UML Class Diagrams 2

Martin Nečaský
Dept. of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
Previous Lecture Reminder

- What are UML class diagrams?
  - basic constructs
    - classes
    - class properties (attributes and association ends)
    - binary and n-ary associations
    - association classes
    - navigability
    - multiplicities
    - property modifiers
Formal model of Properties
Attributes vs. Associations

- attribute
  - property related to a class by `ownedAttribute`

- association end
  - property related to an association by `memberEnd`

- note: property can be both attribute and association end; such property must be owned by the class

- note: association ends of associations with more than two ends must be owned by the associations
Attributes vs. Associations

- **Person**
  - instance of **Class**
- **name**
  - instance of **Property**
  - related to **Person** by **ownedAttribute**

- **String**
  - value of **type** (inherited from **TypedElement**)

```java
class Person {
    String name;
    String email;  // 0..* strings
    String phone;  // 0..*
    Date registrationDate;
    URL homepage;
    URL dblp;
}
```
Attributes vs. Associations

- **Institution, Address**
  - instance of **Class**
- **anonymous**
  - instance of **Association**
- **address**
  - instance of **Property**
  - related to **Institution** by **ownedAttribute**
  - related to **anonymous** by **memberEnd**
  - owned by **Institution**
Attributes vs. Associations

- **Person, Institution**
  - instances of *Class*

- **anonymous**
  - instance of *Association*

- **employee, employer**
  - instances of *Property*
  - related to **anonymous** by *memberEnd* and *ownedEnd*
  - **owned by anonymous**
(Property Values)

- **UML specification**
  - property represents a declared state of one or more instances in terms of a named relationship to one or more values

- **mathematically**
  - property P is a function
    \[ P : I(C_1) \times \cdots \times I(C_n) \rightarrow 2^T \]

  where
  - classes \( C_1, \ldots, C_n \) are determined by the owner of P
    - \( \{C_1, \ldots, C_n\} \) is called context of P
  - \( I(C_i) \) denotes the set of instances of \( C_i \)
  - \( \min \leq |P(i_1, \ldots, i_n)| \leq \max \) where \((\min, \max)\) is the multiplicity of P
  - \( P(i_1, \ldots, i_n) \) is an (un)ordered set or multi-set
    - depends on modifiers assigned to P
if the owner of \( P \) is a class \( C \) then
- \( n = 1 \)
- \( C_1 = C \)
if the owner of P is a class C then

- \( n = 1 \)
- \( C_1 = C \)
Property Values

- if the owner of P is an association R then
  - \( n = |R.\text{memberEnd}| - 1 \)
  - \( C_1, ..., C_n = R.\text{memberEnd.type} \setminus P.\text{type} \)
if the owner of P is an association R then
- \( n = |R\text{.memberEnd}| - 1 \)
- \( C_1, \ldots, C_n = R\text{.memberEnd}\text{.type} \setminus P\text{.type} \)
Property Values

- \( P(i_1, \ldots, i_n) \) is an (un)ordered set or multi-set
  - depends on modifiers assigned to \( P \)
  - \{ordered\} \( \rightarrow P\text{.isOrdered} = \text{true} \) (default false)
  - \{non-unique\} \( \rightarrow P\text{.isUnique} = \text{false} \) (default true)

<table>
<thead>
<tr>
<th>isOrdered</th>
<th>isUnique</th>
<th>Collection Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
<td>set</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>list (ordered set)</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>bag (multi-set)</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>sequence (ordered multi-set)</td>
</tr>
</tbody>
</table>
Specializing Attributes and Associations

- common misunderstanding is that only classes can be specialized
- specializing attributes and associations is also possible!
  - via property subsetting and redefining

```
<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ isDerived:  boolean = false</td>
</tr>
<tr>
<td>+ isReadOnly:  boolean = false</td>
</tr>
<tr>
<td>+ isDerivedUnion:  AggregationKind = none</td>
</tr>
<tr>
<td>+ aggregation:  AggregationKind = none</td>
</tr>
<tr>
<td>+ isID:  boolean = false</td>
</tr>
</tbody>
</table>
```

![Diagram](attachment://diagram.png)
Property Subsetting and Redefining

```java
class Subsetting

Team Person
+ skill: string [0..*]
Researcher
+ skill: ResearchSkill [1..*]
+ researchField: ResearchSkill [1..*]
+ team 0..*
+ member 1..*
+ externalTeam 0..*
{subsets team}
+ externalMember 0..*
{subsets member}
+ leaderOf 0..*
{subsets team}
+ leader 0..1
{subsets member}
```
Property Subsetting

- set of instances of a property $P_{sub}$ is a subset of a set of instances of another property $P$
Property Subsetting – Constraints

- name of $P_{sub}$ must differ from the name of $P$
Property Subsetting – Constraints

- type of $P_{\text{sub}}$ must be the same as the type of $P$ or it must be its specialization.
Property Subsetting – Constraints

- upper multiplicity of $P_{sub}$ must be less then the upper multiplicity of $P$
  - lower multiplicity is not restricted, i.e. multiplicity of $P_{sub}$ does not necessarily need to be a sub-interval of the multiplicity of $P$
Property Subsetting – Constraints

- context of $P_{sub}$ must conform to the context of $P$

\[(\forall C_{sub} \in \text{context}(P_{sub}))(\exists C \in \text{context}(P)) \quad (C = C_{sub} \lor C \text{ is generalization of } C_{sub})\]
specific property $P_{\text{red}}$ redefines a general property $P$ in order to augment, constraint or override $P$
Property Redefining – Constraints

- P must be inherited by the type of $P_{red}$ from its general class

```
class Subsetting
Team Person
+ skill:  string [0..*]
Researcher
+ skill:  ResearchSkill [1..*]
+ researchField:  ResearchSkill [1..*]
+ team 0..* {subsets team}
+ member 1..* {subsets member}
+ leaderOf 0..*
{subsets team}
+ leader 0..1
{subsets member}
+ externalTeam 0..*
{subsets team}
+ externalMember 0..*
{subsets member}
```

+ skill:  ResearchSkill [1..*] {redefines skill}
+ researchField:  ResearchSkill [1..*] {subsets skill}
Property Redefining – Constraints

- multiplicity of \( P_{\text{red}} \) is a sub-interval of the multiplicity of \( P \)
Property Redefining – Constraints

- name, visibility and default value of the property can be redefined in any way
Property Navigability

- association R is navigable in an end E from the opposite ends iff E is
  - owned by its class, or
  - navigable owned end of R
Property Navigability

- **anonymous** association is navigable in address from the opposite association end
  - **address** is owned by **Institution**

```
class Association
Institution
+ officialTitle:  String
+ homepage:  URL
Address
+ street:  String
+ city:  String
+ country:  String
1
+address
0..*
```

```
+class 0..1
+ownedAttribute *
```

```
+association
0..1
+association
0..1
{subsets owningAssociation}

+association
0..1
{subsets association}
```

```
+memberEnd 2..*
```

```
+associations
0..1
+associations
0..1
{subsets association}
```

```
+memberEnd
```

```
+navigableOwnedEnd
```

```
+ownedEnd
```

```
+ownedAttribute
```

```
+owningAssociation
0..1
{subsets association}
```

```
+ownedEnd
```

```
+memberEnd
```

```
+associations
0..1
+associations
0..1
{subsets owningAssociation}
```

```
+association
0..1

+association
0..1
{subsets association}
```

```
+memberEnd
```

```
+navigableOwnedEnd
```

```
+ownedEnd
```

```
+ownedAttribute
```

```
+owningAssociation
0..1
{subsets association}
```
Property Navigability

- **anonymous** association is navigable in worker from the opposite association ends
  - worker is navigable owned end of anonymous

```plaintext
<table>
<thead>
<tr>
<th>Class</th>
<th>+class 0..1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>+project 0..1</td>
</tr>
<tr>
<td>Person</td>
<td>+name: String 0..*</td>
</tr>
<tr>
<td>Team</td>
<td>+team 0..1</td>
</tr>
<tr>
<td>Association</td>
<td>+association 0..1</td>
</tr>
<tr>
<td>StructuralFeature</td>
<td>+navigableOwnedEnd</td>
</tr>
<tr>
<td>Property</td>
<td>+ownedEnd</td>
</tr>
<tr>
<td>SMember</td>
<td>+memberEnd 2..*</td>
</tr>
<tr>
<td>+ownedAttribute</td>
<td></td>
</tr>
</tbody>
</table>
```

Formal Foundations of Software Engineering
Aggregations and Compositions

class PropertiesDetail

Property

+ isDerived: boolean = false
+ isReadOnly: boolean = false
+ isDerivedUnion: boolean = false
+ aggregation: AggregationKind = none
+ isID: boolean = false

AggregationKind

- none
- shared
- composite
Aggregations and Compositions

- **aggregation** or, also called, *shared aggregation*
- **part-of relationship**
  - parts can be shared by different owners
Aggregations and Compositions

- **composition** or, also called, composite aggregation
- part-of relationship
  - parts cannot be shared by different owners (exclusive ownership)
  - parts cannot exist without owners
  - upper bound must be 1 (or unspecified)
Class Specialization/Generalization

- Generalization is a taxonomic relationship between a more general class and a more specific class.
- Each instance of the specific class is also an instance of the general class (inheritance).
  - Features specified for instances of the general class are implicitly specified for instances of the specific class.
  - Constraints applying to instances of the general class also apply to instances of the specific class.
Class Specialization/Generalization

```
class SimpleGeneralization
Person
- name: String
- email: String
- phone: String

Student
- schoolWork: URL

Teacher

Researcher
- project: URL [1..*]

Practitioner

Lecture
- title: String

Paper
- title: String
- journal: URL
```

1..* 1..*
Generalization Sets

**Class Diagram**

- **Person**
  - **Gender**: Male, Female
  - **PersonKind**: Student, Teacher, Researcher, Practitioner
  - **Specialization**: Biologist, Chemist, Software Engineer
  - **Gender** relationships: {complete, disjoint}
  - **PersonKind** relationships: {complete, overlapping}

**Diagram Notes**
- The diagram illustrates the relationships between different roles and types of individuals, focusing on gender and professional specialization.
- generalization set can be associated with a class whose instances are specific classes involved in the generalization set

- **What are power types good for?**
Operations

- behavioral feature
  - action that can be performed on class instances

\[ <\text{operation}> ::= \]
\[
[<\text{visibility}>] \ <\text{name}>
\]
\[
\left(\ [<\text{parameter-list}>] \right)'
\]
\[
[:: [<\text{return-type}>] [:: [<\text{multiplicity}>] ]]
\]
\[
\left[:: \ <\text{oper-property}> \ [:: \ <\text{oper-property}>]^* \right]
\]

- \(<\text{oper-property}>\) indicates various properties of the operation
  - redefines \(<\text{oper-name}>\): operation redefines inherited operation
  - query: operation does not change the state of the system (read-only)
  - if max multiplicity > 1 then
    - ordered: return values are ordered
    - unique: return values do not contain duplicities
Operations

- may be supplied with *-conditions
  - pre-condition
    - condition which must be true before the operation is invoked
  - post-condition
    - condition which must be true when the invocation of the operation completes successfully
  - body-condition
    - condition which constraints the return result
    - may be overridden
Operations

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ getName(): String</td>
</tr>
<tr>
<td>+ setName(_n: String)</td>
</tr>
<tr>
<td>+ addAddress(_a: Address, position: int)</td>
</tr>
<tr>
<td>+ getAddresses(): Address [1..3] {ordered}</td>
</tr>
</tbody>
</table>
Enumerations

«enumeration»
DegreeType

researcher
assistant_prof
associated_prof
full_prof

class Conceptual Mo...
Researcher
+ topic: String [1..*]
+ degree: DegreeType = assistant_prof
Stereotypes

- disadvantage of a graphical notation (like UML) is that you have to remember what the symbols mean
- to reduce the number of symbols, UML introduces the notion of stereotypes
  - stereotype is a string keyword
- if you need to model something which is not part of UML but is similar to a UML construct, use that construct and label it with a keyword

<<stereotype>>
Stereotypes

- profile = set of stereotypes suitable for certain purpose, e.g.
  - profile for modeling schemas of relational databases
  - profile for modeling XSD XML Schemas
Class Dependencies

- **class responsibility** (sometimes called *functionality*)
  - each class is responsible for something in the software application

- sometimes a class cannot handle all expected functionality
  - class A asks other classes $B_1, \ldots, B_n$ for help
  - A is called *client*
  - $B_1, \ldots, B_n$ are called *suppliers*
  - relationship between client and supplier is called *dependency*
Class Dependencies

- dependency is a relationship that signifies that a client requires a supplier for its specification or implementation
  - semantics of the client is not complete without the supplier
- modification of the supplier may impact the client
Class Dependencies
Class Dependencies

© http://www.uml-diagrams.org
Interfaces and Abstract Classes

- abstract class is a class that cannot be directly instantiated
  - instantiate their specific classes (if not abstract as well)
  - may or may not have one or more abstract operations
    • but abstract class may have sense even without abstract operations

- interface is a class that has no implementation
  - all its features are abstract
Interfaces and Abstract Classes

```
class InterfacesAbstract
«interface»
Paintable
+ draw() : void
+ color(Color) : void
+ resize(int) : void
Shape
- x: int
- y: int
- color: Color
+ draw() : void
+ color(Color) : void
+ resize(int) : void
+ getArea() : Area
Circle
- radius: int
+ draw() : void
+ resize(int) : void
+ getArea() : Area
«use»
```

Dependency (requires)
Few Recommendations

- strictly distinguish conceptual and implementation level of modeling
- conceptual level is for all stakeholders
  - requirements analysis
- software level is for developers
  - design and implementation
  - programming code, or its skeleton, can be generated
  - different schemas for different kinds of developers (e.g. database, application logic, GUI, ...)
  - see Model-Driven Development
- at conceptual level be scope of
  - navigability
  - visibility
  - operations
  - types may also be off importance