Introduction to Software Engineering

6. Modeling Objects and Classes
Roadmap

- UML Overview
- Classes, attributes and operations
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- Patterns, Constraints and Contracts
Sources


Roadmap

- UML Overview
- Classes, attributes and operations
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- Patterns, Constraints and Contracts
What is UML?

> uniform notation: Booch + OMT + Use Cases (+ state charts)
  — UML is not a method or process
  — … The Unified Development Process is

Why a Graphical Modeling Language?

> Software projects are carried out in team
> Team members need to communicate
  — … sometimes even with the end users
> “One picture conveys a thousand words”
  — the question is only which words
  — Need for different views on the same software artifact
Why UML?

Why UML?

> Reduces *risks* by documenting assumptions
  — domain models, requirements, architecture, design, implementation …

> Represents industry *standard*
  — more tool support, more people understand your diagrams, less education

> Is reasonably *well-defined*
  — ... although there are interpretations and dialects

> Is *open*
  — stereotyes, tags and constraints to extend basic constructs
  — has a meta-meta-model for advanced extensions
UML History

> 1994: Grady Booch (Booch method) + James Rumbaugh (OMT) at Rational
> 1994: Ivar Jacobson (OOSE, use cases) joined Rational — “The three amigos”
> 1996: Rational formed a consortium to support UML
> 1997: UML1.0 submitted to OMG by consortium
> 1997: UML 1.1 accepted as OMG standard — However, OMG names it UML1.0
> 1998-…: Revisions UML1.2 - 1.5
> 2005: Major revision to UML2.0, includes OCL
Roadmap

- UML Overview
- **Classes, attributes and operations**
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- Patterns, Constraints and Contracts
“Class diagrams show generic descriptions of possible systems, and object diagrams show particular instantiations of systems and their behaviour.”

Attributes and operations are also collectively called features.

**Danger:** class diagrams risk turning into data models. Be sure to focus on behaviour.
Visibility and Scope of Features

Stereotype
(what “kind” of class is it?)

Underlined attributes have class scope

Don’t worry about visibility too early!

User-defined properties
(e.g., readonly, owner = “Pingu”)

Italic attributes are abstract

An ellipsis signals that further entries are not shown

+ = “public”
# = “protected”
- = “private”
Attributes and Operations

Attributes are specified as:

name: type = initialValue { property string }

Operations are specified as:

name (param: type = defaultValue, ...) : resultType
Roadmap

- UML Overview
- Classes, attributes and operations
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- Patterns, Constraints and Contracts
UML Lines and Arrows

- Constraint (usually annotated)
- Association e.g., «uses»
- Dependency e.g., «requires», «imports» ...
- Navigable association e.g., part-of
- Realization e.g., class/template, class/interface
- "Generalization" i.e., specialization (!) e.g., class/superclass, concrete/abstract class
- Aggregation i.e., “consists of”
- "Composition" i.e., containment
Roadmap

- UML Overview
- Classes, attributes and operations
- UML Lines and Arrows
- **Parameterized Classes, Interfaces and Utilities**
- Objects, Associations
- Inheritance
- Patterns, Constraints and Contracts
Parameterized Classes

Parameterized (aka “template” or “generic”) classes are depicted with their parameters shown in a **dashed box**.

![Diagram of Parameterized Classes]

- **T**, k: `Integer` are the template parameters.
- T has type `Classifier` by default.
- The parameters are used in the template body.
- In this template, the multiplicity of the array is fixed by the binding.
- Implicit binding. This class has an anonymous name.
- Explicit binding: «bind» (Address, 24)
- FArray<Point, 3>
- AddressList

**Figure 13-180.** Template notation with use of parameter as a reference.
Interfaces

Interfaces, equivalent to abstract classes with no attributes, are represented as classes with the stereotype «interface» or, alternatively, with the “Lollipop-Notation”:

![Diagram showing the representation of interfaces.](image)

**Figure B-5.** Realization of an interface
Utilities

A utility is a grouping of global attributes and operations. It is represented as a class with the stereotype «utility». Utilities may be parameterized.

<table>
<thead>
<tr>
<th>«Utility»</th>
</tr>
</thead>
<tbody>
<tr>
<td>MathPack</td>
</tr>
<tr>
<td>randomSeed : long = 0</td>
</tr>
<tr>
<td>pi : long = 3.14158265358979</td>
</tr>
<tr>
<td>sin (angle : double) : double</td>
</tr>
<tr>
<td>cos (angle : double) : double</td>
</tr>
<tr>
<td>random () : double</td>
</tr>
</tbody>
</table>

return sin (angle + pi/2.0);

NB: A utility’s attributes are already interpreted as being in class scope, so it is redundant to underline them.

A “note” is a text comment associated with a view, and represented as box with the top right corner folded over.
Roadmap

> UML Overview
> Classes, attributes and operations
> UML Lines and Arrows
> Parameterized Classes, Interfaces and Utilities
> Objects, Associations
> Inheritance
> Patterns, Constraints and Contracts
Objects are shown as rectangles with their name and type underlined in one compartment, and attribute values, optionally, in a second compartment.

![Diagram of an object notation example]

**Figure 13-134. Object notation**

*At least one of the name or the type must be present.*
**Associations** represent structural relationships between objects

— usually *binary* (but may be ternary etc.)
— optional *name* and *direction*
— (unique) *role names* and *multiplicities* at end-points

![Diagram](image)

*Figure 4-2. Association notation*
The multiplicity of an association constrains how many entities one may be associated with.

- Examples:

<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>Zero or one entity</td>
</tr>
<tr>
<td>1</td>
<td>Exactly one entity</td>
</tr>
<tr>
<td>*</td>
<td>Any number of entities</td>
</tr>
<tr>
<td>1..*</td>
<td>One or more entities</td>
</tr>
<tr>
<td>1..n</td>
<td>One to n entities</td>
</tr>
</tbody>
</table>

And so on …
Associations and Attributes

> Associations may be implemented as attributes
— But need not be …
Aggregation is denoted by a *diamond* and indicates a *part-whole dependency*:

A *hollow diamond* indicates a *reference*; a *solid diamond* an *implementation* (i.e., ownership).

**Aggregation:** parts may be shared.

**Composition:** one part belongs to one whole.
An association may be an instance of an association class:

In many cases the association class only stores attributes, and its name can be left out.
Qualified Associations

A **qualified association** uses a special *qualifier value* to identify the object at the other end of the association.

*NB: Qualifiers are part of the association, not the class*

![Qualified Association Diagram](image)

**Figure 4-4.** Qualified association
Roadmap

- UML Overview
- Classes, attributes and operations
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- Patterns, Constraints and Contracts
Generalization

A **subclass** specializes its superclass:

![Diagram](image)

Figure 4-7. Generalization notation
What is Inheritance For?

> New software often builds on old software by *imitation*, *refinement* or *combination*.

> Similarly, classes may be *extensions*, *specializations* or *combinations* of existing classes.
Generalization expresses ...

**Conceptual hierarchy:**
> conceptually related classes can be organized into a *specialization* hierarchy
  — people, employees, managers
  — geometric objects ...

**Polymorphism:**
> objects of distinct, but related classes may be *uniformly treated* by clients
  — array of geometric objects

**Software reuse:**
> related classes may *share* interfaces, data structures or behaviour
  — geometric objects ...
The different faces of inheritance

- Is-a
- Polymorphism
- Reuse
Roadmap

- UML Overview
- Classes, attributes and operations
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- Patterns, Constraints and Contracts
Design Patterns as Collaborations

The CallQueue class plays the subject role in the collaboration.

CallQueue
- queue: List of Call
- source: Object
- waitAlarm: Alarm
- capacity: Integer

Observer

binding of the Observer pattern

{ handler.reading = length (subject.queue)
  range = (0 .. capacity) }

SlidingBarIcon
- reading: Real
- color: Color
- range: Interval

The SlidingBarIcon class plays the handler role.

some constraints on the pattern

Figure 13-144. Binding of a pattern to make a collaboration
**Constraints**

Constraints are *restrictions* on values attached to classes or associations.

**Figure 4-12. Constraints**
OCL — Object Constraint Language

> Used to express queries and constraints over UML diagrams
  — Navigate associations:
    – `Person.boss.employer`
  — Select subsets:
    – `Company.employee->select(title="Manager")`
  — Boolean and arithmetic operators:
    – `Person.salary < Person.boss.salary`
Combine constraints with stereotypes:

*NB: «invariant», «precondition», and «postcondition» are predefined in UML.*

**Figure 13-147. Precondition**

**Figure 13-145. Postcondition**
Using the Notation

**During Analysis:**
- Capture classes visible to *users*
- Document *attributes and responsibilities*
- Identify *associations and collaborations*
- Identify *conceptual hierarchies*
- Capture all *visible features*

**During Design:**
- Specify *contracts and operations*
- *Decompose* complex objects
- Factor out *common interfaces* and functionalities

*The graphical notation is only one part of the analysis or design document. For example, a data dictionary cataloguing and describing all names of classes, roles, associations, etc. must be maintained throughout the project.*
What you should know!

> How do you represent classes, objects and associations?
> How do you specify the visibility of attributes and operations to clients?
> How is a utility different from a class? How is it similar?
> Why do we need both named associations and roles?
> Why is inheritance useful in analysis? In design?
> How are constraints specified?
Can you answer the following questions?

- Why would you want a feature to have class scope?
- Why don’t you need to show operations when depicting an object?
- Why aren’t associations drawn with arrowheads?
- How is aggregation different from any other kind of association?
- How are associations realized in an implementation language?
License

Attribution-ShareAlike 3.0 Unported

You are free:
- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:
- Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.

http://creativecommons.org/licenses/by-sa/3.0/