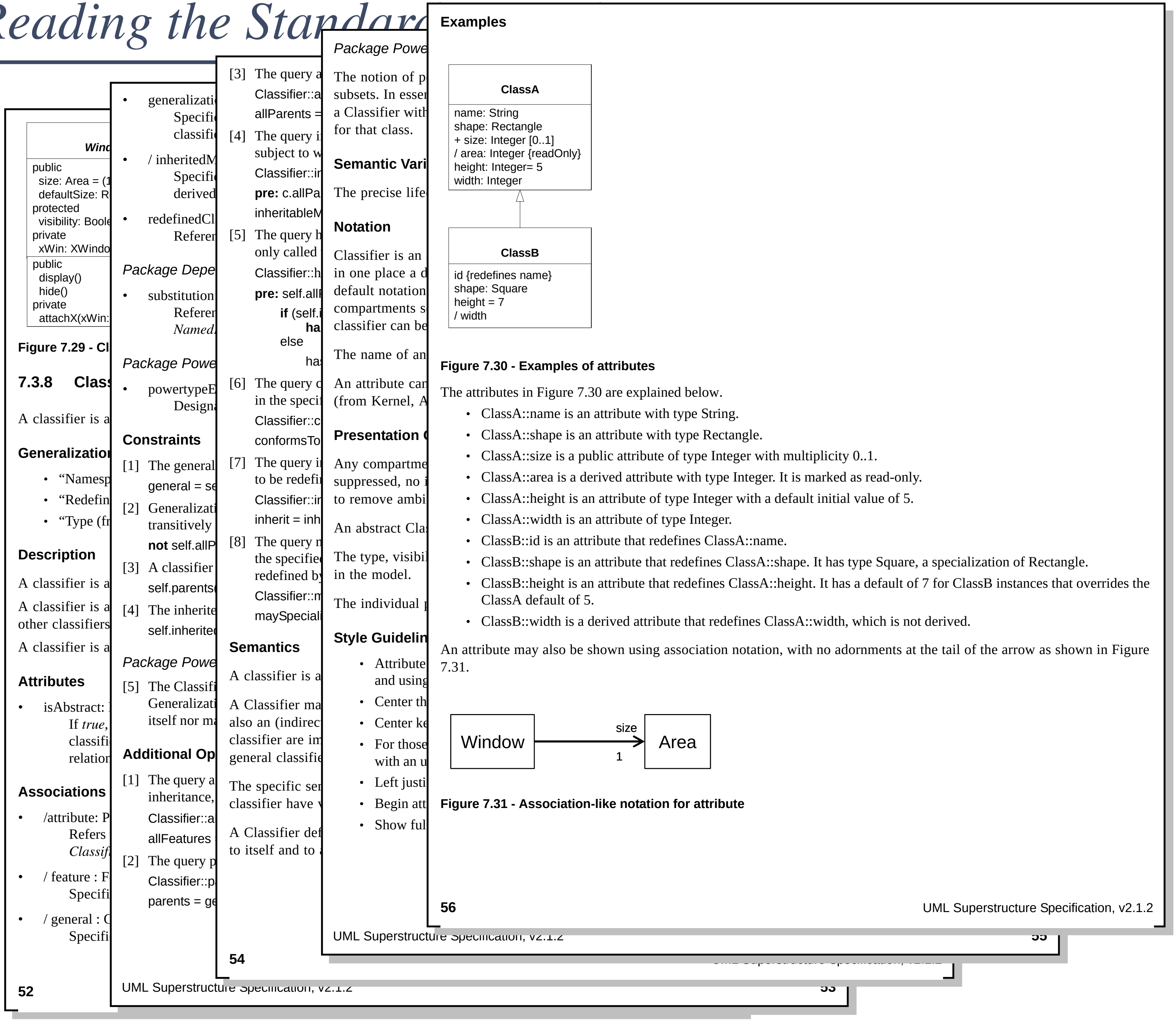
The individual properties of an attribute can be shown in columns rather than as a continuous string.

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 Stan…



Reading the Standar

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.

Meta Object Facility (MOF)

Open Questions...

- Now you've been "**tricked**".
  - 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).

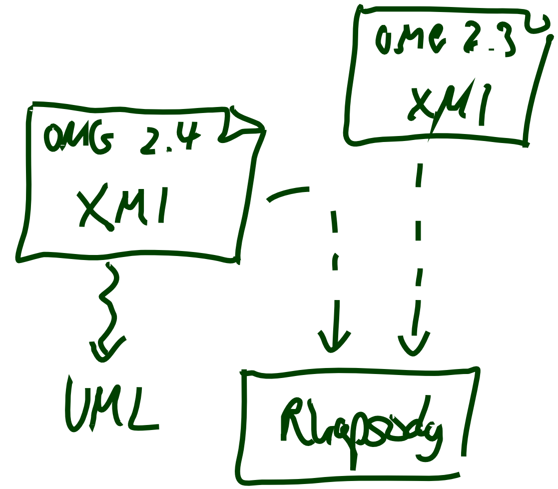
- So: things on meta level
  - M0 are object diagrams/system states
  - M1 are **words of the language UML**
  - M2 are **words of the language MOF**
  - M3 are **words of the language** ... MOF ?

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.

- 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. Buschermöhle and Oelerink (2008).

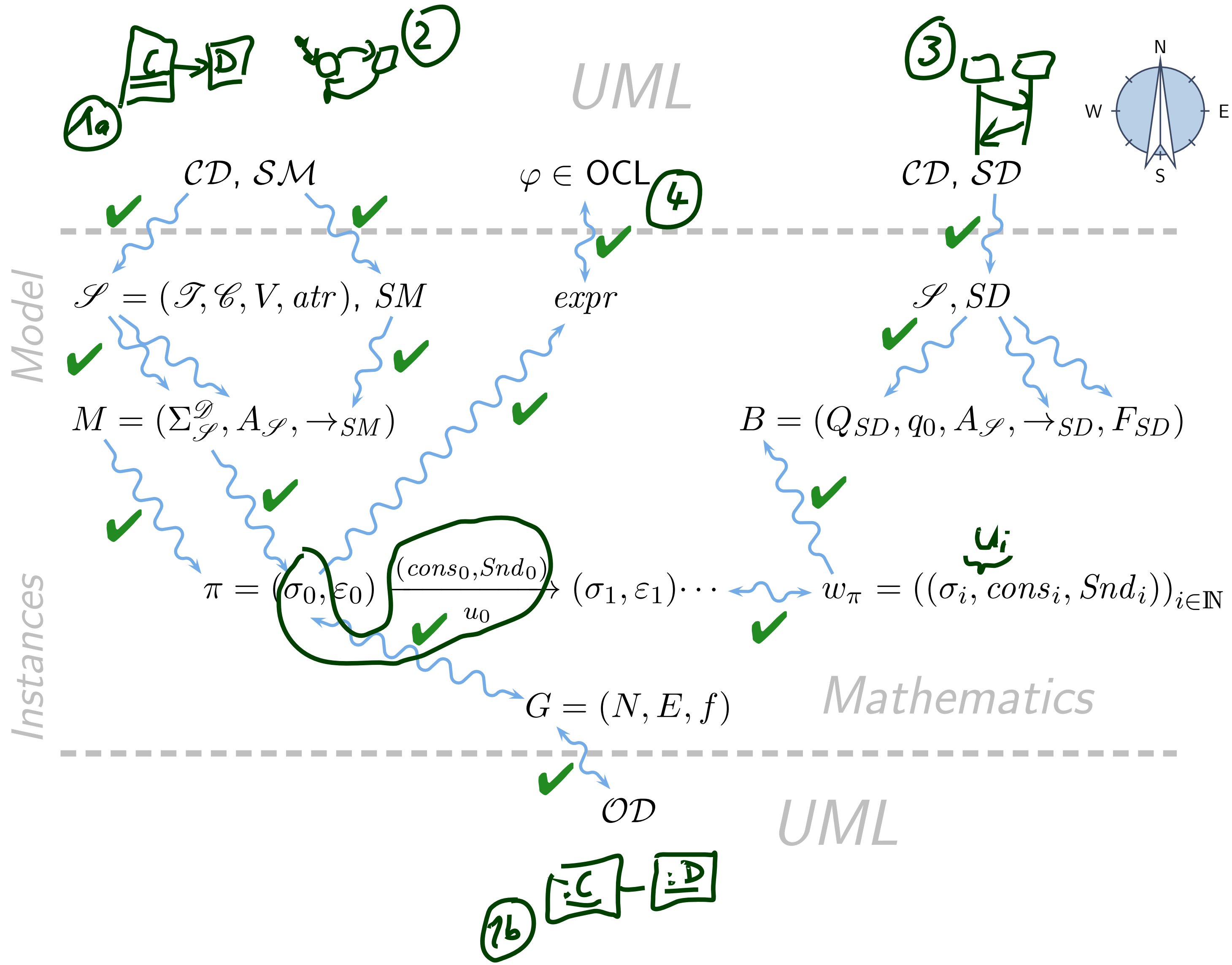
Benefits



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

And That's It!

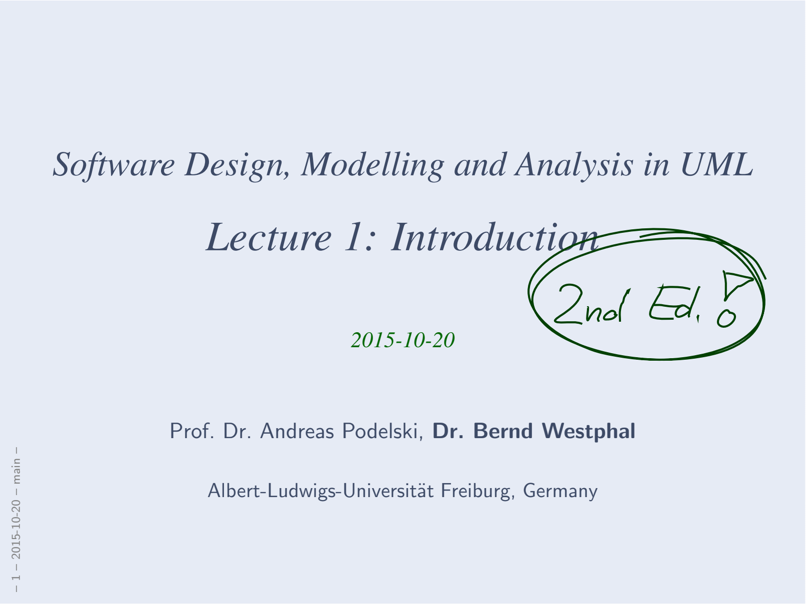
The Map



UML
1a
2
3
4
N
W
E
S
CD, SM
φ ∈ OCL
CD, SD
Model
𝒮 = (𝒯, 𝒞, V, atr), SM
expr
𝒮, SD
M = (Σ^𝒟_𝒮, A_𝒮, →_SM)
B = (Q_SD, q_0, A_𝒮, →_SD, F_SD)
Instances
π = (σ_0, ε_0) —(cons_0, Snd_0) / u_0→ (σ_1, ε_1) ···
w_π = ((σ_i, cons_i, Snd_i))_{i∈ℕ}
u_i
G = (N, E, f)
Mathematics
OD
UML
1b

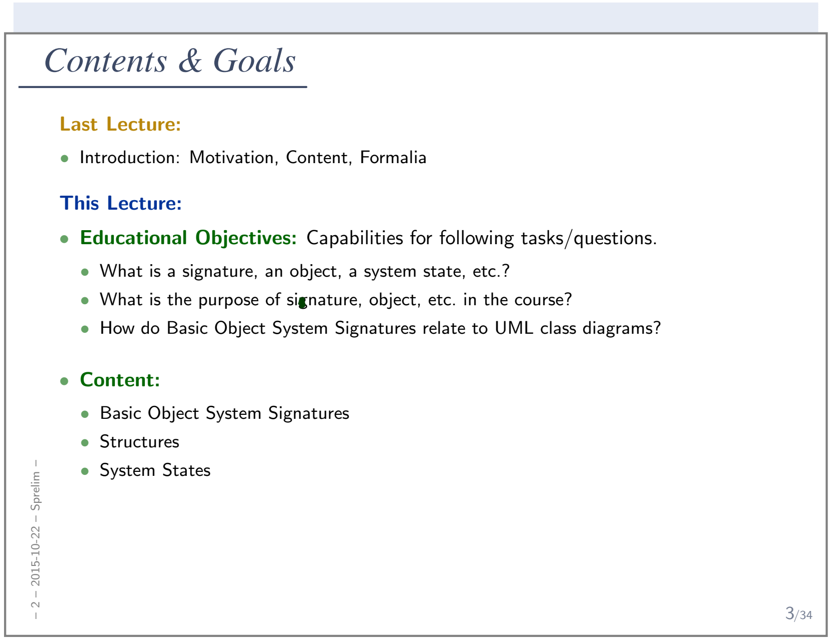
Content

- **Lecture 1**: Introduction



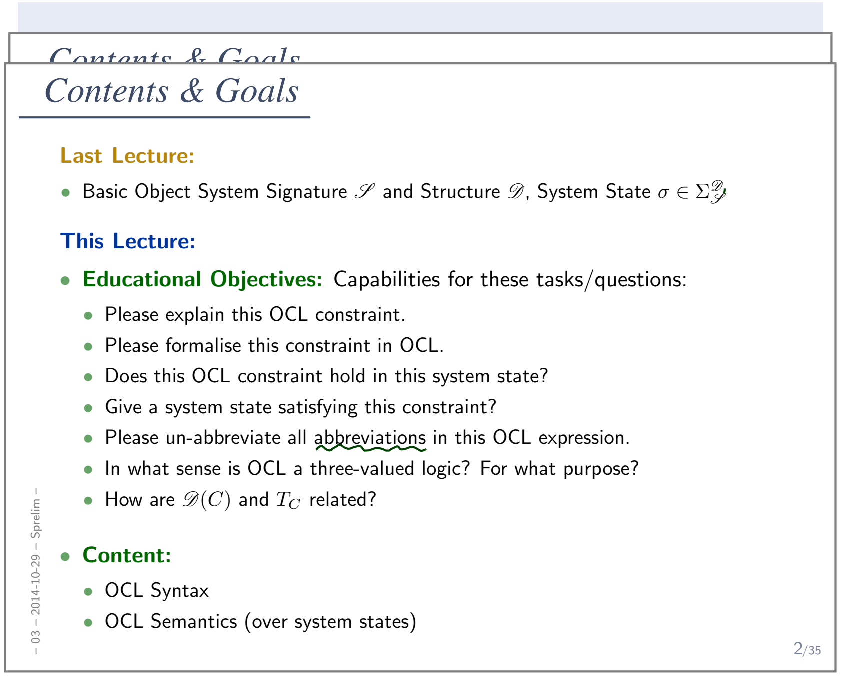
Content

- **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:

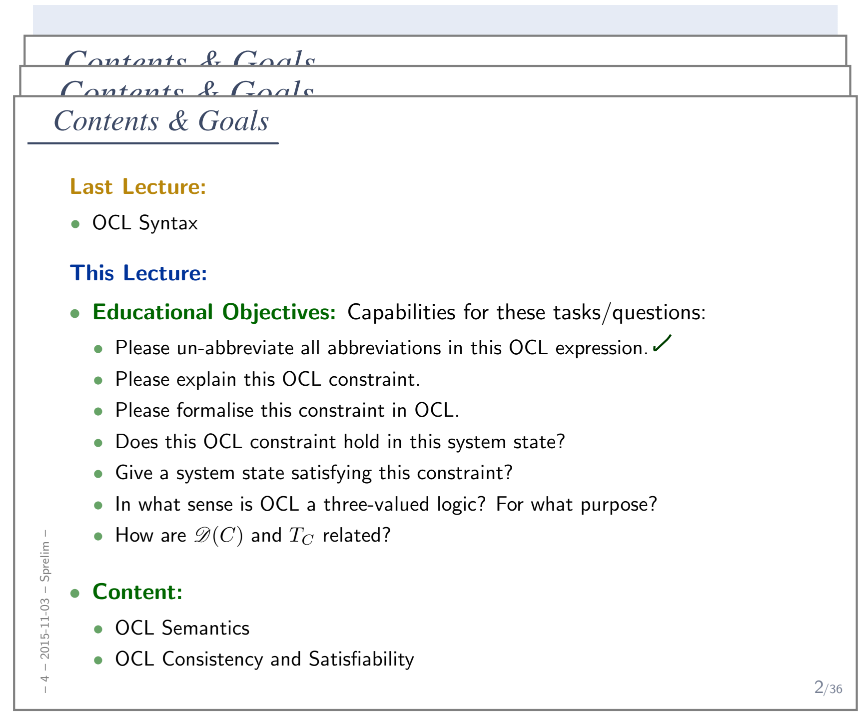
- **Educational Objectives:** Capabilities for these tasks/questions:
  - Please explain this OCL constraint.
  - Please formalise this constraint in OCL.
  - Does this OCL constraint hold in this system state?
  - Give a system state satisfying this constraint?
  - Please un-abbreviate all abbreviations 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:

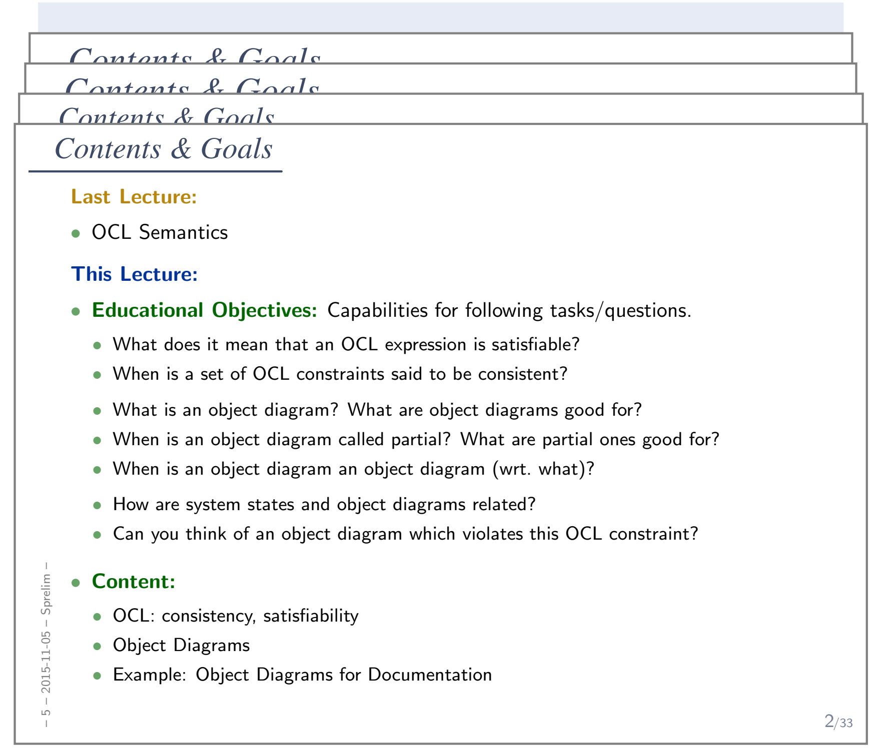
- **Educational Objectives:** Capabilities for these tasks/questions:
  - Please un-abbreviate all abbreviations in this OCL expression. ✓
  - Please explain this OCL constraint.
  - Please formalise this constraint in OCL.
  - Does this OCL constraint hold in this system state?
  - Give a system state 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

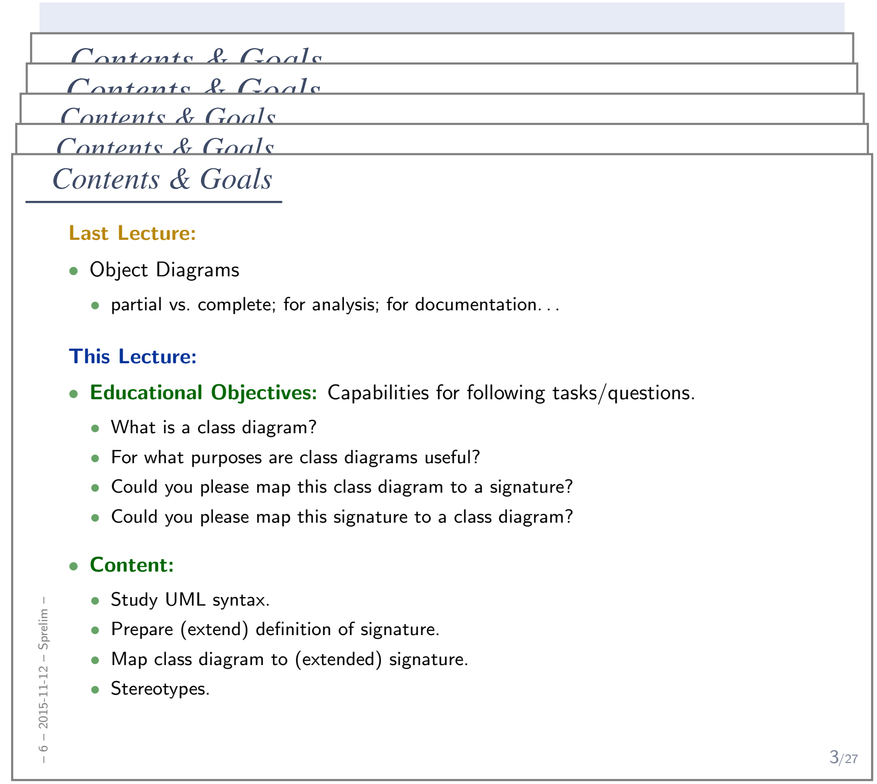
Content

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



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

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Last Lecture:

- Representing class diagrams as (extended) signatures — for the moment without associations: later.

This Lecture:

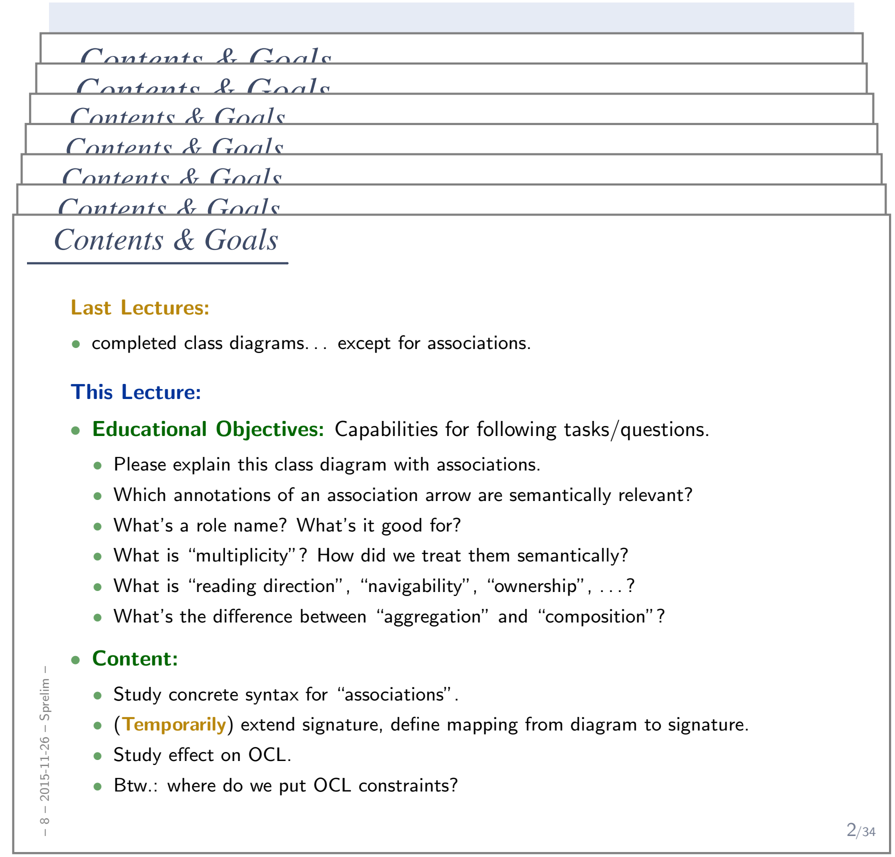
- **Educational Objectives:** Capabilities for following tasks/questions.
  - Could you please map this class diagram to a signature?
  - What if things are missing?
  - Could you please map this signature to a class diagram?
  - What is the semantics of 'abstract'?
  - What is visibility good for?
- **Content:**
  - Map class diagram to (extended) signature cont'd.
  - Stereotypes – for documentation.
  - Visibility as an extension of well-typedness.

– 7 – 2015-11-17 – Sprelim –

2/23

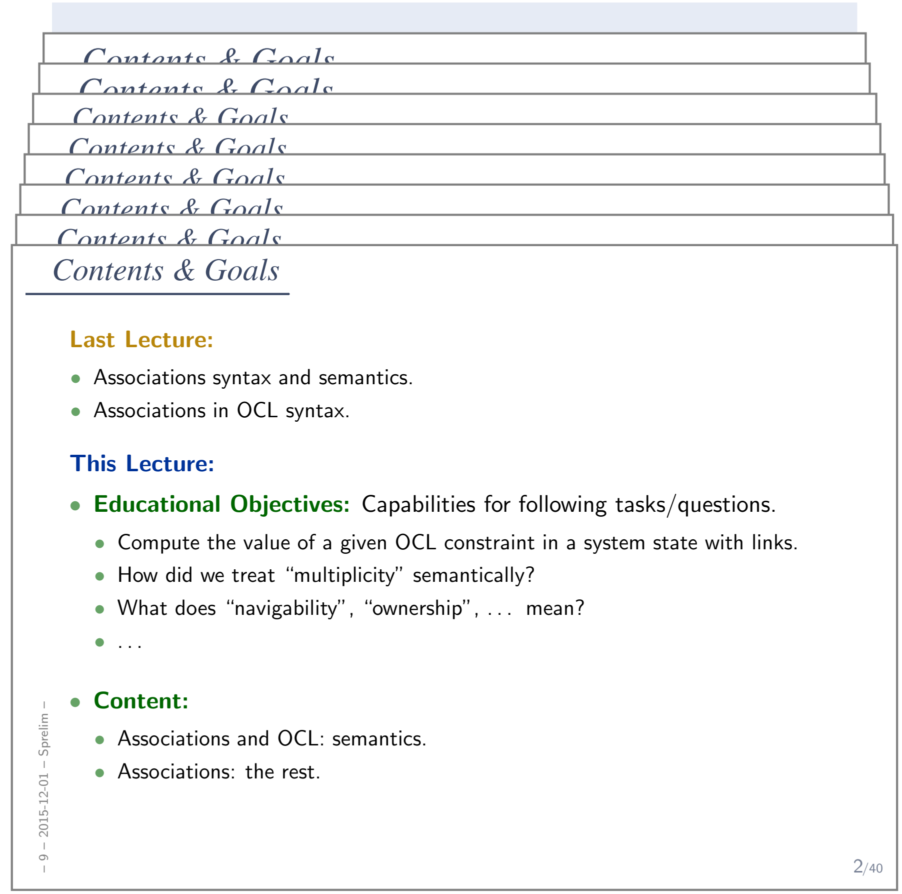
Content

- **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



Contents & Goals

Last Lecture:

- Associations syntax and semantics.
- Associations in OCL syntax.

This Lecture:

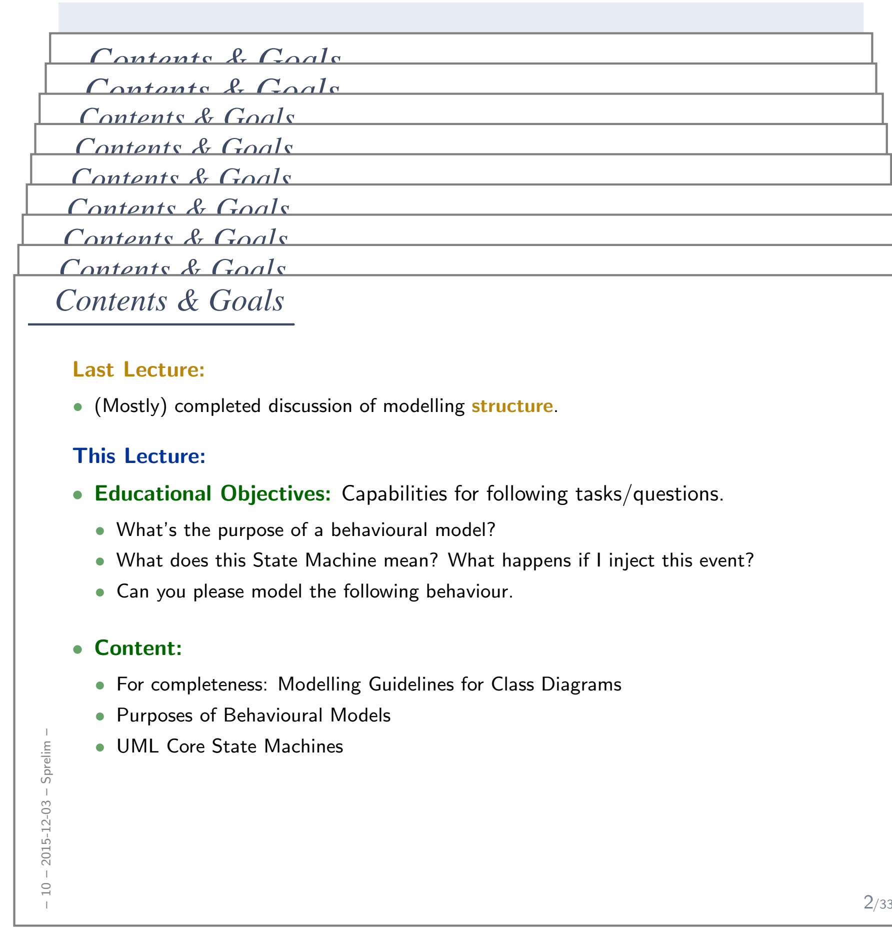
- **Educational Objectives:** Capabilities for following tasks/questions.
  - Compute the value of a given OCL constraint in a system state with links.
  - How did we treat "multiplicity" semantically?
  - What does "navigability", "ownership", . . . mean?
  - . . .
- **Content:**
  - Associations and OCL: semantics.
  - Associations: the rest.

– 9 – 2015-12-01 – Sprelim –

2/40

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



Contents & Goals

Last Lecture:

- (Mostly) completed discussion of modelling **structure**.

This Lecture:

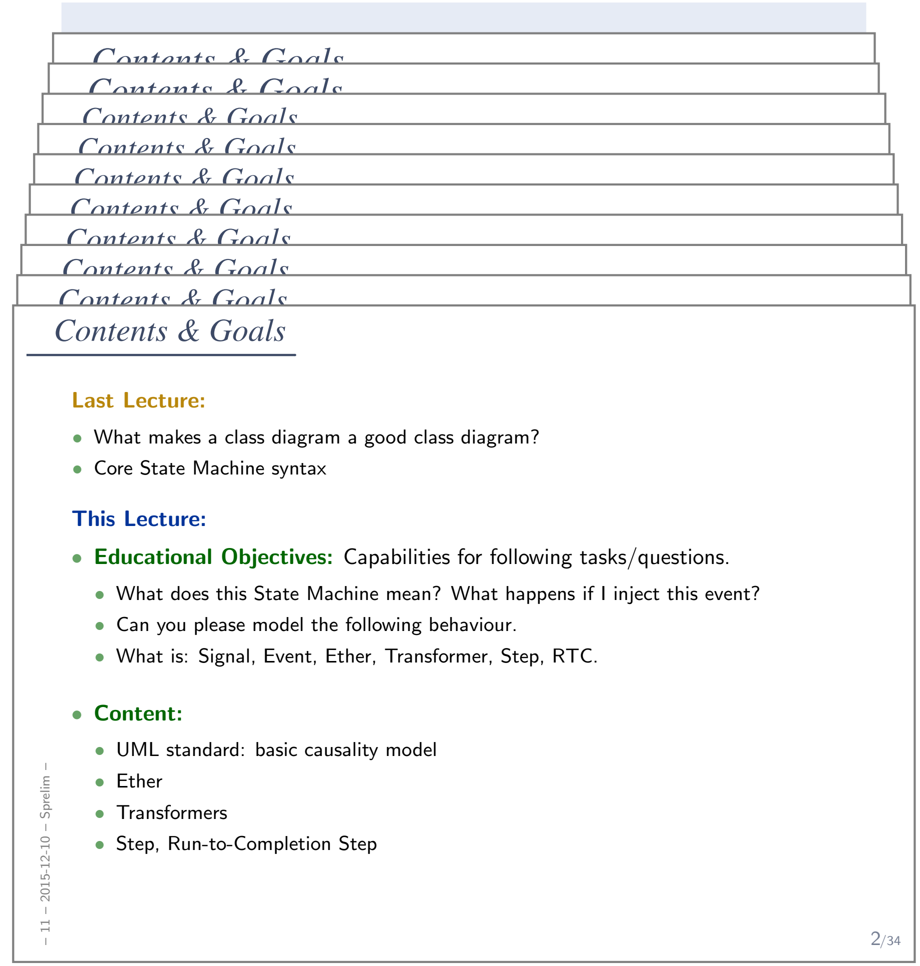
- **Educational Objectives:** Capabilities for following tasks/questions.
  - What's the purpose of a behavioural model?
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.
- **Content:**
  - For completeness: Modelling Guidelines for Class Diagrams
  - Purposes of Behavioural Models
  - UML Core State Machines

– 10 – 2015-12-03 – 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
- **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



Contents & Goals

Last Lecture:

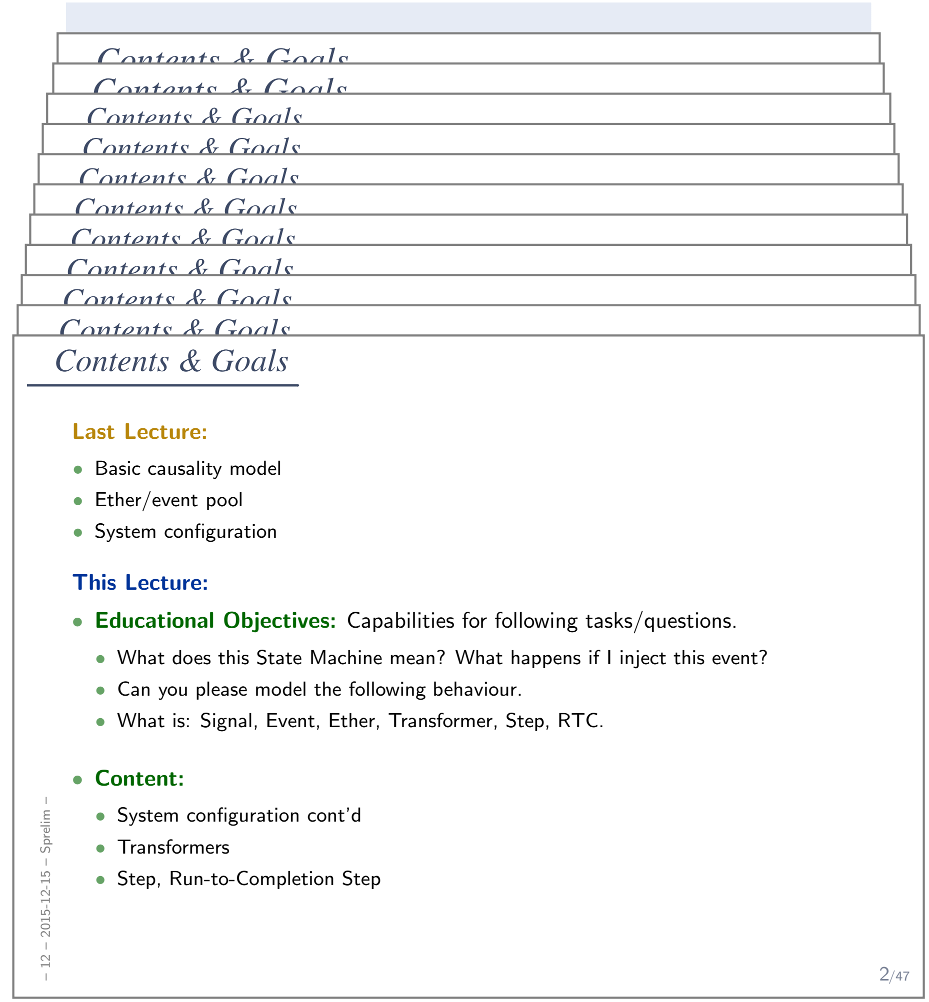
- What makes a class diagram a good class diagram?
- Core State Machine syntax

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.
  - What is: Signal, Event, Ether, Transformer, Step, RTC.
- **Content:**
  - UML standard: basic causality model
  - Ether
  - Transformers
  - Step, Run-to-Completion Step

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


Contents & Goals

Last Lecture:

- Basic causality model
- Ether/event pool
- System configuration

This Lecture:

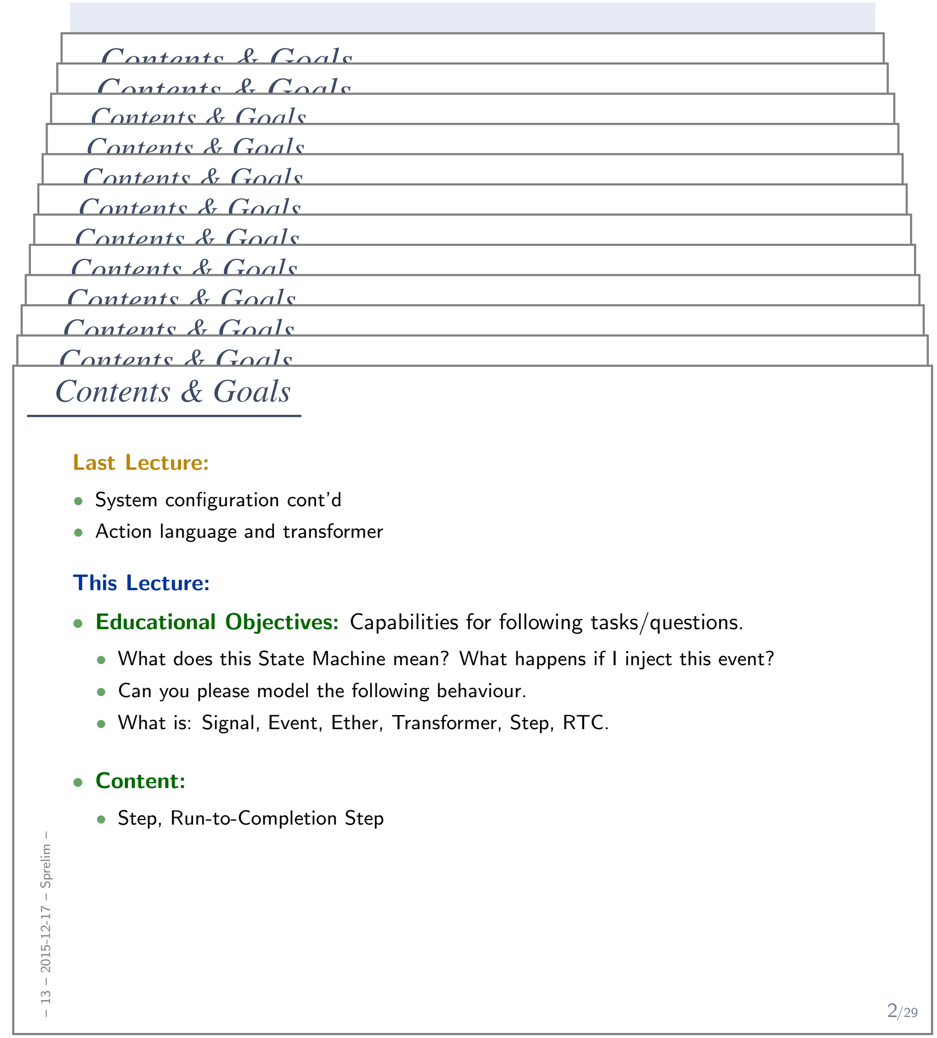
- **Educational Objectives:** Capabilities for following tasks/questions.
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.
  - What is: Signal, Event, Ether, Transformer, Step, RTC.
- **Content:**
  - System configuration cont'd
  - Transformers
  - Step, Run-to-Completion Step

– 12 – 2015-12-15 – Sprelim –

2/47

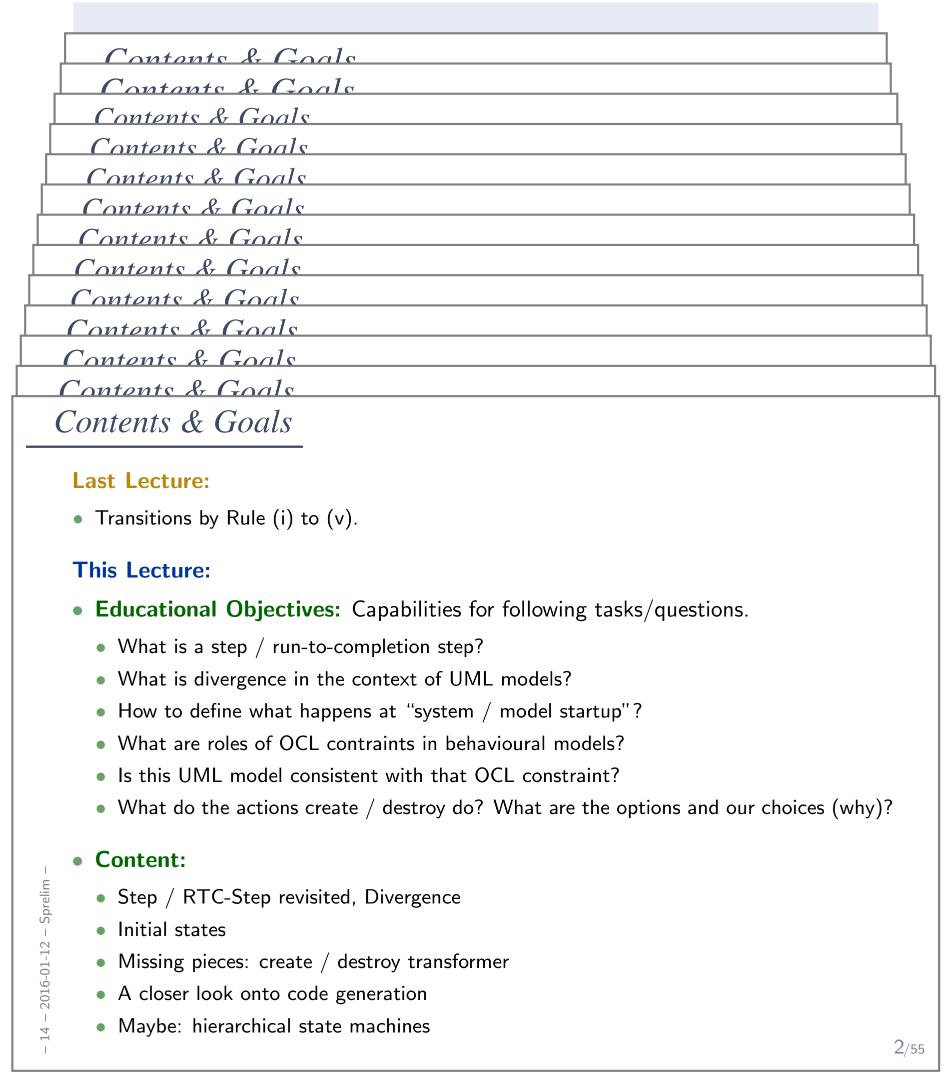
Content

- **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**: Core State Machines IV


Contents & Goals

Last Lecture:

- Transitions by Rule (i) to (v).

This Lecture:

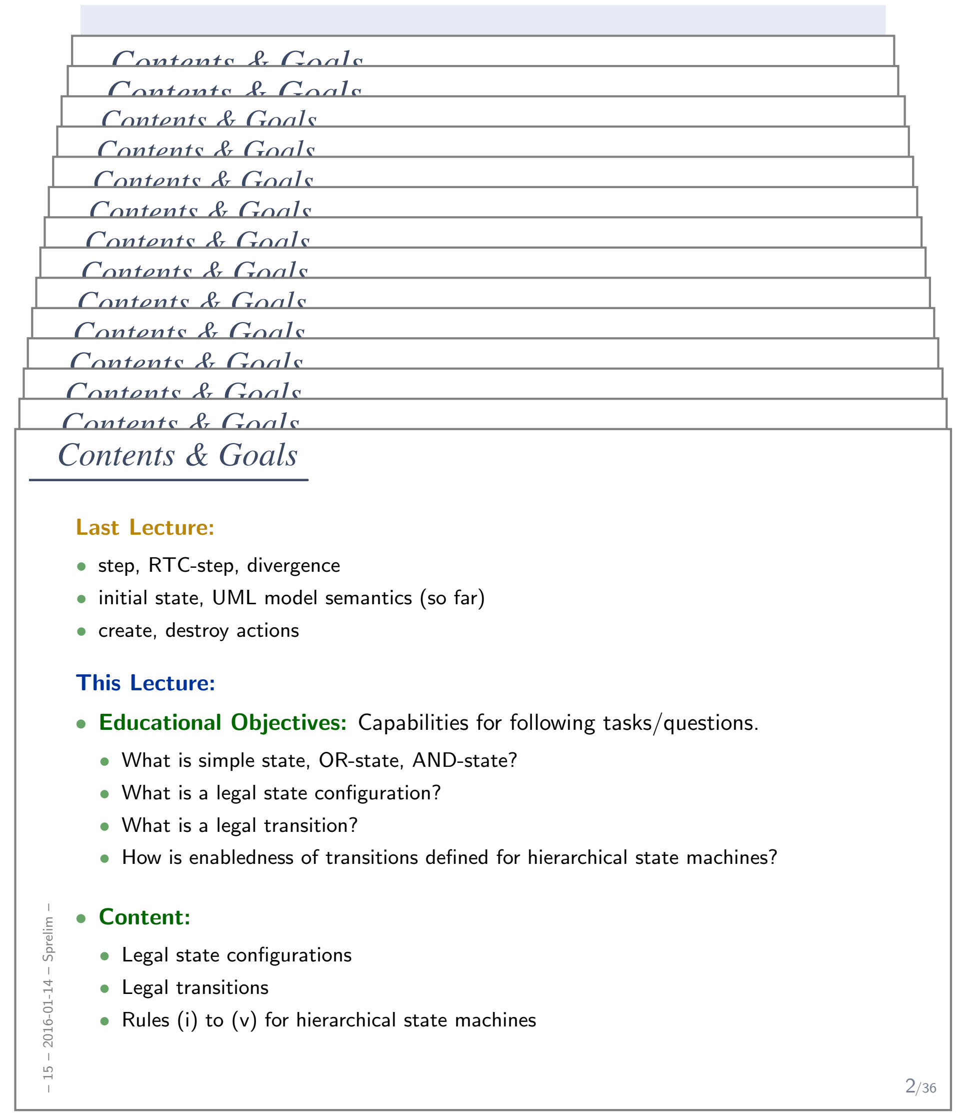
- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is a step / run-to-completion step?
  - What is divergence in the context of UML models?
  - How to define what happens at "system / model startup"?
  - What are roles of OCL contraints in behavioural models?
  - Is this UML model consistent with that OCL constraint?
  - What do the actions create / destroy do? What are the options and our choices (why)?
- **Content:**
  - Step / RTC-Step revisited, Divergence
  - Initial states
  - Missing pieces: create / destroy transformer
  - A closer look onto code generation
  - Maybe: hierarchical state machines

– 14 – 2016-01-12 – Sprelim –

2/55

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**: Core State Machines IV
- **Lecture 15**: Hierarchical State Machines I


Contents & Goals

Last Lecture:

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

This Lecture:

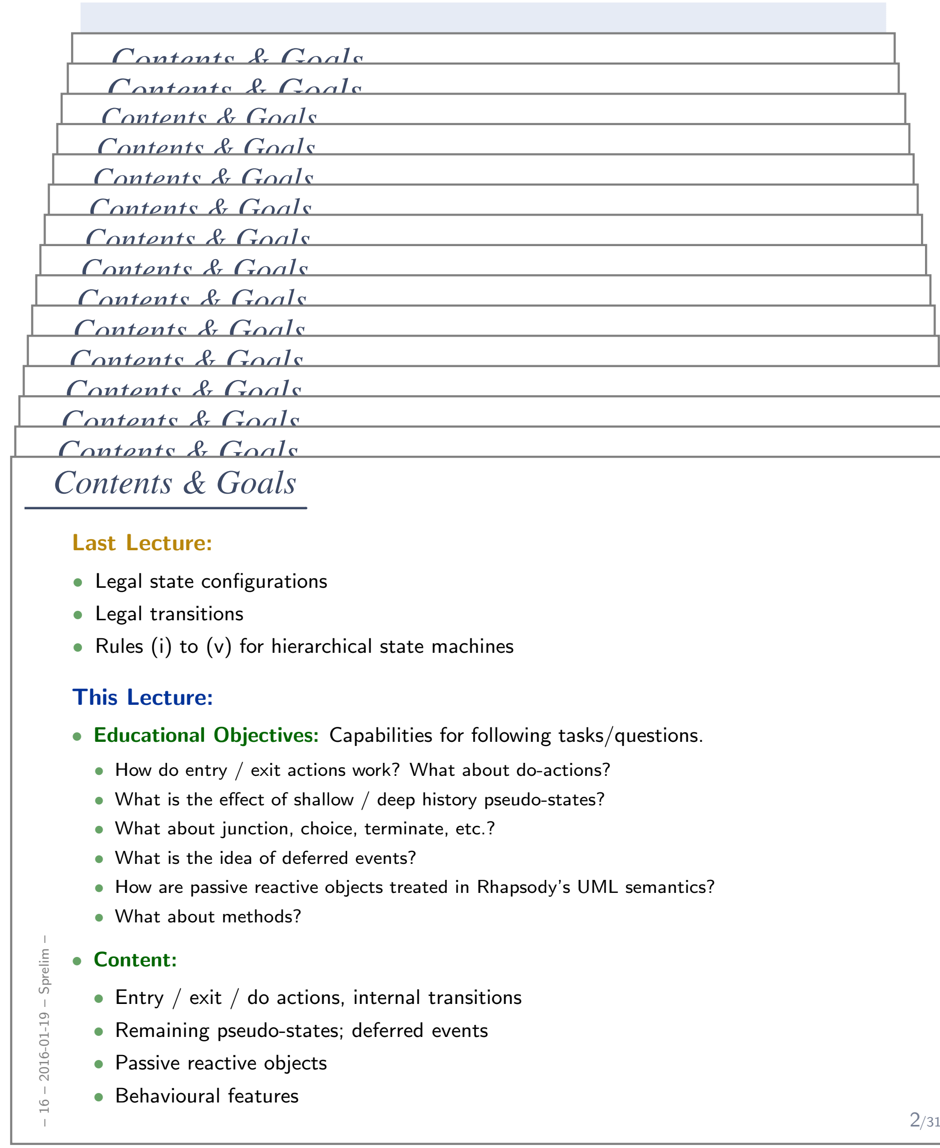
- Educational Objectives: Capabilities for following tasks/questions.
  - What is simple state, OR-state, AND-state?
  - What is a legal state configuration?
  - What is a legal transition?
  - How is enabledness of transitions defined for hierarchical state machines?
- Content:
  - Legal state configurations
  - 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**: Core State Machines IV
- **Lecture 15**: Hierarchical State Machines I
- **Lecture 16**: Hierarchical State Machines II


Contents & Goals

Last Lecture:

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

This Lecture:

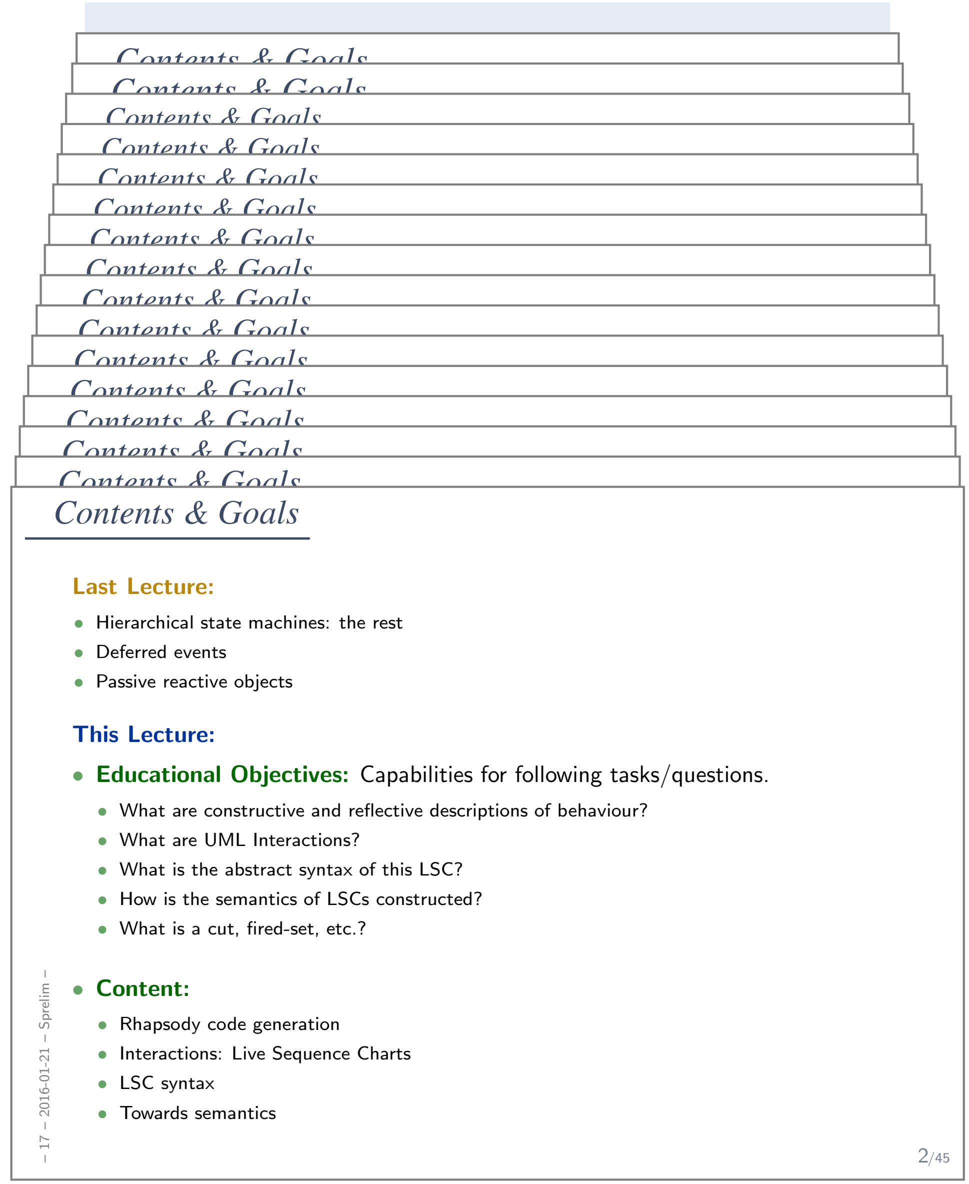
- **Educational Objectives:** Capabilities for following tasks/questions.
  - How do entry / exit actions work? What about do-actions?
  - What is the effect of shallow / deep history pseudo-states?
  - What about junction, choice, terminate, etc.?
  - What is the idea of deferred events?
  - How are 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

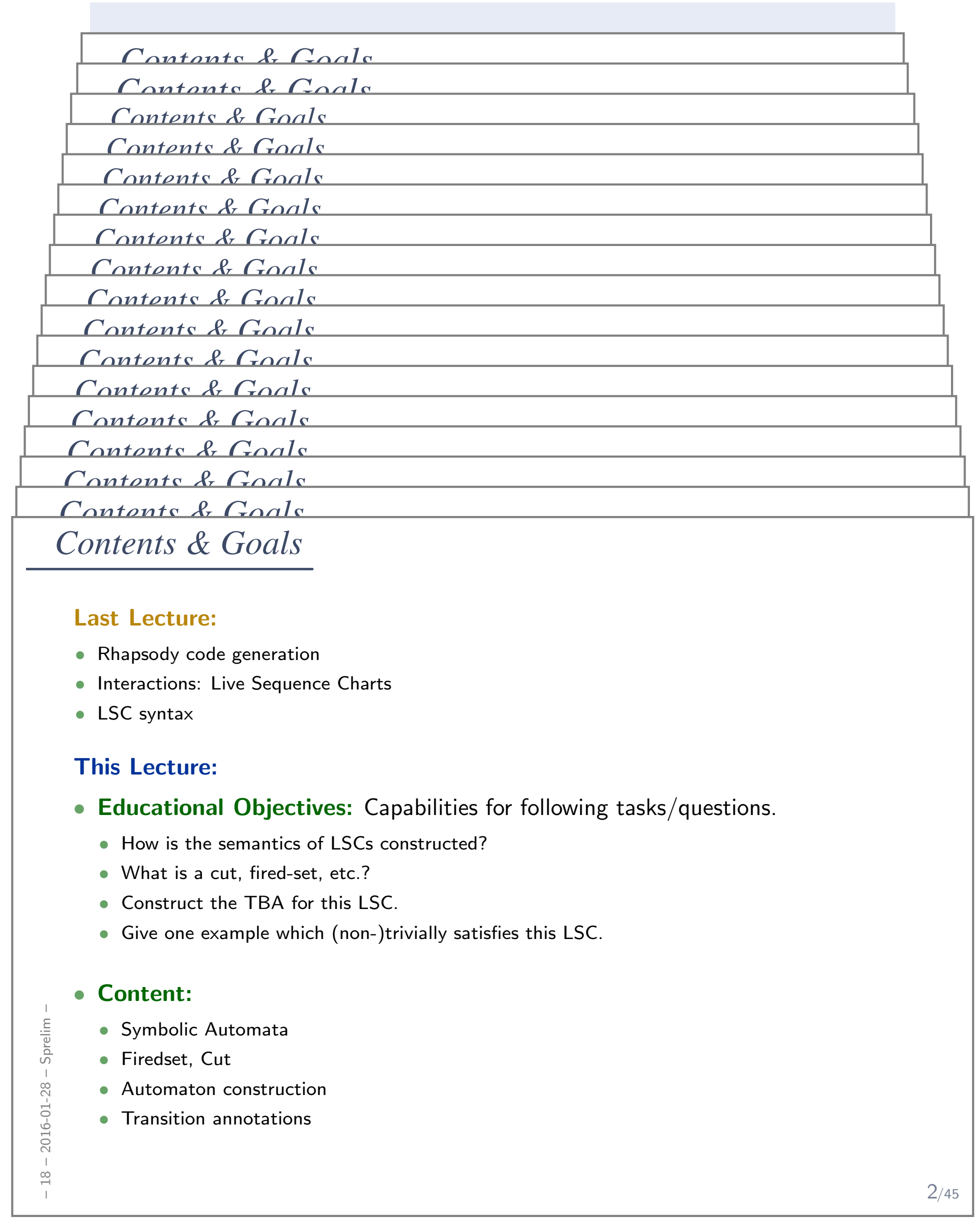
Content

- **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**: Core State Machines IV
- **Lecture 15**: Hierarchical State Machines I
- **Lecture 16**: Hierarchical State Machines II
- **Lecture 17**: Live Sequence Charts I



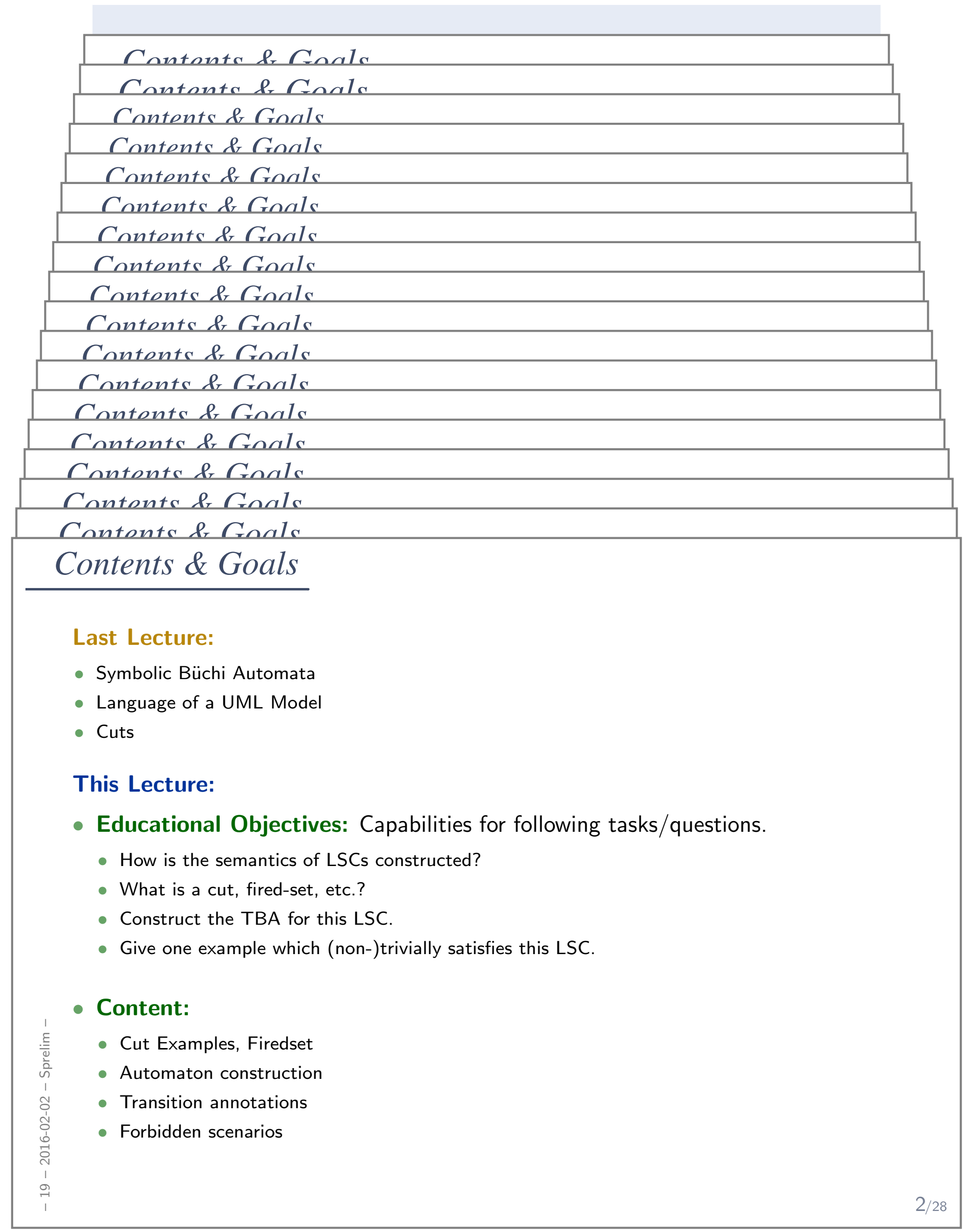
Content

- **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**: Core State Machines IV
- **Lecture 15**: Hierarchical State Machines I
- **Lecture 16**: Hierarchical State Machines II
- **Lecture 17**: Live Sequence Charts I
- **Lecture 18**: Live Sequence Charts 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
- **Lecture 14**: Core State Machines IV
- **Lecture 15**: Hierarchical State Machines I
- **Lecture 16**: Hierarchical State Machines II
- **Lecture 17**: Live Sequence Charts I
- **Lecture 18**: Live Sequence Charts II
- **Lecture 19**: Live Sequence Charts III



Contents & Goals

Last Lecture:

- Symbolic Büchi Automata
- Language of a UML Model
- Cuts

This Lecture:

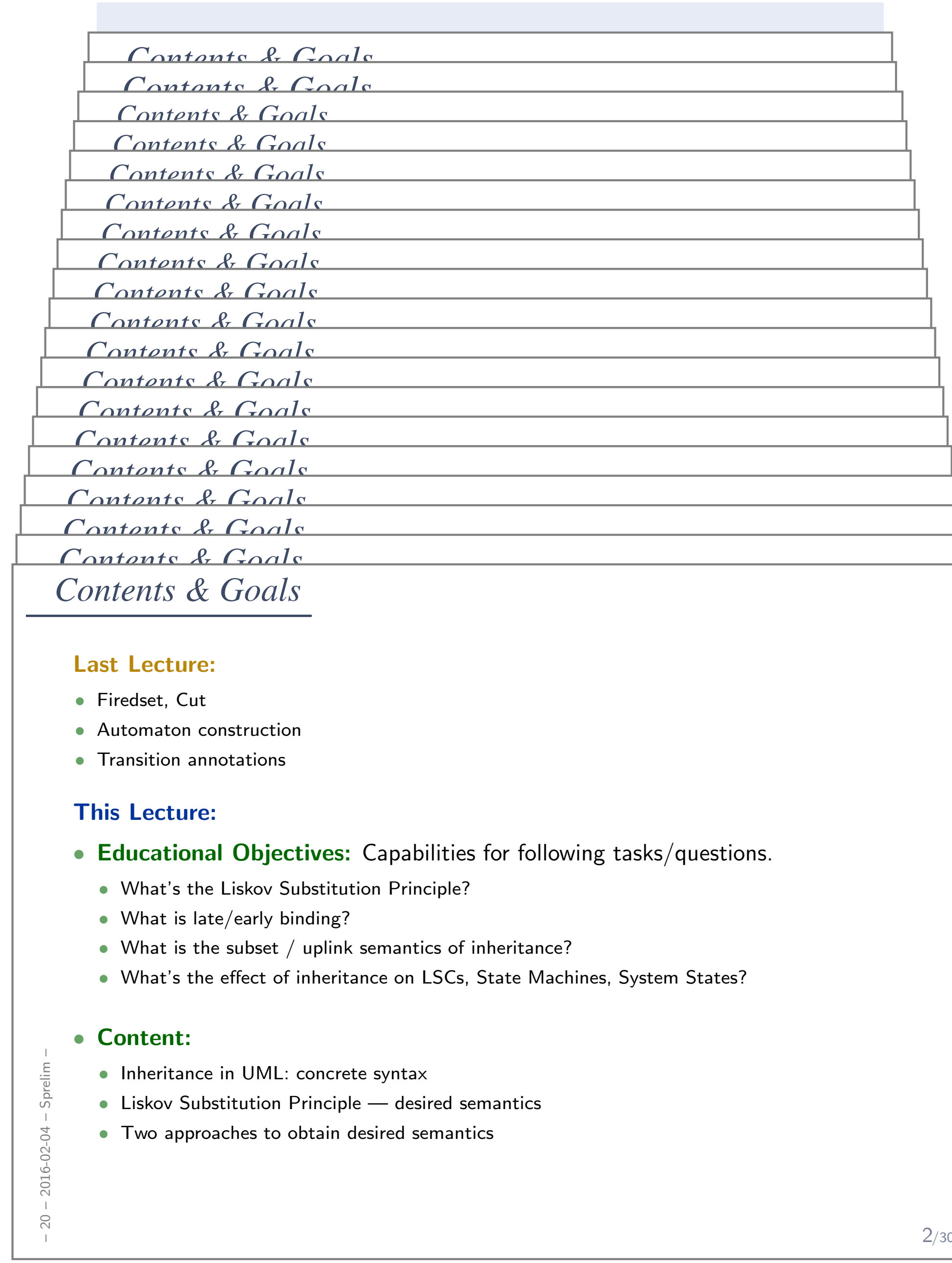
- **Educational Objectives:** Capabilities for following tasks/questions.
  - How is the semantics of LSCs constructed?
  - What is a cut, fired-set, etc.?
  - Construct the TBA for this LSC.
  - Give one example which (non-)trivially satisfies this LSC.
- **Content:**
  - Cut Examples, Firedset
  - Automaton construction
  - Transition annotations
  - Forbidden scenarios

– 19 – 2016-02-02 – Sprelim –

2/28

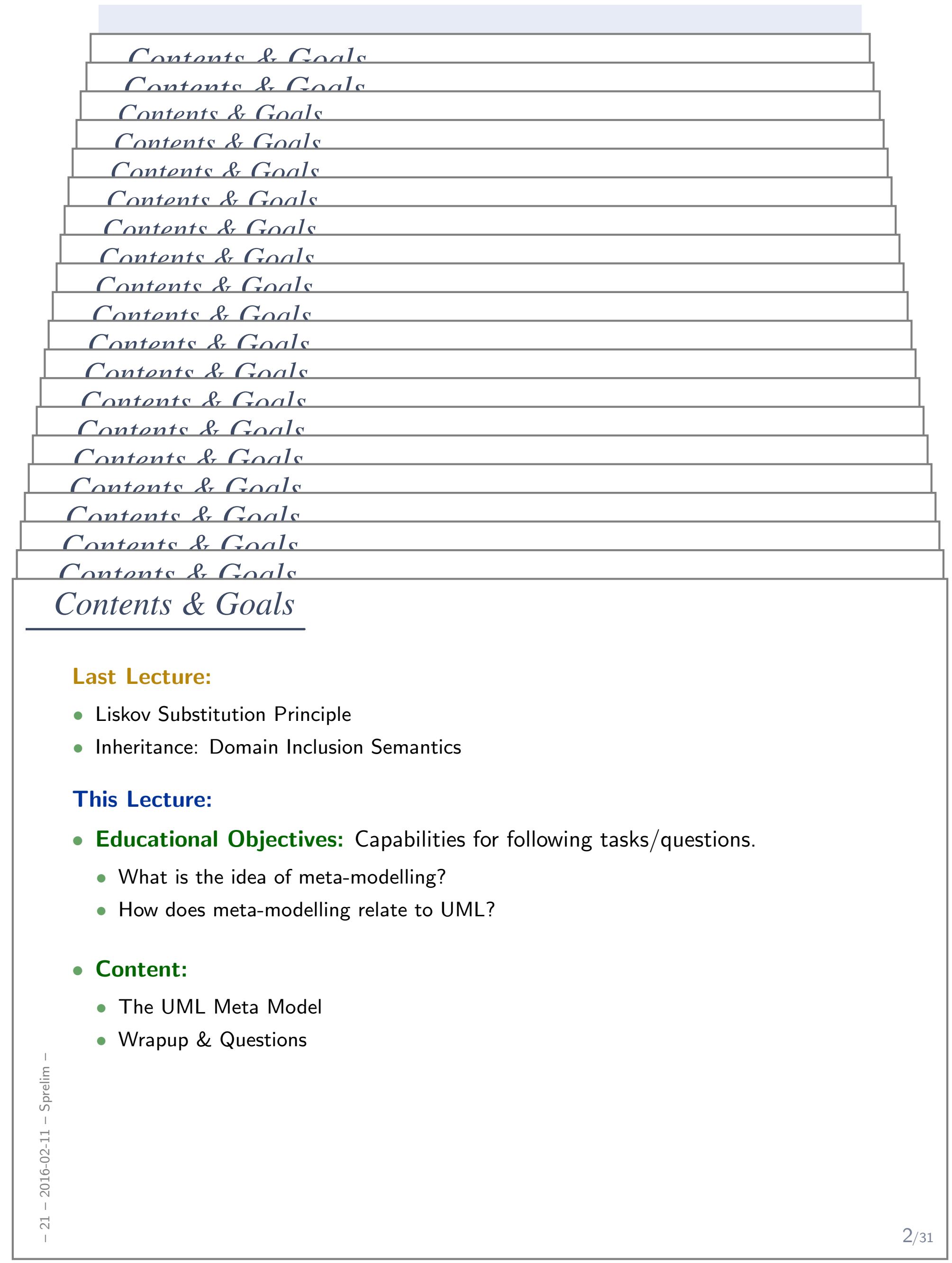
Content

- **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**: Core State Machines IV
- **Lecture 15**: Hierarchical State Machines I
- **Lecture 16**: Hierarchical State Machines II
- **Lecture 17**: Live Sequence Charts I
- **Lecture 18**: Live Sequence Charts II
- **Lecture 19**: Live Sequence Charts III
- **Lecture 20**: 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**: Core State Machines IV
- **Lecture 15**: Hierarchical State Machines I
- **Lecture 16**: Hierarchical State Machines II
- **Lecture 17**: Live Sequence Charts I
- **Lecture 18**: Live Sequence Charts II
- **Lecture 19**: Live Sequence Charts III
- **Lecture 20**: Inheritance
- **Lecture 21**: Meta-Modelling



References

References

Buschermöhle, R. and Oelerink, J. (2008). Rich meta object facility. In *Proc. 1st IEEE Int'l workshop UML and Formal Methods*.

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.