2: Modeling Class Architecture with UML Class Diagrams

Slide adapted from Eran Toch's lecture series

System Development Process

<table>
<thead>
<tr>
<th>Phase</th>
<th>Actions</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiation</td>
<td>Raising a business need</td>
<td>Business documents</td>
</tr>
<tr>
<td>Requirements</td>
<td>Interviewing stakeholders, exploring the system environment</td>
<td>Organized documentation</td>
</tr>
<tr>
<td>Analysis &amp; Specification</td>
<td>Analyze the engineering aspect of the system, building system concepts</td>
<td>Logical System Model</td>
</tr>
<tr>
<td>Design</td>
<td>Define architecture, components, data types, algorithms</td>
<td>Implementation Model</td>
</tr>
<tr>
<td>Implementation</td>
<td>Program, build, unit-testing, integrate, documentation</td>
<td>Testable system</td>
</tr>
<tr>
<td>Testing &amp; Integration</td>
<td>Integrate all components, verification, validation, installation, guidance</td>
<td>Testing results, Working sys</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Bug fixes, modifications, adaptation</td>
<td>System versions</td>
</tr>
</tbody>
</table>

Outline

- Introduction
- Classes, attributes and operations
- Relations
- Generalization
- Guidelines for effective class modeling

Logical System Models

- Logical system models depict what a system is or what a system must do—not how the system will be implemented.
  - Process models (e.g., data flow diagrams)
  - Data models (e.g., entity relationship diagrams)
  - Interface models (e.g., context diagrams)
  - Object models (e.g., Uniform Modeling Language diagrams)
- They depict it using a modeling language
  - Formal (or partially formal)
  - Understandable (visual or textual)
Modeling Approaches

Modeling approaches differ from each other according to their view of the world

<table>
<thead>
<tr>
<th>Object-Oriented</th>
<th>Process-Oriented</th>
<th>State-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focused on objects, which are concrete elements, combining information and actions</td>
<td>Focused on processes, which are patterns of transformation (of something). Processes can be concrete or abstract</td>
<td>Focused on the different states – values and status of the system, and how and why these states change.</td>
</tr>
</tbody>
</table>

Design Process

From Requirements to Structure

1. Administrator enters course name, code and description
2. System validates course code
3. System adds the course to the data base and shows a confirmation message

Requirements Document

Structure (what’s the constant things of the system)

A structural design defines the artifact unchanging characteristics, which do not change over time.
Structural Modeling in Information Systems

- Static structure of the model
  - the entities that exist (e.g., classes, interfaces, components, nodes)
  - relationship between entities
  - internal structure

- Do not show
  - temporal information
  - behavior
  - runtime constraints

Outline

- Introduction
- Classes, attributes and operations
  - Relations
  - Generalization
  - Guidelines for effective class modeling

Object-Oriented Approach

- Objects are abstractions of real-world or system entities

Class Name
attributes
operations

A class is a template for actual, in-memory, instances

Intro | Classes | Relations | Generalization | Guidelines

Intro | Classes | Relations | Generalization | Guidelines

Intro | Classes | Relations | Generalization | Guidelines
Attributes - Signature

[visibility] name [([multiplicity]) [: type] [=initial value] [[{property}]]

- visibility: the access rights to the attribute
- multiplicity: how many instances of the attribute are they:
  - middleName [0..1] : String, phoneNumber [1..*]
- type: the type of the attribute (integer, string, Person, Course)
- initial value: a default value of the attribute
  - salary : Real = 10000, position : Point = (0,0)
- property: predefined properties of the attribute
  - Changeable, readOnly, addOnly, frozen (C++: const, Java: final)

Operations - Signature

[visibility] name [([parameter-list])] [: return-type] [[{property}]]

- An operation can have zero or more parameters, each has the syntax:
  - [direction] name : type [=default-value]
  - direction can be: in (input parameter - can’t be modified), out (output parameter - may be modified), inout (both, may be modified)
- Property:
  - {leaf} – concrete operation
  - {abstract} – cannot be called directly
  - {isQuery} – operation leaves the state of the operation unchanged
  - ...

Attributes - Examples

+ isLightOn : boolean = false
- numOfPeople : int

mySport
+ passengers : Customer[0..10]
- id : long {readOnly}

Operations - Examples

+ isLightOn() : boolean
+ addColor(newColor : Color)
+ addColor(newColor : Color) : void
# convertToPoint(x : int, y : int) : Point
- changeItem([in] key : string, [out] newItem : Item) : int

What’s the difference?
Visibility

- public (+) – external objects can access the member
- private (-) – only internal methods can access the member
- protected(#) – only internal methods, or methods of specialized objects (descendants) can access the member

We will try to keep the visibility as minimal as possible

Object Diagram

- In an Object Diagram, class **instances** can be modeled

Full Blown Class

```
<<abstract>>
Window
{transient status=tested}
+size:Area = (100,100)
#visibility:Boolean = invisible
+default-size:Rectangle
#maximum-size:Rectangle
-xptr:XWindow
+display()
+hide()
+create()
-attachXWindow(xwin:Xwindow*)
```

Outline

- Introduction
- Classes, attributes and operations
  - Relations
  - Generalization
- Guidelines for effective class modeling
Intro | Classes | Relations | Generalization | Guidelines

## Relations

- A relation is a template for a connection between two instances.
- Relations are organized in a hierarchy:
  - Dependency: dynamic relations
  - Associations: consistent relations
  - Composition: whole-part relations

### Associations

- Objects on both sides of the association can find each other
- The relation is consistent in time (unless removed)

### Navigation

- If an association is directed, messages can pass only on that direction
- If the association does not have directions, then it’s a bidirectional association
- By default, all relations should be **directed**, unless the requirements dictate a bidirectional relation

### Multiplicity

- Indicates cardinality
  - `1:1` default
  - `3` – exactly 3 object
  - `*(or n)` – unbounded
  - `1..*` – 1 to eternity
  - `3..9` – 3 to 9

---

Intro | Classes | Relations | Generalization | Guidelines
Association Classes

Denoted as a class attached to the association, and specify properties of the association.

An association class is a “normal” class, and may include relations, inheritance etc.

According to the requirements, each product can appear in several orders, and each order may include several products.

Class Normalization

- Classes should be normalized, if:
  1. Attributes are selected from large or infinite sets
  2. Relations with attributes are in n:n form
  3. Groups of attributes are related

Relations & Attributes

- Relations are denoted with associations, not attributes.
- Implementation (pointers, arrays, vectors, ids etc) is left to the detailed design phase.
Role Names

- Names may be added at each end of the association
- Provide better understanding of the association meaning
- Especially helpful in self-associated classes

```
Worker * Person +..1 Manages Company
 1..0 employee employer
```

Qualifiers

- A qualifier is an attribute or list of attributes whose values serve to partition the set of objects associated with an object across an association

```
College
  Student ID 1..*
    Student

Chessboard
  rank : int
  file : int
    Square
```

- The qualifier limits the multiplicity of the target object according to the qualifier attribute. Thus, even though a Bank has many persons, it has one or zero person with a particular account #

Ternary Associations

```
Year
  season *

Team * team
  goalkeeper

Record
  goals for
  goals against
  wins
  losses
  ties
```

Constraints

- Constrains are simple properties of associations, classes and many other things in UML
- Specify limitations that implementers need to satisfy

```
Window
  length
  width
  {0.8 ≤ length/width ≤ 1.5}

Dictionary
  Language 1 {ordered}
    Word
```

Denotes explicit order of instance
**Constraints**

- Constraints can be applied to almost every element in UML diagrams, using:
  - natural language
  - mathematical notation
  - OCL (Object Constraint Language)

- Expressing:
  - Invariants: interest > 3%
  - Preconditions: before loan() takes place, salary > 5,000$
  - Postconditions: after loan() takes place, dayCollect = 1 or 10

See [http://www.klasse.nl/ocl/index.html](http://www.klasse.nl/ocl/index.html)

---

**Dependency**

- Notated by a dotted line
- The most general relation between classes
- Indicates that an object affects another object

**AccountingSystem** creates a **Receipt** object

---

**Dependency – cont’d**

- Dependencies are the most abstract type of relations.
- Properties:
  - Dependencies are always directed (If a given class depends on another, it does not mean the other way around).
  - Dependencies do not have cardinality.
- If instances of two classes send messages to each other, but are not tied to each other, then dependency is appropriated.
- Types:
  - «call»
  - «create»
### Aggregation

- “Whole-part” relationship between classes
- Assemble a class from other classes
  - Combined with “many” - assemble a class from a couple of instances of that class

#### Composition vs. Aggregation

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part can be shared by several wholes</td>
<td>Part is always a part of a single whole</td>
</tr>
<tr>
<td>Parts can live independently (i.e., whole cardinality can be 0..*)</td>
<td>Parts exist only as part of the whole. When the wall is destroyed, they are destroyed</td>
</tr>
<tr>
<td>Whole is not solely responsible for the object</td>
<td>Whole is responsible and should create/destroy the objects</td>
</tr>
</tbody>
</table>

#### Composition

- Composition is a stronger form of aggregation
- Contained objects that live and die with the container
- Container creates and destroys the contained objects

#### Outline

- Introduction
- Classes, attributes and operations
- Relations
  - Generalization
- Guidelines for effective class modeling
Generalization – Definitions

• Super Class (Base class)
  – Provides common functionality and data members

• Subclass (Derived class)
  – Inherits public and protected members from the super class
  – Can extend or change behavior of super class by overriding methods

• Overriding
  – Subclass may override the behavior of its super class

Generalization – advantages

• Modularity:
  – Eliminate the details
  – Find common characteristics among classes
  – Define hierarchies

• Reuse:
  – Allow state and behavior to be specialized

Generalization Guidelines

• Look carefully for similar properties between objects, sometimes they are not so obvious
Generalization – cont’d

**Generalization**

- ID and name are common to all classes
- Association is the same as any other attribute

---

**Abstract Class**

- A class that has no direct instances
  - Denoted by an *Italic* name
  - Or by the stereotype “abstract”

---

**Models and Sets**

---

**Generalization and Sets**

---
**What Relations are Missing?**

- **Union**
  - We cannot define a class such as:
    
    \[
    \text{allPeopleInTheCollege} = \text{students} \cup \text{Staff}
    \]

- **Complementary**
  - We cannot create classes which take some of the super-class properties but omit some of them:
    
    \[
    \text{MultiMedia} = \text{Content} \setminus \text{Textual}
    \]

**Dynamic Relations**

- **Dynamic Intersection**
  - We cannot create classes by dynamically intersecting between class properties

**Encapsulation & Information Hiding**

- Encapsulation is the separation between the external aspects of an object and its internals
- An **Interface** is:
  - A collection of method definitions for a set of behaviors – a "contract".
  - No implementation provided.

**Interface Terminology**

- Realization relation:  

\[ \text{Realizes} \]

\[ \text{Class} \]

\[ \begin{align*}
\text{Operation 1 Impl'} \\
\text{Operation 2 Impl'}
\end{align*} \]

\[ \text{Interface} \]

\[ \begin{align*}
\text{Operation 1 Declaration} \\
\text{Operation 2 Declaration}
\end{align*} \]

\[ \text{Realizes} \]

\[ \text{External Object} \]

\[ \begin{align*}
\text{Operation 1 Decl.} \\
\text{Operation 2 Decl.}
\end{align*} \]

\[ \text{Data} \]

\[ \text{Operation 1 Impl'} \]

\[ \text{Operation 2 Impl'} \]
Example: Microsoft’s Common Object Model

Interfaces Notation

Realization:
The object guarantees to carry out the contract specified by the interface.

Outline

- Introduction
- Classes, attributes and operations
- Relations
- Generalization & Encapsulation
- Guidelines for effective class modeling
How to Model?

**Bottom-up Process**
Starting with throwing all classes on the page, and then combining them:

**Top-down Process**
Starting with an overview of the system, and then splitting classes

---

CRC Cards

- CRC Cards: 
  - Class, Responsibility, Collaboration

---

Guidelines for Effective Class Diagram

- **Identifying classes**
  - Very similar to identifying data repositories in DFD. Identify data elements, and model them.
  - Plus, think of classes that handle processes. If operations are complicated enough, we might want to model them separately.
  - Plus, think of the actors. Are all their needs covered by existing operations?

---

General Assumptions

- **Access**
  - Users can execute any public operation of the classes (except when using specialized stereotypes).

- **Lifespan**
  - Objects (except transient objects) have an endless life span.
  - We don’t need to bother with their serialization.

- **Simplicity**
  - No need for get / set values.
  - No need for constructors / distracters.
Finding Objects

• Objects can be found, browsed through and located without any aggregating class.

That's enough for Loan Service to access all instances of Book

Guidelines – Modeling Actors

• A common mistake is to model actors as classes
• Remember -
  – Actors interact with the system directly, they don’t need to be represented a priori
  – Sometimes, the system saves data about customers, but it does not mean that they do all their actions through this class

Guidelines – User Interfaces

• If the user has complicated interactions with the system, then we may want to dedicate a special class as a “user interface”
• Remember – it’s not the same class as the class that contains data about the actor

Guidelines – Modeling Actors

• A common mistake is to model actors as classes
• Remember -
  – Actors interact with the system directly, they don’t need to be represented a priori
  – Sometimes, the system saves data about customers, but it does not mean that they do all their actions through this class

Summary

✓ Introduction
  – Structural modeling
✓ Classes
  – Attributes and operations
✓ Relations
  – Associations, constraints
  – Dependencies, compositions
✓ Generalization
  – Inheritance
  – Interfaces
✓ Object Diagrams
✓ Guidelines for effective class modeling