UML models: Object-orientation and Design Patterns

Software Engineering Theory

Kristian Sandahl (slides by many people)
Department of Computer and Information Science
2014-09-16
Requirements

System Design
(Architecture, High-level Design)

Module Design
(Program Design, Detailed Design)

Implementation of Units (classes, procedures, functions)

Acceptance Test
(Release testing)

System Testing
(Integration testing of modules)

Module Testing
(Integration testing of units)

Unit testing

Validate Requirements, Verify Specification

Verify System Design

Verify Module Design

Verify Implementation

Maintenance

Project Management, Software Quality Assurance (SQA), Supporting Tools, Education
The goals of module design

• Provide the expected function
• Prepare for change:
  • Separation of concern
  • Testability
  • Understandability
• Contribute to quality, eg:
  • Performance
  • Usability
  • Reliability
  • ...
• Map for the implementers and testers
Modelling software

- Models supplement natural language
- Models support both elicitation and design
- The boundaries between specification and design have to be decided
- There are high transition costs from functional to object-oriented models
- **UML** has become the standard notation
- Industry interest in SySML (watch out in the future)
A Single Class

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String[1]</td>
<td>+getNoOfOrders():Integer</td>
</tr>
<tr>
<td>email: String [0..2]</td>
<td>+getOrderStatus():String</td>
</tr>
<tr>
<td></td>
<td>+ addEmail(email: String)</td>
</tr>
</tbody>
</table>

Visibility:
- + public
- - private
- # protected
- ~ package

Multiplicity:
- 1 exactly one
- 0..1 Zero or one
- * Zero or more
  (same as 0..*)
- 2..8 Between 2 and 8
Relationships (1/6) - overview and intuition
- Association

A \rightarrow B

Association
(with navigability)
Relationships (1/6) - overview and intuition - Association

Both representations are almost equivalent

Date
+ newOrderDate: Date [0..1]

BookTitle
+ newOrderDate: Date [0..1]
+ isLatestEdition: Boolean [1]
+ items: BookItem[*]

BookItem
+ items {ordered}*

Boolean
+ isLatestEdition

Default is unordered, unique
Relationships (1/6) - overview and intuition

- Association

*Car*

class

objects

mycar

Explicitly show that navigation is not allowed

Navigation - mycar can reach the wheels, but not the opposite

mycar has links to 4 wheels
What does it mean to have a * here? What if we have multiplicity 1 instead?

"*

A wheel can only be liked to one car instance

"1"

A wheel can be linked to more than one car instance
Relationships (1/6) - overview and intuition
- Association

Associations are the "glue" that ties a system together

association instance = link

An association describes a relation between objects at run-time.

{(mycar1,wheel1),
 (mycar1,wheel2),
 (mycar1,wheel3),
 (mycar1,wheel4)}
**Relationships (2/6) - overview and intuition**

**- Aggregation**

- **Association** (with navigability)

  "A" has a reference(s) to instance(s) of "B". Alternative: attributes

- **Aggregation**
Relationships (2/6) - overview and intuition
- Aggregation

Common vague interpretations: "owns a" or "part of"

What does this mean? What is the difference to association?

Vague definitions  → Inconsistency and misunderstandings

Aggregation was added to UML with little semantics. Why?

Jim Rumbaugh
"Think of it as a modeling placebo"

**Recommendation:**
- Do not use it in your models.
- If you see it in other's models, ask them what they actually mean.
Relationships (3/6) - overview and intuition
- Composition

**Association** *(with navigability)*
"A" has a reference(s) to instance(s) of "B". Alternative: attributes

**Aggregation**
Avoid it to avoid misunderstandings

**Composition**
Yes! First, multiplicity must be 1 or 0..1. An instance can only have one owner.

But, isn't this equivalent to what we showed with associations?

Well, in this case...
Using composition...

- Composition

Car -> Wheel
Wheel -> MotorCycle

- Ok for wheels to be part of mycar1 or mybike1

mycar1 -> wheel1, wheel2, wheel3, wheel4
mybike1 -> wheel5, wheel6
Using composition...

Can mycar1 and mybike1 share the same wheels?

**NO!**
Not with composition!

**Key concepts**
- "No sharing" rule
- The owner is responsible for managing its parts, e.g. allocation and deallocation.
Using associations...

(Note the difference. The diamond is removed.)

Can mycar1 and mybike1 share the same wheels this time?

Yes! Associations do not have a "no sharing" rule.

However, in this case it is a strange model...
Relationships (4/6) - overview and intuition
- Generalization

**Association** (with navigability)  "A" has a reference(s) to instance(s) of "B". Alternative: attributes

**Aggregation**  Avoid it to avoid misunderstandings

**Composition**  An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

**Generalization**
1. **Inheritance** ~ relation implementation

- **Vehicle**
  - `+ drive()`
- **Car**
  - `+ reverse()`
- **MotorCycle**
  - `+ drive()`

**Overrides** `drive()`

- **Inherits** the code for `drive()`. New operation `reverse()`

---

**An instance of a class can have many types** = (subtyping) **polymorphism**

2. **Subtyping** ~ relation on interfaces

**Visible Type**: Vehicle.
- Instance of: MotorCycle.
  - Can we `drive()`? Can we `reverse()`?

**Visible Type**: Car.
- Instance of: Car
  - Can we `drive()`? Can we `reverse()`?

**Visible Type**: Vehicle.
- Instance of: Car
  - Can we `drive()`? Can we `reverse()`?

**static typing**: safe substitution
Relationships - (5/6) overview and intuition

- **Realization**
  - Alternative: attributes

- **Association** (with navigability)
  - "A" has a reference(s) to instance(s) of "B"

- **Aggregation**
  - Avoid it to avoid misunderstandings

- **Composition**
  - An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

- **Generalization**
  - 1) "A" inherits all properties and operations of "B"
  - 2) An instance of "A" can be used where an instance of "B" is expected

- **Realization**
**Realization**

- provides a specified interface

---

**Interface** (no implementation)

<table>
<thead>
<tr>
<th>Door</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ open()</td>
</tr>
</tbody>
</table>

**Specifier**

<table>
<thead>
<tr>
<th>Vehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ drive()</td>
</tr>
</tbody>
</table>

**Implementation**

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ drive()</td>
</tr>
<tr>
<td>+ reverse()</td>
</tr>
<tr>
<td>+ open()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MotorCycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ drive()</td>
</tr>
</tbody>
</table>

**Realization**

Can we create an instance of Vehicle?

<table>
<thead>
<tr>
<th>Vehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ drive()</td>
</tr>
</tbody>
</table>

Can we create an instance of AnotherVehicle?

<table>
<thead>
<tr>
<th>AnotherVehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ drive()</td>
</tr>
<tr>
<td>+ open()</td>
</tr>
</tbody>
</table>

---

Provides the **Door** interface

Must implement the interface

Abstract class (Italic)

Abstract operation
What is the difference between an interface and an abstract class?

An abstract class with only abstract operations is conceptually the same as an interface.
### Relationships - (6/6) overview and intuition

#### Realization

"A" provides an implementation of the interface specified by "B".

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

#### Association (with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

#### Aggregation

Avoid it to avoid misunderstandings

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

#### Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7.png" alt="Diagram" /></td>
<td><img src="image8.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

#### Generalization

1) "A" inherits all properties and operations of "B".

2) An instance of "A" can be used where a instance of "B" is expected.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image9.png" alt="Diagram" /></td>
<td><img src="image10.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

#### Dependency

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image11.png" alt="Diagram" /></td>
<td><img src="image12.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Relationships - (6/6) overview and intuition
- Dependency

Schedule viewer

Lecture

client

Dependency

supplier

<<use>>
Relationships - overview and intuition

**Association**

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

**Aggregation**

Avoid it to avoid misunderstandings

**Composition**

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

**Generalization**

1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a instance of "B" is expected.

**Realization**

"A" provides an implementation of the interface specified by "B".

**Dependency**

"A" is dependent on "B" if changes in the definition of "B" causes changes of "A".
Software Design Patterns

A Design Pattern is a standard solution for a standard design problem in a certain context.

Goal: reuse design information
Example: Facade
Example: Facade
How to describe design patterns?
Facade

**Intent**

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

**Motivation**

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. … example …
Facade

Applicability

Use the Facade pattern when:

• you want to provide a simple interface to a complex subsystem. This makes subsystems more reusable and easier to customize.

• there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from other subsystems, thereby promoting subsystem independence and portability.

• you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.
Facade

Consequences

The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making subsystem easier to use.

2. It promotes weak coupling between subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.

3. It doesn't prevent applications from using subsystem classes if they need to.
Facade

- Structure
- Participants
- Collaborations
- Implementation
- Sample Code
- Known Uses
- Related Patterns
Observer

\[ a = 10\% \]
\[ b = 30\% \]
\[ c = 40\% \]
Observer

Applicability

• When an abstraction has two aspects, one dependent on the other.
• When a change to one object requires changing others.
• When an object should be able to notify other objects without making assumptions about who these objects are.
Observer, structure

Subject

Stock
- attach(Observer)
- detach(Observer)
- notify()

Investor
- update()

IBM
- subjectState
- getState()
- setState()

Goldman Sachs
- observerState
- update()

Observer

ConcreteSubject

ConcreteObserver
Observer, collaborations

Diagram showing the relationships and interactions between a concrete subject and two concrete observers, highlighting methods such as `setState()`, `notify()`, `update()`, and `getState()`.
Observer, consequences

- Abstract coupling between Subject and Observer
- Support for broadcast communication
- Unexpected updates
**Strategy**

**Name:** Strategy  
**Also known as:** Policy

**Problem:**
- Need to use **different variants** of the same algorithm in a class
- Different algorithms will be appropriate at **different time**.
- It is hard to **add new algorithms** and to change existing ones.

**Example:**

```
Input (Plain Text)  
Cryptographic Module  
Output (cipher text)
```

**Algorithms:**
- AES
- DES
- 3DES
- RC5

**Intent (from GoF):**
"Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."
Strategy

**Structure:**

- **Context**
  - +contextInterface()

- **Strategy**
  - -strategy *
  - +algorithmInterface()

In example: Part of crypto module. Holds data, keys etc.

- **ConcreteStrategyA**
  - +algorithmInterface()
- **ConcreteStrategyB**
  - +algorithmInterface()
- **ConcreteStrategyC**
  - +algorithmInterface()

Reference to a strategy type

Abstract

In example: e.g. class EncryptAlg

In Example: Implements e.g. algorithm AES

E.g. AlgDES

E.g. AlgRC5
Strategy

• Suppose we add a new strategy:
• Storage media:
  • Disc
  • USB-stick
  • DVD
  • Cloud
  • ....

Input (Plain Text)

Cryptographic Module

Output (Cipher text on media)
Two strategies

Backup
+EncryptInt()
+StoreInt()

EncryptAlg
+Encrypt()

Encrypted

AES
+Encrypt()

AlgDES
+Encrypt()

AlgRC5
+Encrypt()

Media
+Store()

Disc
+Store()

DVD
+Store()

Cloud
+Store()