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
Previous Lecture Reminder

```
Person
+ name: String
+ email: String [0..*]
+ phone: String [0..*]
+ registrationDate: Date
+ homepage: URL
+ dblp: URL

Team
+ name: String
+ homepage: URL
+ notify(Project): void

Team
+ principalProject: 0..*
+ principalPartner: 0..1
+ managedTeam: 0..*

Project
+ name: String
+ email: String [0..*]
+ description: String
+ homepage: URL
+ status: int
+ close(): void

Researchers
+ topic: String [1..*]
+ degree: DegreeType = assistant_prof

Institution
+ officialTitle: String
+ homepage: URL

University
+ membership: 0..*

Membership
+ from: date
+ to: date
+ position: Position

Worker
+ manager: 0..1
+ partner: 0..*
+ colleague: 0..*

Worker
+ member: 1..*
{non-unique}

Student
+ manager: 0..*
+ principalPartner: 0..1
+ managedTeam: 0..*

Student
+ employer: 0..*

DegreeType
+ enumeration

researcher

assistant_prof

associated_prof

full_prof
```
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*)
Attributes vs. Associations

- **Institution**
  - instance of **Class**
- **anonymous**
  - instance of **Association**
- **address**
  - instance of **Property**
  - related to **Institution** by **ownedAttribute**
  - or
  - related to **anonymous** by **memberEnd**
  - owned by **Institution**
- **Address**
  - value of **type** (inherited from **TypedElement**)

Attribute & Association End

<table>
<thead>
<tr>
<th>Institution</th>
<th>Institution</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>officialTitle: String</td>
<td>+ officialTitle: String</td>
<td></td>
</tr>
<tr>
<td>homepage: URL</td>
<td>+ homepage: URL</td>
<td></td>
</tr>
<tr>
<td>address: Address [0..*]</td>
<td>+ address: Address [0..*]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Association</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ association 0..1</td>
<td>+ ownedEnd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ association 0..1</td>
<td>+ association 0..1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>+association 0..1</td>
<td>+ class 0..1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TypedElement</th>
<th>TypedElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ownedAssociation</td>
<td>+ownedEnd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StructuralFeature</th>
<th>StructuralFeature</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ownedAssociation</td>
<td>+ownedEnd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute &amp; Association End</th>
<th>Attribute &amp; Association End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Institution</td>
<td>Institution</td>
</tr>
<tr>
<td>officialTitle: String</td>
<td>+ officialTitle: String</td>
</tr>
<tr>
<td>homepage: URL</td>
<td>+ homepage: URL</td>
</tr>
<tr>
<td>address: Address [0..*]</td>
<td>+ address: Address [0..*]</td>
</tr>
</tbody>
</table>
Attributes vs. Associations

- **Person, Institution**
  - instances of **Class**
- **anonymous Association**
  - instance of **Association**
- **employee, employer**
  - instances of **Property**
  - related to **anonymous** by **memberEnd** and **ownedEnd**
  - owned by **anonymous**
- **Institution**
  - value of **type** (inherited from **TypedElement**) of **employer**
- **Person**
  - value of **type** (inherited from **TypedElement**) of **employee**

**Association Ends**

**Diagram:**
- **Person**
  - +employee 1..*
- **Institution**
  - +employer 0..*
  - +association
- **Class**
- **TypedElement**
- **Property**
  - +class 0..1
  - +ownedAttribute *
  - +memberEnd 2..*
- **Association**
  - +association 0..1
  - +navigableOwnedEnd
  - +ownedEnd

**NSWI026 - Software Engineering - Lecture 03**
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
- C₁ = C
if the owner of P is a class C then
  - n = 1
  - C₁ = C
if the owner of P is an association R then
- \( n = |R.\text{memberEnd}| - 1 \)
- \( C_1, \ldots, C_n = R.\text{memberEnd} \setminus P.\text{type} \)
Property Values

- if the owner of P is an association R then
  - \( n = |R.memberEnd| - 1 \)
  - \( C_1, ..., C_n = R.memberEnd \setminus P.type \)
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.isOrdered = true (default false)
  - \{non-unique\} \rightarrow P.isUnique = 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>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

- A common misunderstanding is that only classes can be specialized.
- Specializing attributes and associations is also possible!
  - Via property subsetting and redefining.
Property Subsetting and Redefining

```
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}
```

{redefines skill}
{subsets skill}
Property Subsetting

- set of instances of a property $P_{\text{sub}}$ 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 $P$

$$(\forall C_{sub} \in context(P_{sub}))(\exists C \in context(P))
(C = C_{sub} \lor C \text{ is generalization of } C_{sub})$$
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
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**
Property Navigability

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

```
Person
+ name: String 0..*
+ worker 0..*

Team
+ project 0..1
+ team 0..1
+ worker 0..*

Project
+ project 0..1

Class
+ class 0..1

Association
+ association 0..1
+ owningAssociation 0..1

StructuralFeature Property
+ navigableOwnedEnd
+ ownedEnd
+ ownedAttribute *

Nary
Person
- name: String

Team
Project
+ project 0..1
+ team 0..1
+ worker 0..*

Class
+ class 0..1
```
Aggregations and Compositions

class PropertiesDetail

Property

+ isDerived: boolean = false
+ isReadOnly: boolean = false
+ isDerivedUnion: boolean = false
+ aggregation: AggregationKind = none
+ isID: boolean = false
+ redefinedProperty 0..*
+ property 0..*
+ subsettedProperty 0..*
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..*]

Practicioner

Lecture
- title: String

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

1..* 1..*
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

\[
\text{<operation>} ::= \left[ \begin{array}{l}
\text{<visibility>} \ [ \text{<name>} ] \text{'}
\text{} \left[ \text{'} \left[ \text{<parameter-list>} \right] \text{'} \right]'
\text{'} \left[ \text{'} \left[ \text{<return-type>} \right] \left[ \text{'} \left[ \text{<multiplicity>} \right] \text{'} \right] \right]
\text{'} \left[ \text{'<oper-property>} \left[ \text{'} \left[ \text{<oper-property>} \right] \text{'} \right] \text{'} \right] \right]
\end{array} \right]
\]

- \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
</enumeration>

class Conceptual Model
+ 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 $B_1, ..., B_n$ for help
  - $A$ is called *client*
  - $B_1, ..., B_n$ 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

PIM::Organization
- legalName
- officialNumber

PSM_RELATIONAL::Organization
«column»
* legalName: VARCHAR2(50)
* officialNumber: NUMBER(9)
*PK organizationId: NUMBER(8)
*FK addressId: NUMBER(8)

«PK»
+ FK_Organization_Address(NUMBER)

«unique»
+ UQ_Organization_officialNumbe(NUMBER)

«abstraction»
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

```
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

View
Square
- sizeX: int
- sizeY: int
+ draw() : void
+ resize(int) : void
+ getArea() : Area

Circle
- radius: int
+ draw() : void
+ resize(int) : void
+ getArea() : Area
```

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 (later in this semester)
- at conceptual level be scope of
  - navigability
  - visibility
  - operations
  - types may also be off importance
- non-sense at the conceptual level
  - interfaces