

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

-22-2017-02-07-main-

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?**

-22-2017-02-07-Content-

Inheritance

Abstract Syntax

A **signature with inheritance** is a tuple

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

where

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



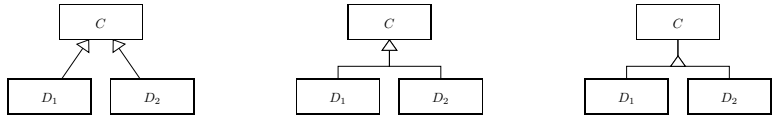
In the following (for simplicity), we assume that all attribute (method) names are of the form $C::v$ and $C::f$ for some $C \in \mathcal{C} \cup \mathcal{E}$ (“**fully qualified names**”).

Read $C \triangleleft D$ as...

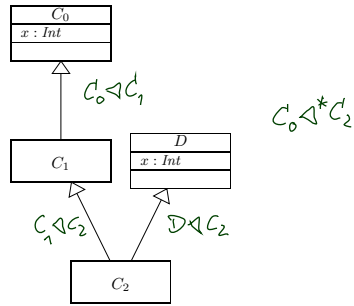
- D **inherits** from C ,
- C is a **generalisation** of D ,
- D is a **specialisation** of C ,
- C is a **super-class** of D ,
- D is a **sub-class** of C ,
- ...

Inheritance: Concrete Syntax

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



Mapping Concrete to Abstract Syntax by Example:



-21-2007-02-07-Syntax-

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).

-21-2007-02-07-Sub-

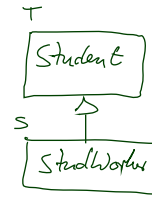
Desired Semantics of Specialisation: Subtyping

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 ."

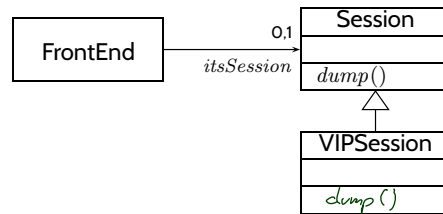


In other words: Fischer and Wehrheim (2000)

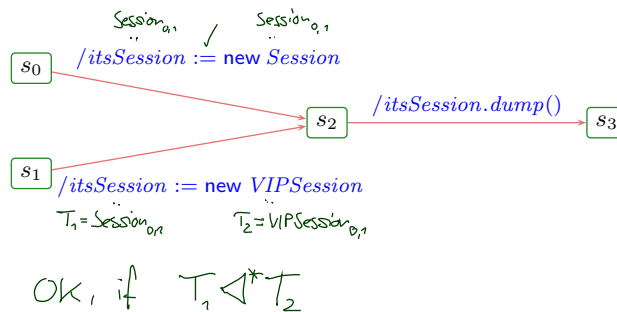
"An instance of the **sub-type** shall be usable
 whenever an instance of the supertype was expected,
without a client being able to tell the difference."

-21-2007-02-07-Step-

Static Sub-Typing



In FrontEnd's state machine:



-21-2007-02-07-Substenance-

Domain Inclusion vs. Uplink Semantics

-21-2007-02-07 - Semantics -

System States with Inheritance

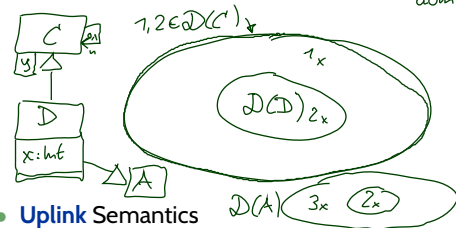
Wanted: a formal representation of “if $C \triangleleft^* D$ then D ‘is a’ C ”, that is,

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

Two approaches to semantics:

(not disjoint any more)

- **Domain-inclusion Semantics**



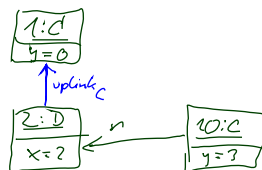
for $u \in D(C)$:
 $\text{dom } \sigma(u) = \bigcup_{C_0 \triangleleft^* C} \text{attrs } C_0$

$1: C$
 $y=0$

$2: D$
 $y=1$
 $x=2$

(more theoretical)

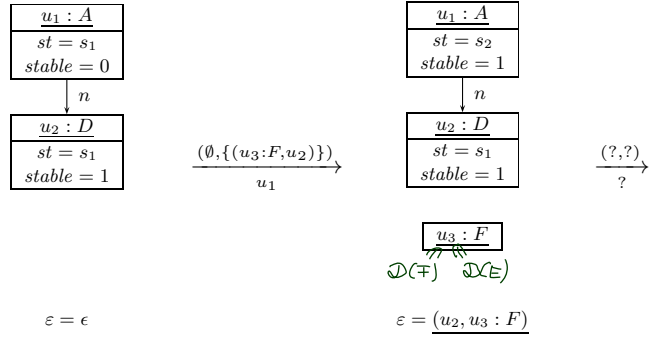
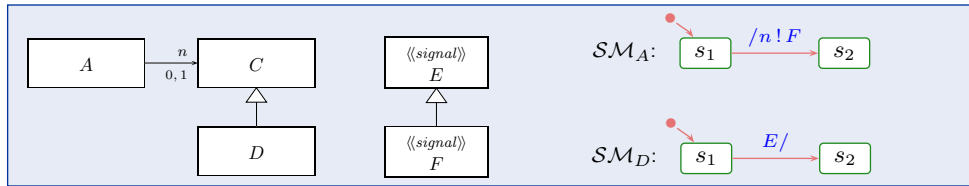
- **Uplink Semantics**



(more technical)

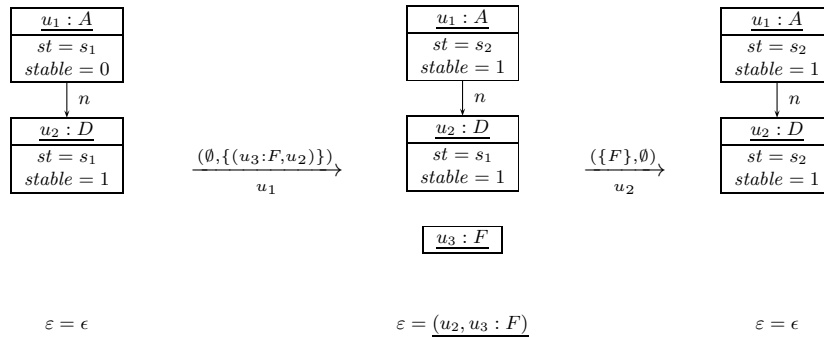
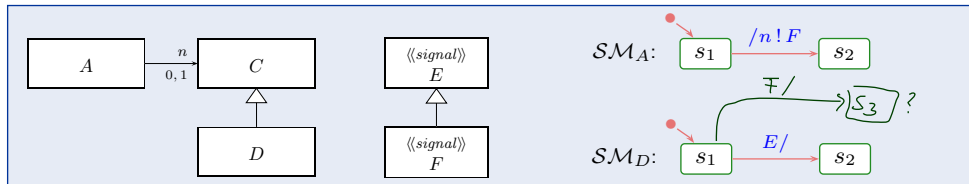
-21-2007-02-07 - Semantics -

Inheritance and State-Machines: Example



-21- 2007-02-07 - Spherasence -

Inheritance and State-Machines: Example



-21- 2007-02-07 - Spherasence -

(ii) Dispatch

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

if

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

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

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

$$\text{e.g. } \boxed{s_s} \xrightarrow{E[\text{params}_E, x \mapsto x]} \boxed{s_c}$$

and

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

$$\begin{aligned} & (\sigma'', \varepsilon') \in t_{act}[u](\tilde{\sigma}, \varepsilon \ominus u_E), \\ \sigma' = & (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset]) \left| \overbrace{\mathcal{D}(\varepsilon) \setminus \{u_E\}}^{\text{remove } u_E} \right. \end{aligned}$$

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

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

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

21/32

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 .”

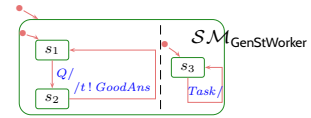
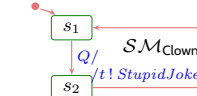
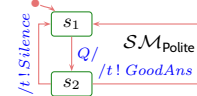
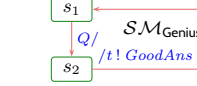
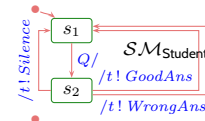
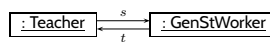
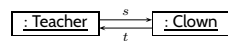
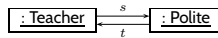
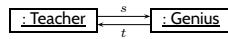
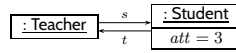
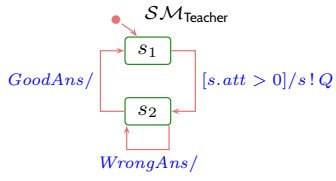
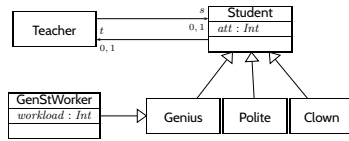
In other words: Fischer and Wehrheim (2000)

“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



-21-2007-02-07 - Sthelence -

Meta-Modelling: Idea

-21-2007-02-07 - main -

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**?

-21- 2007-02-07 - 5mm -

15/41

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**.

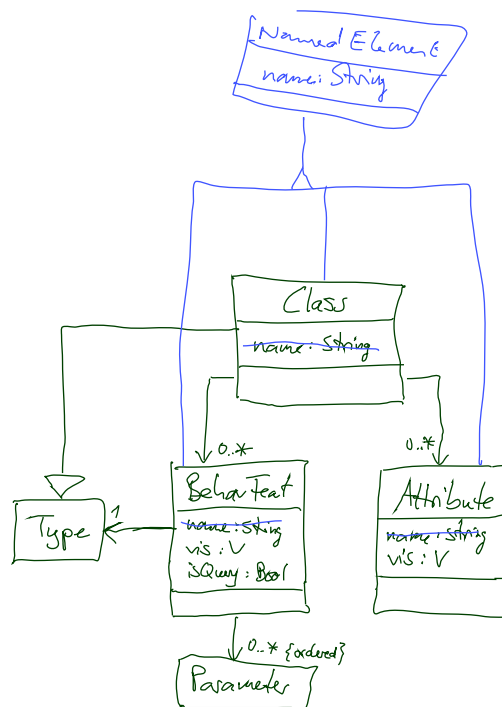
Each of the latter two has

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

Behavioural features in addition have

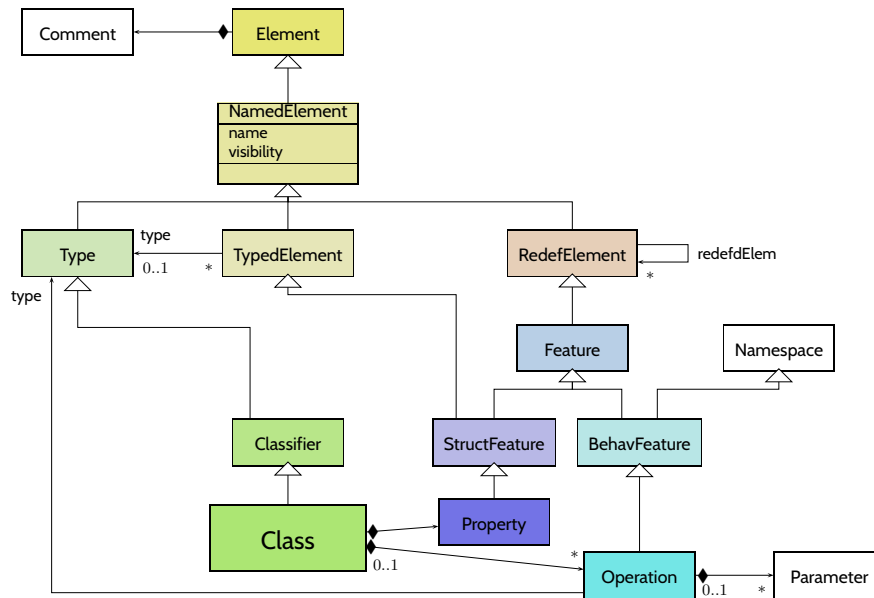
- a boolean attribute **isQuery**,
- any number of parameters,
- a return type.

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



-21- 2007-02-07 - 5mm -

16/41

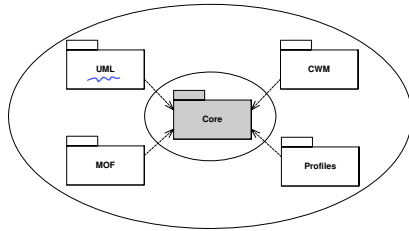
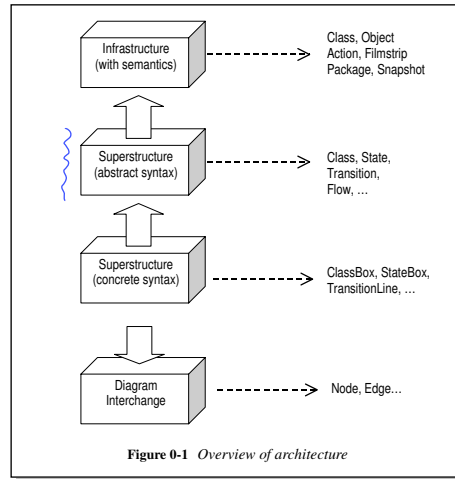


-20-2007-02-07-5mm-

The UML 2.x Standard Revisited

-20-2007-02-07-main-

- 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.



-22- 2007-02-07 - Sunilmm -

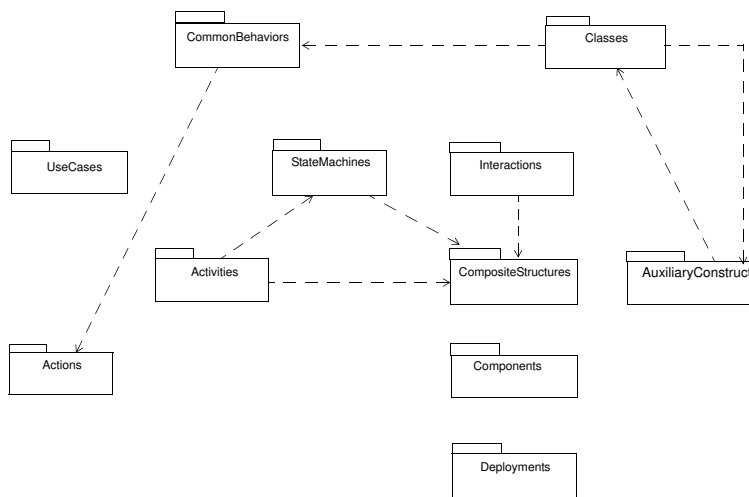
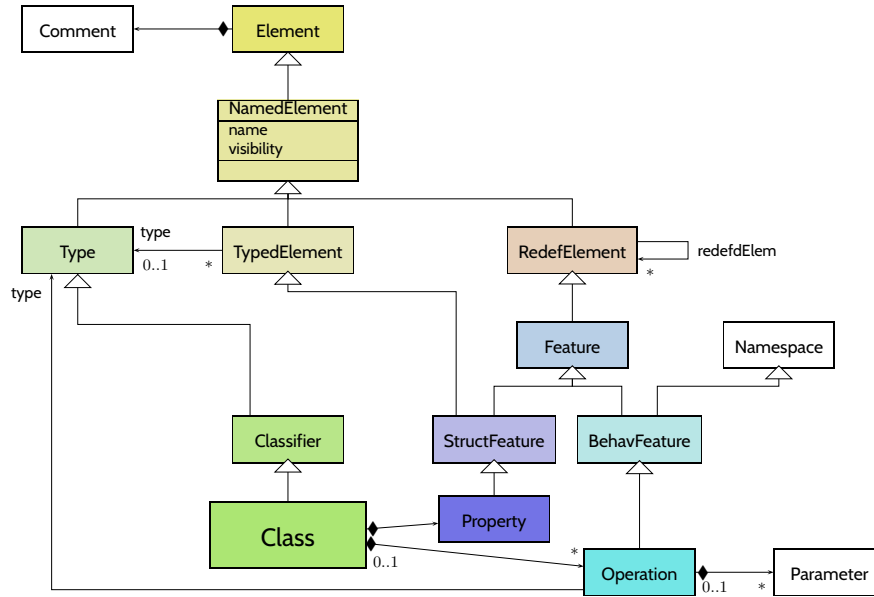


Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

-22- 2007-02-07 - Sunilmm -

Claim: Extract from UML 2.0 Standard



-22- 2007-02-07 - Sunilmm -

Classes (OMG, 2007b, 32)

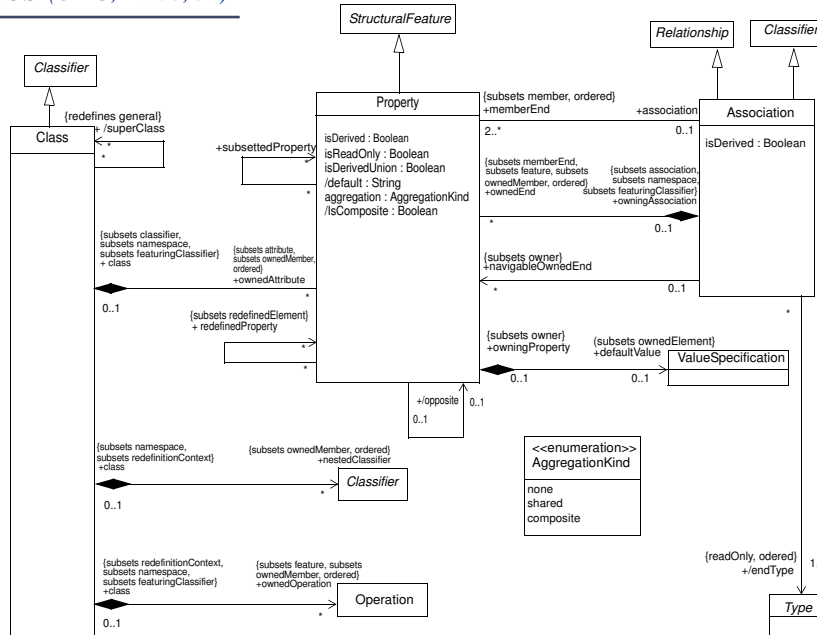


Figure 7.12 - Classes diagram of the Kernel package

-22- 2007-02-07 - Sunilmm -

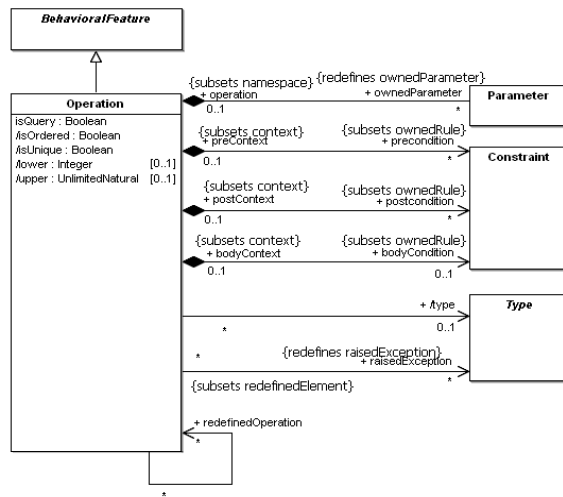


Figure 7.11 - Operations diagram of the Kernel package

-21- 2007-02-07 - Sunilmm -

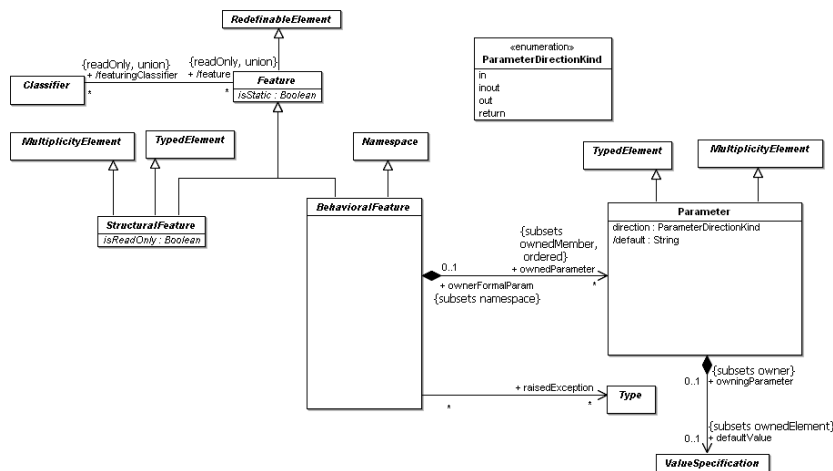
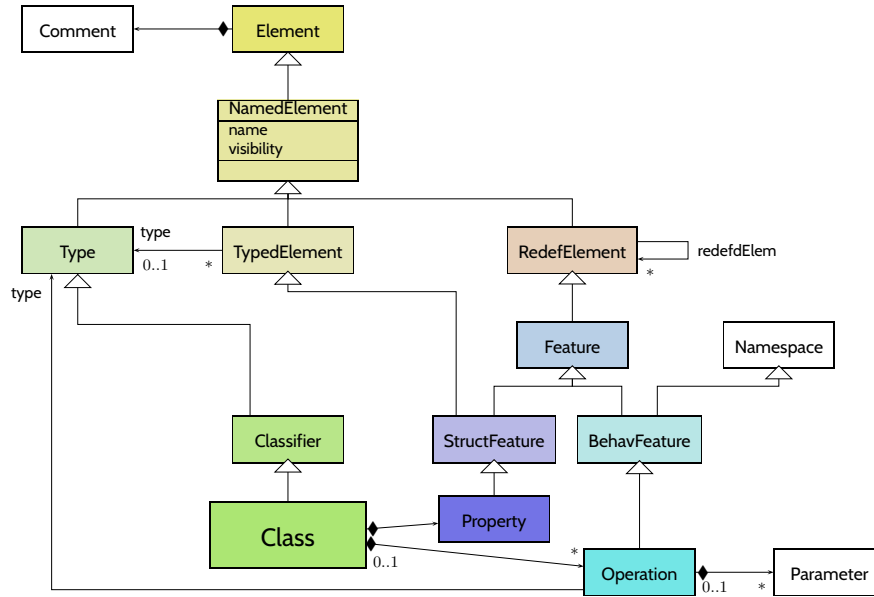


Figure 7.10 - Features diagram of the Kernel package

-21- 2007-02-07 - Sunilmm -

Claim: Extract from UML 2.0 Standard



-22- 2007-02-07 - Sunilmm -

Classifiers (OMG, 2007b, 29)

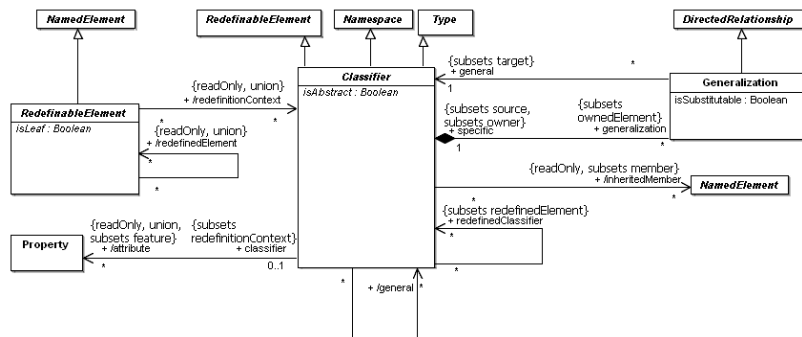


Figure 7.9 - Classifiers diagram of the Kernel package

-22- 2007-02-07 - Sunilmm -

Namespaces (OMG, 2007b, 26)

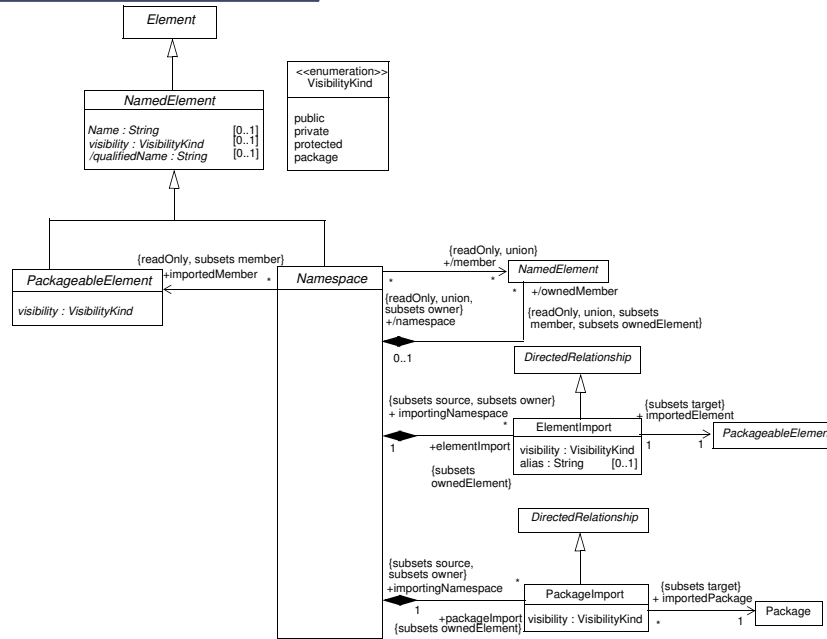


Figure 7.4 - Namespaces diagram of the Kernel package

-22- 2007-02-07 - Sunilmm -

27/41

Root Diagram (OMG, 2007b, 25)

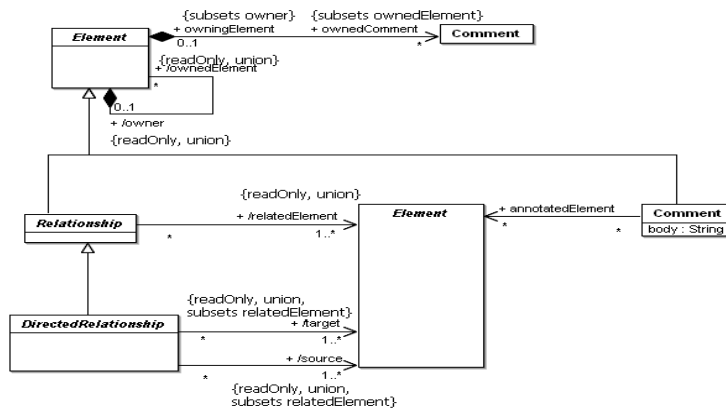


Figure 7.3 - Root diagram of the Kernel package

-22- 2007-02-07 - Sunilmm -

28/41

Table of Contents	
1. Scope	1
2. Conformance	1
2.1 Language Units	2
2.2 Compliance Levels	2
2.3 Meaning and Types of Compliance	6
2.4 Compliance Level Contents	8
3. Normative References	10
4. Terms and Definitions	10
5. Symbols	10
6. Additional Information	10
6.1 Changes to Adopted OMG Specifications	10
6.2 Architectural Alignment and MDA Support	10
6.3 On the Run-Time Semantics of UML	11
6.3.1 The Basic Premises	11
6.3.2 The Semantics Architecture	11
6.3.3 The Basic Causality Model	12
6.3.4 Semantics Descriptions in the Specification	13
6.4 The UML Metamodel	13
6.4.1 Models and What They Model	13
6.4.2 Semantic Levels and Naming	14
6.5 How to Read this Specification	15
6.5.1 Specification format	15
6.5.2 Diagram format	18
6.6 Acknowledgements	19
Part I - Structure	21
7. Classes	23

UML Superstructure Specification, v2.1.2

-23- 2007-02-07 - Reading -

Table of Contents	
1. Scope	1
2. Conformance	1
2.1 Language Units	2
2.2 Compliance Levels	2
2.3 Meaning and Types	6
2.4 Compliance Level Contents	8
3. Normative References	10
4. Terms and Definitions	10
5. Symbols	10
6. Additional Information	10
6.1 Changes to Adopted	10
6.2 Architectural Alignment	10
6.3 On the Run-Time Semantics of UML	11
6.3.1 The Basic Premises	11
6.3.2 The Semantics Architecture	11
6.3.3 The Basic Causality Model	12
6.3.4 Semantics Descriptions in the Specification	13
6.4 The UML Metamodel	13
6.4.1 Models and What They Model	13
6.4.2 Semantic Levels	14
6.5 How to Read this Specification	15
6.5.1 Specification format	15
6.5.2 Diagram format	18
6.6 Acknowledgements	19
Part I - Structure	21
7. Classes	23
7.1 Overview	23
7.2 Abstract Syntax	24
7.3 Class Descriptions	38
7.3.1 Abstraction (from Dependencies)	38
7.3.2 AggregationKind (from Kernel)	38
7.3.3 Association (from Kernel)	39
7.3.4 AssociationClass (from AssociationClasses)	47
7.3.5 BehavioralFeature (from Kernel)	48
7.3.6 BehavedClassifier (from Interfaces)	49
7.3.7 Class (from Kernel)	49
7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)	52
7.3.9 Comment (from Kernel)	57
7.3.10 Constraint (from Kernel)	58
7.3.11 DataType (from Kernel)	60
7.3.12 Dependency (from Dependencies)	62
7.3.13 DirectedRelationship (from Kernel)	63
7.3.14 Element (from Kernel)	64
7.3.15 ElementImport (from Kernel)	65
7.3.16 Enumeration (from Kernel)	67
7.3.17 EnumerationLiteral (from Kernel)	68
7.3.18 Expression (from Kernel)	69
7.3.19 Feature (from Kernel)	70
7.3.20 Generalization (from Kernel, PowerTypes)	71
7.3.21 GeneralizationSet (from PowerTypes)	75
7.3.22 InstanceSpecification (from Kernel)	82
7.3.23 InstanceValue (from Kernel)	85
7.3.24 Interface (from Interfaces)	86
7.3.25 InterfaceRealization (from Interfaces)	89
7.3.26 LiteralBoolean (from Kernel)	89
7.3.27 LiteralInteger (from Kernel)	90
7.3.28 LiteralNull (from Kernel)	91
7.3.29 LiteralSpecification (from Kernel)	92
7.3.30 LiteralString (from Kernel)	92
7.3.31 LiteralUnlimitedNatural (from Kernel)	93
7.3.32 MultiplicityElement (from Kernel)	94
7.3.33 NamedElement (from Kernel, Dependencies)	97
7.3.34 Namespace (from Kernel)	99
7.3.35 OpaqueExpression (from Kernel)	101
7.3.36 Operation (from Kernel, Interfaces)	103
7.3.37 Package (from Kernel)	107
7.3.38 PackageableElement (from Kernel)	109
7.3.39 PackageImport (from Kernel)	110
7.3.40 PackageMerge (from Kernel)	111
7.3.41 Parameter (from Kernel, AssociationClasses)	120
7.3.42 ParameterDirectionKind (from Kernel)	122
7.3.43 PrimitiveType (from Kernel)	122
7.3.44 Property (from Kernel, AssociationClasses)	123
7.3.45 Realization (from Dependencies)	129
7.3.46 RedefinableElement (from Kernel)	130

UML Superstructure Specification, v2.1.2

-23- 2007-02-07 - Reading -

Table of Contents 1. Scope 2. Conformance 2.1 Language Units 2.2 Compliance Levels 2.3 Meaning and Types 2.4 Compliance Level C 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 A 6.3.3 The Basic Causal 6.3.4 Semantics Descr 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	7.1 Overview 7.2 Abstract Syntax 7.3 Class Descriptions 7.3.1 Abstraction (from 7.3.2 AggregationKind 7.3.3 Association (from 7.3.4 AssociationClass 7.3.5 BehavioralFeatur 7.3.6 BehavioredClass 7.3.7 Class (from Kern 7.3.8 Classifier (from K 7.3.9 Comment (from K 7.3.10 Constraint (from 7.3.11 DataType (from 7.3.12 Dependency (fr 7.3.13 DirectedRelator 7.3.14 Element (from K 7.3.15 ElementImport (f 7.3.16 Enumeration (fr 7.3.17 EnumerationLite 7.3.18 Expression (fron 7.3.19 Feature (from Ke 7.3.20 Generalization (f 7.3.21 GeneralizationS 7.3.22 InstanceSpecific 7.3.23 InstanceValue (f 7.3.24 Interface (from I 7.3.25 InterfaceRealiza 7.3.26 LiteralBoolean (f 7.3.27 LiteralInteger (fr 7.3.28 LiteralNull (from 7.3.29 LiteralSpecialize 7.3.30 LiteralString (fr 7.3.31 LiteralUnlimited 7.3.32 MultiplicityElem 7.3.33 NamedElement 7.3.34 Namespace (fro 7.3.35 OpaqueExpress 7.3.36 Operation (from 7.3.37 Package (from K 7.3.38 PackageableEle 7.3.39 PackageImport 7.3.40 PackageMerge (f 7.3.41 Parameter (from 7.3.42 ParameterDirect 7.3.43 PrimitiveType (fr 7.3.44 Property (from K 7.3.45 Realization (from 7.3.46 RedefinableEler	7.3.47 Relationship (from Kernel) 132 7.3.48 Slot (from Kernel) 132 7.3.49 StructuralFeature (from Kernel) 133 7.3.50 Substitution (from Dependencies) 134 7.3.51 Type (from Kernel) 135 7.3.52 TypedElement (from Kernel) 136 7.3.53 Usage (from Dependencies) 137 7.3.54 ValueSpecification (from Kernel) 137 7.3.55 VisibilityKind (from Kernel) 139 7.4 Diagrams 140 8. Components 143 8.1 Overview 143 8.2 Abstract syntax 144 8.3 Class Descriptions 146 8.3.1 Component (from BasicComponents, PackagingComponents) 146 8.3.2 Connector (from BasicComponents) 154 8.3.3 ConnectorKind (from BasicComponents) 157 8.3.4 ComponentRealization (from BasicComponents) 157 8.4 Diagrams 159 9. Composite Structures 161 9.1 Overview 161 9.2 Abstract syntax 161 9.3 Class Descriptions 166 9.3.1 Class (from StructuredClasses) 166 9.3.2 Classifier (from Collaborations) 167 9.3.3 Collaboration (from Collaborations) 168 9.3.4 CollaborationUse (from Collaborations) 171 9.3.5 ConnectableElement (from InternalStructures) 174 9.3.6 Connector (from InternalStructures) 174 9.3.7 ConnectorEnd (from InternalStructures, Ports) 176 9.3.8 EncapsulatedClassifier (from Ports) 178 9.3.9 InvocationAction (from InvocationActions) 178 9.3.10 Parameter (from Collaborations) 179 9.3.11 Port (from Ports) 179 9.3.12 Property (from InternalStructures) 183 9.3.13 StructuredClassifier (from InternalStructures) 186 9.3.14 Trigger (from InvocationActions) 190 9.3.15 Variable (from StructuredActivities) 191 9.4 Diagrams 191 10. Deployments 193 UML Superstructure Specification, v2.1.2
	UML Superstructure Specification, v2.1.2	UML Superstructure Specification, v2.1.2

-21- 2007-02-07 - Reading -

Window
<pre> public size: Area = (100, 100) defaultSize: Rectangle protected visibility: Boolean = true private xWin: XWindow public display() hide() private attach(XWin: XWindow) </pre>

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.

-21- 2007-02-07 - Reading -

Win

```

public size: Area = f
defaultSize: R
protected visibility: Bool
private xWin: XWin
public display()
hide()
private attach(XWin)
        
```

Figure 7.29 - Class

7.3.8 Class

A classifier is a

Generalization

- "NameSpace"
- "Redefined"
- "Type (if...)"

Description

A classifier is a

A classifier is a

A classifier is a

Attributes

- isAbstract: If true, classification relation

Associations

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

52

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

53

-22-2007-02-07-Reading-

Win

```

public size: Area = f
defaultSize: R
protected visibility: Bool
private xWin: XWin
public display()
hide()
private attach(XWin)
        
```

Figure 7.29 - Class

7.3.8 Class

A classifier is a

Generalization

- "NameSpace"
- "Redefined"
- "Type (if...)"

Description

A classifier is a

A classifier is a

A classifier is a

Attributes

- isAbstract: If true, classification relation

Associations

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

52

- generalization: Specific Classifier
- /inheritedMember: Specific derived
- redefinedClassifier: Referer

Package Dependencies

- substitution: Referer Named

Package PowerTypes

- powertypeExtent: Design

Constraints

- [1] The general general = self
- [2] Generalization transitively not self.allParents()
- [3] A classifier self.parents()
- [4] The inherits self.inherits

Package PowerTypes

- [5] The Classifier GeneralizationSet itself nor may its instances also be its subclasses.

Additional Operations

- [1] 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.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(m: NamedElement) : Boolean;
 - pre: self.allParents()->collect(c | c.member)->includes(m)
 - if (self.inheritedMember->includes(m)) then
 - hasVisibilityOf = (m.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(rhs: Set(NamedElement)): Set(NamedElement);
 - inherit = rhs
- [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.ocl::KindOf(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

-22-2007-02-07-Reading-

UML Superstructure Specification, v2.1.2

52

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

UML Superstructure Specification, v2.1.2

52

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

56

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

UML Superstructure Specification, v2.1.2

56

Window

```

public
size: Area = 1
defaultSize: P
protected
visibility: Bool
private
width: XWidth
height: YHeight
display()
hide()
private
attach(XWindow)
        
```

Figure 7.29 - Class

7.3.8 Class

A classifier is a

Generalization

- "Name"sp
- "Redefin
- "Type (f

Description

A classifier is a

A classifier is a

A classifier is a

Attributes

- isAbstract: B
- If true, classifi
- relation

Associations

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

Package PowerTypes

- generalizati
- Specific
- classifi
- /inheritidB
- Specific
- derived
- redefinedCI
- Referer

Package Dependence

- substitution
- Referer
- Named

Package PowerTypes

- powertype
- Design

Constraints

(1) The general

general = se

(2) Generalizat

transitively

not self.allP

(3) A classifier

self.parents

(4) The inherite

self.inherite

Package PowerTypes

(5) The Classif

Generalizat

itself nor m

Additional Op

(1) The query a

inheritance,

Classifier:ca

allFeatures

(2) The query p

Classifier:p

parents = ge

Examples

The notion of p

subsets. In esse

a Classifier with

for that class.

Semantic Vari

The precise life

Notation

Classifier is an

in one place a d

default notation

compartments s

classifier can be

The name of an

ha

(6) The query e

in the speci

Classifier:c

conformsTo

Presentation

(7) The query i

to be redefi

Classifier:si

inher = inh

An abstract Cla

(8) The query i

the specific

redefined by

The type, visibi

in the model.

The individual

Classifier:si

maySpecial

Style Guidelin

An attribute ma

7.31.

- Attribute
- and using
- Center th
- Center k
- For those
- with an u
- Left just
- Begin an
- Show ful

Figure 7.30 - Ex

The attributes in

- ClassA:..
- ClassA:..
- ClassA:..
- ClassA:..
- ClassA:..
- ClassB:..
- ClassB:..
- ClassA:..
- ClassB:..

Figure 7.31 - A

An attribute ma

7.31.

Window

Figure 7.31 - A

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.

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 class, 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

- **multiplicity**body: 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

-21-2007-02-07 - Reading -

-21-2007-02-07 - main -

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**

-22- 2007-02-07 - Smof -

32/41

Benefits

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

-22- 2007-02-07 - Smof -

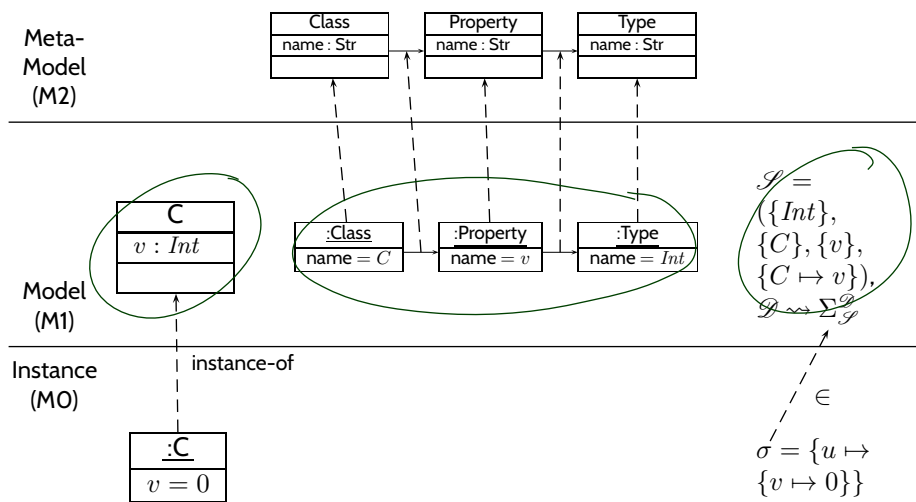
33/41

Meta-Modelling: Principle

-21- 2007-02-07 - main -

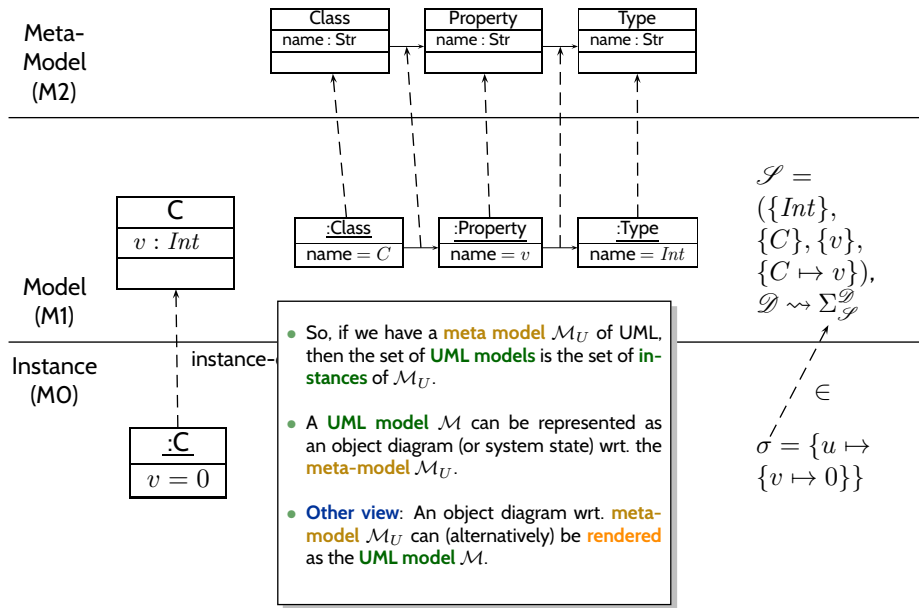
34/41

Modelling vs. Meta-Modelling



-21- 2007-02-07 - Sprinckle -

35/41



-21- 2007-02-07 - Sprinckle -

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 .

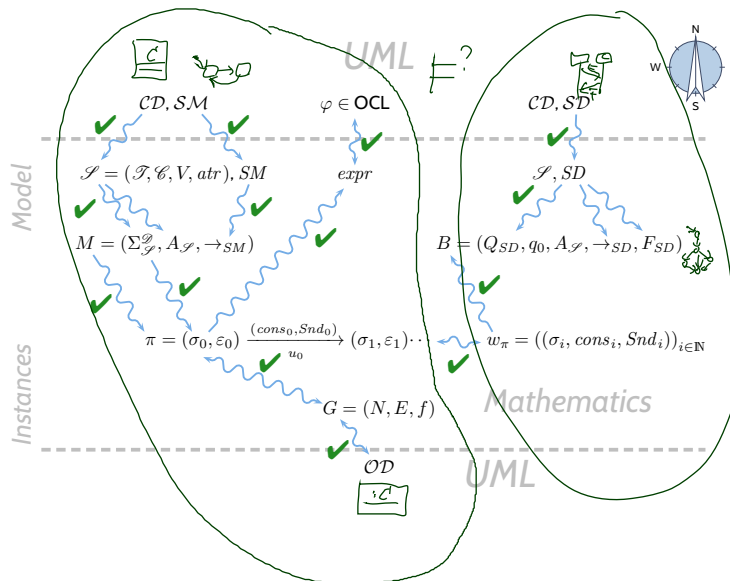
If O_1 is a **valid** object diagram of \mathcal{M}_U (i.e. satisfies all invariants from $Inv(\mathcal{M}_U)$), then \mathcal{M} is a well-formed UML model.

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.

-21- 2007-02-07 - Sprinckle -

And That's It!

The Map



Content

- **Lecture 1:** Introduction

-23- 2017-02-07 - main-

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

© 2016-2018

39/41

Content

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

-23- 2017-02-07 - main-

Contents & Goals

Last Lecture:

- Introduction: Motivation, Content, Formalia

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 Basic Object System Signatures relate to UML class diagrams?
- **Content:**
 - Basic Object System Signatures
 - Structures
 - System States

-2- 2016-10-18 - SemIn-

3/34

39/41

Content

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

-21-2007-02-07-main-

39/41

Content

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

-21-2007-02-07-main-

39/41

Contents & Goals

Contents & Goals

Last Lecture:

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

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 $\mathcal{S}(C)$ and \mathcal{I}_C related?
- **Content:**
 - OCL Syntax
 - OCL Semantics (over system states)

-4-20141219-Spr14-

2/35

Contents & Goals

Contents & Goals

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 $\mathcal{S}(C)$ and \mathcal{I}_C related?
- **Content:**
 - OCL Semantics
 - OCL Consistency and Satisfiability

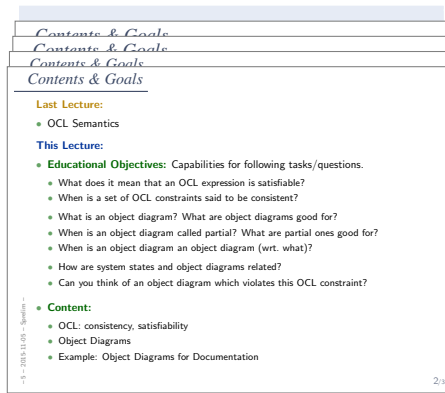
-4-20151108-Spr15-

2/36

Content

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

-22- 2007-02-07 - main -



Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
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

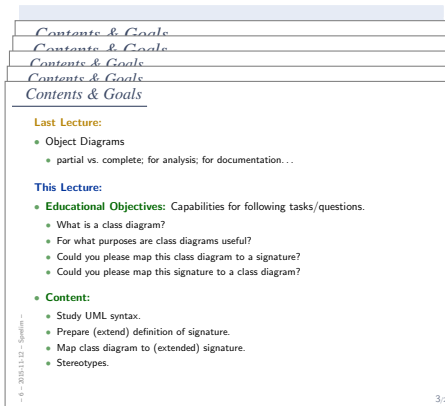
2/31

39/41

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

-22- 2007-02-07 - main -



Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
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.

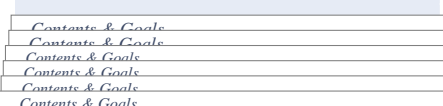
3/27

39/41

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

-23- 2007-02-07 - main -



Contents & Goals
Contents & Goals
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 - 2005-11-17 - Sprint11 -

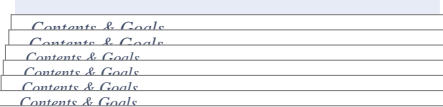
2/23

39/41

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

-23- 2007-02-07 - main -



Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals

Contents & Goals

Last Lectures:

- completed class diagrams... except for associations.

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
 - Please explain this class diagram with associations.
 - Which annotations of an association arrow are semantically relevant?
 - What's a role name? What's it good for?
 - What is "multiplicity"? How did we treat them semantically?
 - What is "reading direction", "navigability", "ownership", ...?
 - What's the difference between "aggregation" and "composition"?
- **Content:**
 - Study concrete syntax for "associations".
 - (**Temporarily**) extend signature, define mapping from diagram to signature.
 - Study effect on OCL.
 - Btw.: where do we put OCL constraints?

→ 8 - 2005-11-29 - Sprint11 -

2/34

39/41

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
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
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.

- 2015-2016 - Spring -

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
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
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

- 2015-2016 - Spring -

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
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
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

11 - 2019 12 30 - Sprint 11 -

2/34

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
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
Contents & Goals
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

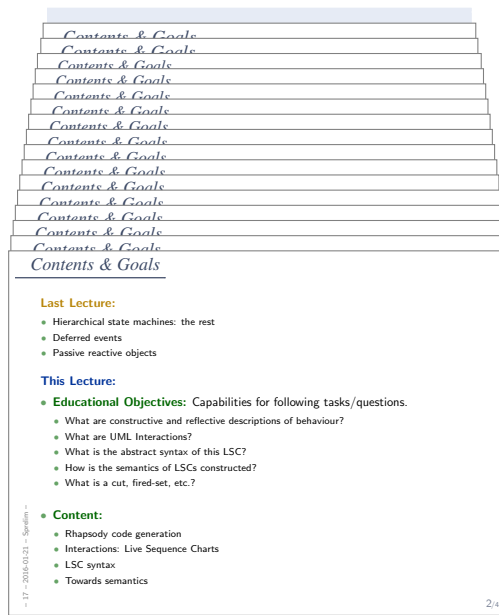
12 - 2019 12 15 - Sprint 12 -

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

-21- 2017-02-07 - main -



Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Last Lecture:

- Hierarchical state machines: the rest
- Deferred events
- Passive reactive objects

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
 - What are constructive and reflective descriptions of behaviour?
 - What are UML Interactions?
 - What is the abstract syntax of this LSC?
 - How is the semantics of LSCs constructed?
 - What is a cut, fired-set, etc.?

Content:

- Rhapsody code generation
- Interactions: Live Sequence Charts
- LSC syntax
- Towards semantics

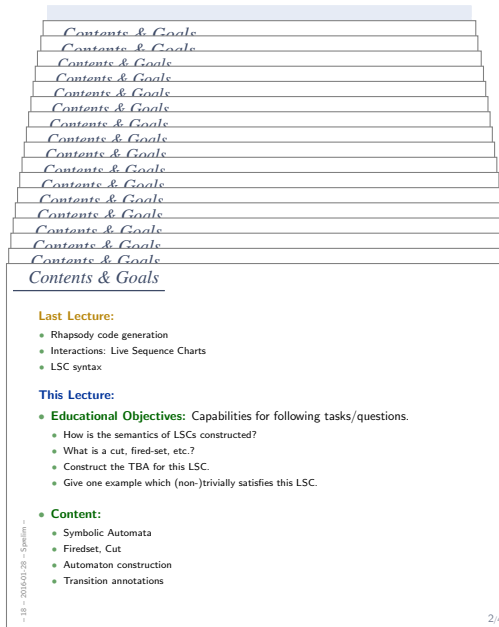
-17- 2016/02/21 - Spafin -

2/45

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

-21- 2017-02-07 - main -



Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Contents & Goals

Last Lecture:

- Rhapsody code generation
- Interactions: Live Sequence Charts
- LSC syntax

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:

- Symbolic Automata
- Firedset, Cut
- Automaton construction
- Transition annotations

-18- 2016/02/28 - Spafin -

2/45

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

-21-2007-02-07-main-

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

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:** 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

-21-2007-02-07-main-

Contents & Goals

Last Lecture:

- Firedset, Cut
- Automaton construction
- Transition annotations

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
 - What's the Liskov Substitution Principle?
 - What is late/early binding?
 - What is the subset / uplink semantics of inheritance?
 - What's the effect of inheritance on LSCs, State Machines, System States?
- **Content:**
 - Inheritance in UML: concrete syntax
 - Liskov Substitution Principle — desired semantics
 - Two approaches to obtain desired semantics

2/30

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

-21-2017-02-07-main-

Contents & Goals

Last Lecture:

- Firedset, Cut
- Automaton construction
- Transition annotations

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
 - What's the Liskov Substitution Principle?
 - What is late/early binding?
 - What is the subset / uplink semantics of inheritance?
 - What's the effect of inheritance on LSCs, State Machines, System States?
- **Content:**
 - Inheritance in UML: concrete syntax
 - Liskov Substitution Principle — desired semantics
 - Two approaches to obtain desired semantics

2/30

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

-21-2017-02-07-main-

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

2/31

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.