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: Why and What

- **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.
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)?
Meta-Modelling: Principle
Modelling vs. Meta-Modelling

Meta-Model (M2)

Model (M1)

Instance (M0)

\[ S = (\{\text{Int}\}, \text{Int}, \{C\}, \{v\}, \{C \mapsto \{v\}\}, \emptyset \mapsto \Sigma) \]

\[ \sigma = \{u \mapsto \{v \mapsto 0\}\} \]
### Modelling vs. Meta-Modelling

<table>
<thead>
<tr>
<th>Meta-Model (M2)</th>
<th>Model (M1)</th>
<th>Instance (M0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C</strong></td>
<td>:Class</td>
<td>:C</td>
</tr>
<tr>
<td><em>v : Int</em></td>
<td>*name = C</td>
<td><em>v = 0</em></td>
</tr>
<tr>
<td></td>
<td>:Property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*name = v</td>
<td></td>
</tr>
<tr>
<td></td>
<td>:Type</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>name = Int</em></td>
<td></td>
</tr>
</tbody>
</table>

#### 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** $\mathcal{M}$.
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() \(\rightarrow\) 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 \(\text{Inv} (\mathcal{M}_U)\)), 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
Figure 7.12 - Classes diagram of the Kernel package
Figure 7.11 - Operations diagram of the Kernel package
Figure 7.10 - Features diagram of the Kernel package
Figure 7.9 - Classifiers diagram of the Kernel package
Namespaces (OMG, 2007b, 26)

Figure 7.4 - Namespaces diagram of the Kernel package
Figure 7.3 - Root diagram of the Kernel package
Interesting: Declaration/Definition (OMG, 2007b, 424)

Figure 13.6 - Common Behavior
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
Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure
# 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
# Table of Contents

1. **Scope**  
2. **Conformance**  
   2.1 Language Units  
   2.2 Compliance Levels  
   2.3 Meaning and Types  
   2.4 Compliance Level Comments  
3. **Normative References**  
4. **Terms and Definitions**  
5. **Symbols**  
6. **Additional Information**  
   6.1 Changes to Adopted Terminology  
   6.2 Architectural Alignments  
   6.3 On the Run-Time Semantics of UML  
   6.4 The UML Metamodel  
   6.5 How to Read this Specification  
   6.6 Acknowledgements  
7. **Classes**  
   7.1 Overview  
   7.2 Abstract Syntax  
   7.3 Class Descriptions  
      7.3.1 Abstraction (from Dependencies)  
      7.3.2 AggregationKind (from Kernel)  
      7.3.3 Association (from Kernel)  
      7.3.4 AssociationClass (from AssociationClasses)  
      7.3.5 BehavioralFeature (from Kernel)  
      7.3.6 BehavioralClassifier (from Interfaces)  
      7.3.7 Class (from Kernel)  
      7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)  
      7.3.9 Comment (from Kernel)  
      7.3.10 Constraint (from Kernel)  
      7.3.11 DataType (from Kernel)  
      7.3.12 Dependency (from Dependencies)  
      7.3.13 DirectedRelationship (from Kernel)  
      7.3.14 Element (from Kernel)  
      7.3.15 ElementImport (from Kernel)  
      7.3.16 Enumeration (from Kernel)  
      7.3.17 EnumerationLiteral (from Kernel)  
      7.3.18 Expression (from Kernel)  
      7.3.19 Feature (from Kernel)  
      7.3.20 Generalization (from Kernel, PowerTypes)  
      7.3.21 GeneralizationSet (from PowerTypes)  
      7.3.22 InstanceSpecification (from Kernel)  
      7.3.23 InstanceValue (from Kernel)  
      7.3.24 Interface (from Interfaces)  
      7.3.25 InterfaceRealization (from Interfaces)  
      7.3.26 LiteralBoolean (from Kernel)  
      7.3.27 LiteralInteger (from Kernel)  
      7.3.28 LiteralNull (from Kernel)  
      7.3.29 LiteralSpecification (from Kernel)  
      7.3.30 LiteralString (from Kernel)  
      7.3.31 LiteralUnlimitedNatural (from Kernel)  
      7.3.32 MultiplicityElement (from Kernel)  
      7.3.33 NamedElement (from Kernel, Dependencies)  
      7.3.34 Namespace (from Kernel)  
      7.3.35 OpaqueExpression (from Kernel)  
      7.3.36 Operation (from Kernel, Interfaces)  
      7.3.37 Package (from Kernel)  
      7.3.38 PackageableElement (from Kernel)  
      7.3.39 PackageImport (from Kernel)  
      7.3.40 PackageMerge (from Kernel)  
      7.3.41 Parameter (from Kernel, AssociationClasses)  
      7.3.42 ParameterDirectionKind (from Kernel)  
      7.3.43 PrimitiveType (from Kernel)  
      7.3.44 Property (from Kernel, AssociationClasses)  
      7.3.45 Realization (from Dependencies)  
      7.3.46 RedefinableElement (from Kernel)  
   7.4 Normative References
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.
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)"
- "RedefinableElement (from Kernel)"
- "Type (from Kernel)"

Description

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. An abstract classifier is intended to be used by other classifiers (e.g., as the target of general metarelationships or generalization relationships). Attributes

- isAbstract: Boolean
  If true, the Classifier does not provide a complete declaration and cannot typically 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.
- /feature: Feature [*]
  Specifies each feature defined in the classifier. This is a derived union.
- /general: Classifier [*]
  Specifies the general Classifiers for this Classifier. This is derived.

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.

\[ \text{general} = \text{self.parents()} \]

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

\[ \text{not self.allParents() -> includes(self)} \]

[3] A classifier may only specialize classifiers of a valid type.

\[ \text{self.parents() -> forAll(c | self.maySpecializeType(c))} \]

[4] The inheritedMember association is derived by inheriting the inheritable members of the parents.

\[ \text{self.inheritedMember -> includesAll(self.inherit(self.parents() -> collect(p | p.inheritableMembers(self))))} \]

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.

\[ \text{Classifier::allFeatures() -> Set(Feature)}; \]

\[ \text{allFeatures = member -> select(oclIsKindOf(Feature))} \]


\[ \text{Classifier::parents() -> Set(Classifier)}; \]

\[ \text{parents = generalization.general} \]
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())
```

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))
```

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
```

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))
```

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);

```
h = inhs
inherited = self.parents()->select(p | p.inherits->includes(h))
```

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)
```

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.

```
Design: Classifier
  classifier: Classifier
    display(): Element::display()
    hide(): Element::hide()
    referredClassifiers(): Classifier[*]
    visible: Visibility = #public
  inherit(): classifier->select(c | c.inherits->includes(self))
  maySpecializeType(c: Classifier): Boolean
    maySpecializeType = self.oclIsKindOf(c.oclType)
```

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.

```
Classifier::maySpecializeType(c: Classifier): Boolean
  maySpecializeType = self.oclIsKindOf(c.oclType)
```

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 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.

Constraints

1. The general classifier general = self parents()->select(p | p.inherits->includes(self))
2. Generalizations transitively
3. A classifier self.parents()->select(p | p.inherits->includes(self))
4. The inherited classifier self.inherited = self.parents()->select(p | p.inherits->includes(self))
5. The Classifier Generalization self = self.parents()->select(p | p.inherits->includes(self))
6. A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.
7. Attributes
   - isAbstract: If true, the classifier is not a specific classifier
8. Additional Operations
   - /attribute: F
     Refers: Classifier::allFeatures
   - /feature: F
     Refers: Classifier::allFeatures
   - /general: G
     Refers: Classifier::allParents
```

```
Figure 7.29 - Classifier
7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)

- powerTypeExtent: GeneralizationSet
  Describes the GeneralizationSet of which the associated Classifier is a power type.
  Constraints
  - powerTypeExtent = self.allParents()->select(p | p.inherits->includes(self))

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

- Classifier::inherit(): Classifier
  Inherits = self.parents()->select(p | p.inherits->includes(h))

- Classifier::maySpecializeType(): Classifier
  MaySpecializeType = self.oclIsKindOf(c.oclType)
```

A classifier is intended to be used by other classifiers (e.g., as the target of general metarelationships or generalization relationships). A classifier is not a specific classifier itself nor may its instances also be its subclasses.
Classifiers 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 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 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.
Examples

Class A
name: String
shape: Rectangle
+ size: Integer (0..1) / area: Integer (readOnly)
width: Integer

Class B
id (redefines name)
shape: Square
height: Integer
/ width

Figure 7.30 - Examples of attributes

The attributes in Figure 7.30 are explained below.

- Class A::name is an attribute with type String.
- Class A::shape is an attribute with type Rectangle.
- Class A::size is a public attribute of type Integer with multiplicity 0..1.
- Class A::area is a derived attribute with type Integer. It is marked as readOnly.
- Class A::height is an attribute of type Integer with a default initial value of 5.
- Class A::width is an attribute of type Integer.

- Class B::id is an attribute that redefines Class A::name.
- Class B::shape is an attribute that redefines Class A::shape. It has type Square, a specialization of Rectangle.
- Class B::height is an attribute that redefines Class A::height. It has a default of 7 for Class B instances that overrides the Class A default of 5.
- Class B::width is a derived attribute that redefines Class A::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.

![Diagram of Window](image-url)

Figure 7.31 - Association-like notation for attribute
7.3.8 Classifier

A classifier is an abstract model element that can be explicitly defined or implicitly specified for instances of the specific classifier. A classifier is a component of other classifiers. A classifier is a component of other classifiers. A classifier is a component of other classifiers.

Attributes
- isAbstract: true, false
- Generalization: Classifier[*]
- NameSpace: NameSpace[*]

Additional Options
- [attribute: P
- Feature: Classifier[*]
- General: Classifier[*]

Generalsizations
- “NameSpace” (from Kernel)
- “RedefinedClassifier” (from Kernel)
- “Type (from Kernel)”

Description
A classifier is an abstract model element that can be explicitly defined or implicitly specified for instances of the specific classifier. A classifier is a component of other classifiers. A classifier is a component of other classifiers. A classifier is a component of other classifiers. A classifier is a component of other classifiers.

Generalizations
- “NameSpace” (from Kernel)
- “RedefinedClassifier” (from Kernel)
- “Type (from Kernel)”

Style

7.3.9 Comment (from Kernel)

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

Attributes
- multiplicity: String [0..1]

Associations
- annotatedElement: Element[*]

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: String [0..1]

Associations
- annotatedElement: Element[*]

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.
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**
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

Mathematics

Model

Instances

UML

\[ \varphi \in \text{OCL} \]

\[ M = (\Sigma^\varphi, A^\varphi, \rightarrow_{SM}) \]

\[ \pi = (\sigma_0, \varepsilon_0, (\text{cons}_0, \text{Snd}_0)) \]

\[ \xi_i = ((\sigma_i, \text{cons}_i, \text{Snd}_i))_{i \in \mathbb{N}} \]

\[ G = (N, E, f) \]

\[ B = (Q_{SD}, q_0, A_{SD}, \rightarrow_{SD}, F_{SD}) \]
Content

- **Lecture 1**: Introduction

*Software Design, Modelling and Analysis in UML*

Lecture 1: Introduction

2015-10-20

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
# Contents

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

## 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
Content

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

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

**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.
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

---

**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?
Content

- Lecture 1: Introduction
- Lecture 2: Semantical Model
- Lecture 3: Object Constraint Language (OCL)
- Lecture 4: OCL Semantics
- Lecture 5: Object Diagrams
- Lecture 6: Class Diagrams I
- Lecture 7: Class Diagrams II
- Lecture 8: Class Diagrams III
- Lecture 9: Class Diagrams IV
Content

- **Lecture 1**: Introduction
- **Lecture 2**: Semantical Model
- **Lecture 3**: Object Constraint Language (OCL)
- **Lecture 4**: OCL Semantics
- **Lecture 5**: Object Diagrams
- **Lecture 6**: Class Diagrams I
- **Lecture 7**: Class Diagrams II
- **Lecture 8**: Class Diagrams III
- **Lecture 9**: Class Diagrams IV
- **Lecture 10**: State Machines Overview

---

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

---

Contents & Goals

<table>
<thead>
<tr>
<th>Contents &amp; Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Lecture:</td>
</tr>
<tr>
<td>( Mostly) completed discussion of modelling structure.</td>
</tr>
<tr>
<td>This Lecture:</td>
</tr>
<tr>
<td>Educational Objectives: Capabilities for following tasks/questions.</td>
</tr>
<tr>
<td>- What’s the purpose of a behavioural model?</td>
</tr>
<tr>
<td>- What does this State Machine mean? What happens if I inject this event?</td>
</tr>
<tr>
<td>- Can you please model the following behaviour.</td>
</tr>
<tr>
<td>Content:</td>
</tr>
<tr>
<td>- For completeness: Modelling Guidelines for Class Diagrams</td>
</tr>
<tr>
<td>- Purposes of Behavioural Models</td>
</tr>
<tr>
<td>- UML Core State Machines</td>
</tr>
</tbody>
</table>
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
What is: Signal, Event, Ether, Transformer, Step, RTC.

What does this State Machine mean? What happens if I inject this event?

What is “reading direction”, “navigability”, “ownership”, . . . ?

For completeness: Modelling Guidelines for Class Diagrams

Capabilities for following tasks/questions.

Lecture 7

What’s a role name? What’s it good for?

Stereotypes – for documentation.

Basic Object System Signature

Lecture 10

Can you think of an object diagram which violates this OCL constraint?

Educational Objectives:

Structures

What’s the difference between “aggregation” and “composition”?

2

Ether/event pool

Please un-abbreviate all abbreviations in this OCL expression.

Content:

Please formalise this constraint in OCL.

Lecture 1

Could you please map this class diagram to a signature?

Give a system state satisfying this constraint?

How do Basic Object System Signatures relate to UML class diagrams?

How did we treat “multiplicity” semantically?

OCL Consistency and Satisfiability

Content:

Example: Object Diagrams for Documentation

Capabilities for following tasks/questions.

Compute the value of a given OCL constraint in a system state with links.

Study concrete syntax for “associations”.

Content:

What does it mean that an OCL expression is satisfiable?

What does “navigability”, “ownership”, . . . mean?

Stereotypes.

System States

Core State Machine syntax

For what purposes are class diagrams useful?

Associations in OCL syntax.

Content:

What is a signature, an object, a system state, etc.?

Capabilities for these tasks/questions:

Educational Objectives:

Map class diagram to (extended) signature.

2

Content:

How are system states and object diagrams related?

Capabilities for following tasks/questions.

Study effect on OCL.

Please formalise this constraint in OCL.

Associations and OCL: semantics.

What is the purpose of signature, object, etc. in the course?

Could you please map this signature to a class diagram?

Capabilities for following tasks/questions.

Object Diagrams

2

– 11 – 2015-12-10 – Sprelim –

Educational Objectives:

structure

OCL Syntax

Associations: the rest.

Study UML syntax.

Capabilities for following tasks/questions.

Purposes of Behavioural Models

2

Capabilities for following tasks/questions.

Educational Objectives:

Content:

Lecture 8

When is a set of OCL constraints said to be consistent?

Content:

System configuration cont’d

Associations syntax and semantics.

Please explain this OCL constraint.

completed class diagrams. . . except for associations.

Can you please model the following behaviour.

2

When is an object diagram an object diagram (wrt. what)?

Lecture 4

Lecture 11

Does this OCL constraint hold in this system state?

Give a system state satisfying this constraint?

What is: Signal, Event, Ether, Transformer, Step, RTC.

In what sense is OCL a three-valued logic? For what purpose?

Could you please model the following behaviour.

2

Study effect on OCL.

Please formalise this constraint in OCL.

OCL Consistency and Satisfiability

Content:

Example: Object Diagrams for Documentation

Capabilities for following tasks/questions.

Compute the value of a given OCL constraint in a system state with links.

Study concrete syntax for “associations”.

Content:

What does it mean that an OCL expression is satisfiable?

What does “navigability”, “ownership”, . . . mean?

Stereotypes.

System States

Core State Machine syntax

For what purposes are class diagrams useful?

Associations in OCL syntax.

Content:

What is a signature, an object, a system state, etc.?

Capabilities for these tasks/questions:

Educational Objectives:

Map class diagram to (extended) signature.

2

Content:

How are system states and object diagrams related?

Capabilities for following tasks/questions.

Study effect on OCL.

Please formalise this constraint in OCL.

Associations and OCL: semantics.

What is the purpose of signature, object, etc. in the course?

Could you please map this class diagram to a signature?

Which annotations of an association arrow are semantically relevant?

. . .

In what sense is OCL a three-valued logic? For what purpose?

Could you please map this signature to a class diagram?

Capabilities for following tasks/questions.

Object Diagrams

2

– 11 – 2015-12-10 – Sprelim –

Educational Objectives:

structure

OCL Syntax

Associations: the rest.

Study UML syntax.

Capabilities for following tasks/questions.

Purposes of Behavioural Models

2

Capabilities for following tasks/questions.

Educational Objectives:

Content:

Lecture 8

When is a set of OCL constraints said to be consistent?

Content:

System configuration cont’d

Associations syntax and semantics.

Please explain this OCL constraint.

completed class diagrams. . . except for associations.

Can you please model the following behaviour.

2

When is an object diagram an object diagram (wrt. what)?

Lecture 4

Lecture 11

Does this OCL constraint hold in this system state?

Give a system state satisfying this constraint?

What is: Signal, Event, Ether, Transformer, Step, RTC.

In what sense is OCL a three-valued logic? For what purpose?

Could you please model the following behaviour.

2

Study effect on OCL.

Please formalise this constraint in OCL.

Associations and OCL: semantics.

What is the purpose of signature, object, etc. in the course?

Could you please map this class diagram to a signature?
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
### 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

---

### 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
How are system states and object diagrams related?

Capabilities for these tasks/questions:

Educational Objectives:

Could you please map this class diagram to a signature?

What is visibility good for?

Could you please map this signature to a class diagram?

What are roles of OCL contraints in behavioural models?

partial vs. complete; for analysis; for documentation. . .

Associations and OCL: semantics.

What's a role name? What's it good for?

(What is a class diagram?

Step, Run-to-Completion Step

2

Lecture 15

Rules (i) to (v) for hierarchical state machines

What is the semantics of 'abstract'?

Structures

Transformers

Associations in OCL syntax.

How are

Educational Objectives:

Content:

2

Associations: the rest.

What is a signature, an object, a system state, etc.?

What does it mean that an OCL expression is satisfiable?

When is a set of OCL constraints said to be consistent?

Can you please model the following behaviour.

Map class diagram to (extended) signature cont'd.

OCL Syntax

What is: Signal, Event, Ether, Transformer, Step, RTC.

Which annotations of an association arrow are semantically relevant?

Can you think of an object diagram which violates this OCL con straint?

– 14 – 2016-01-12 – Sprelim –

Lecture 12

Introduction: Motivation, Content, Formalia

What is the purpose of signature, object, etc. in the course?

Educational Objectives:

Give a system state satisfying this constraint?

Transitions by Rule (i) to (v).

For what purposes are class diagrams useful?

Capabilities for following tasks/questions.

Please formalise this constraint in OCL.

In what sense is OCL a three-valued logic? For what purpose?

OCL Semantics

– 15 – 2016-01-14 – Sprelim –

2

2

What does this State Machine mean? What happens if I inject th is event?

System States

Educational Objectives:

Content:

2

Capabilities for following tasks/questions.

UML Core State Machines

Map class diagram to (extended) signature.

Stereotypes.

Representing class diagrams as (extended) signatures — for the moment without

Does this OCL constraint hold in this system state?

How to define what happens at “system / model startup”?

Study effect on OCL.

Please explain this OCL constraint.

Lecture 9

When is an object diagram an object diagram (wrt. what)?

Content:

2

System configuration cont'd

Transformers

Basic causality model

Ether

What does this State Machine mean? What happens if I inject th is event?

completed class diagrams. . . except for associations.

– 6 – 2015-11-12 – Sprelim –

Associations syntax and semantics.

Content:

– 11 – 2015-12-10 – Sprelim –

Lecture 3

29

Example: Object Diagrams for Documentation

What is a legal transition?

What if things are missing?

Capabilities for following tasks/questions.

How do Basic Object System Signatures relate to UML class dia grams?

Btw.: where do we put OCL constraints?

Educational Objectives:

Initial states

– 13 – 2015-12-17 – Sprelim –

Visibility as an extension of well-typedness.

A closer look onto code generation

Capabilities for following tasks/questions.

OCL Syntax

Step, Run-to-Completion Step

Educational Objectives:

Contents & Goals

Last Lecture:

•

•

•

•

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
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

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
Educational Objectives:
What does this State Machine mean? What happens if I inject this event?

Rules (i) to (v) for hierarchical state machines

Capabilities for following tasks/questions.
In what sense is OCL a three-valued logic? For what purpose?

Basic causality model
How do entry / exit actions work? What about do-actions?

What is simple state, OR-state, AND-state?

What are UML Interactions?
How are passive reactive objects treated in Rhapsody's UML semantics?

Give a system state satisfying this constraint?

Can you think of an object diagram which violates this OCL constraint?

What is: Signal, Event, Ether, Transformer, Step, RTC.
Passive reactive objects

What is the semantics of 'abstract'?

Stereotypes.
What is "reading direction", "navigability", "ownership", . . . ?

Content:
What is an object diagram? What are object diagrams good for?
When is a set of OCL constraints said to be consistent?
Give a system state satisfying this constraint?

Transformers

OCL Consistency and Satisfiability
Interactions: Live Sequence Charts
Introduction: Motivation, Content, Formalia

Basic Object System Signatures

2

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.?

Last Lecture:
Hierarchical state machines: the rest
Deferred events
Passive reactive objects

This Lecture:

Educational Objectives: Capabilities for following tasks/questions.

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

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
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

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
Inheritance in UML: concrete syntax

What are constructive and reflective descriptions of behaviour?

What is an object diagram? What are object diagrams good for?

Give one example which (non-)trivially satisfies this LSC.

A closer look onto code generation

OCL: consistency, satisfiability

What does this State Machine mean? What happens if I inject this event?

Educational Objectives:

Basic Object System Signature

Passive reactive objects

Associations: the rest.

step, RTC-step, divergence

Transformers

Prepare (extend) definition of signature.

Content:

Is this UML model consistent with that OCL constraint?

Could you please map this signature to a class diagram?

What makes a class diagram a good class diagram?

When is an object diagram an object diagram (wrt. what)?

What is the subset / uplink semantics of inheritance?

What if things are missing?

What is a legal state configuration?

OCL Semantics

How is enabledness of transitions defined for hierarchical state machines?

Introduction: Motivation, Content, Formalia

What is the idea of deferred events?

– 20 – 2016-02-04 – Sprelim –

Educational Objectives:

What is late/early binding?

What is: Signal, Event, Ether, Transformer, Step, RTC.

Construct the TBA for this LSC.

Content:

UML Core State Machines

Ether/event pool

2 . . .

What is the idea of meta-modelling?

Can you please model the following behaviour.

Forbidden scenarios

Study UML syntax.

What is a class diagram?

2

Educational Objectives:

Basic causality model

Purposes of Behavioural Models

Passive reactive objects

What is a cut, fired-set, etc.?

Core State Machine syntax

What is the purpose of signature, object, etc. in the course?

OCL Semantics

2

Capabilities for following tasks/questions.

Capabilities for following tasks/questions.

Symbolic Automata

Please formalise this constraint in OCL.

What is a cut, fired-set, etc.?

How is the semantics of LSCs constructed?

Lecture 5

How are passive reactive objects treated in Rhapsody’s UML semantics?

What about junction, choice, terminate, etc.?

Example: Object Diagrams for Documentation

Educational Objectives:

System configuration cont’d

Step, Run-to-Completion Step

What is the semantics of ‘abstract’?

When is an object diagram called partial? What are partial ones good for?

Content:

Stereotypes.

Lecture 16

Lecture 20

Associations in OCL syntax.

What’s a role name? What’s it good for?

Content:

Lecture 9

Automaton construction

In what sense is OCL a three-valued logic? For what purpose?

Capabilities for following tasks/questions.

Towards semantics

Please explain this class diagram with associations.

What’s the Liskov Substitution Principle?

create, destroy actions

Associations syntax and semantics.

Two approaches to obtain desired semantics

– 20 – 2016-02-11 – main –

– 6 – 2015-11-12 – Sprelim –

– 12 – 2015-12-15 – Sprelim –

– 2 – 2015-10-22 – Sprelim –

– 13 – 2015-12-17 – Sprelim –

– 20 – 2016-02-04 – Sprelim –

– 2 – 2016-02-11 – main –

– 6 – 2015-11-12 – Sprelim –

– 12 – 2015-12-15 – Sprelim –

– 2 – 2015-10-22 – Sprelim –

– 13 – 2015-12-17 – Sprelim –

– 20 – 2016-02-04 – Sprelim –
References
References


