Module Design and UML

Lecture 6

Software Engineering
TDDC88/TDDC93
autumn 2013

Slides by David Broman and Kristian Sandahl
Department of Computer and Information Science
Linköping University, Sweden
kristian.sandahl@ida.liu.se
A Software Life-cycle Model
Which part will we talk about today?

Validate Requirements, Verify Specification

Requirements

System Design
(Architecture, High-level Design)

Module Design
(Program Design, Detailed Design)

Implementation
of Units (classes, procedures, functions)

Verify Module Design

Module Testing
(Integration testing of units)

Verify System Design

System Testing
(Integration testing of modules)

Verify Implementation

Unit testing

Acceptance Test
(Release testing)

Maintenance

Validate Requirements, Verify Specification

Verify Module Design

Verify System Design

Validate Requirements, Verify Specification

Project Management, Software Quality Assurance (SQA), Supporting Tools, Education

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
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
Agenda - What will you learn today?

Part I
Classes and Objects

Part III
State Machines and Sequence Diagrams
Part I
Classes and objects
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 too (watch out in the future)
A Single Class

Class name: **Customer**

**Attributes:**
- `name: String[1]`
- `email: String [0..2]`

**Operations:**
- `+getNoOfOrders():Integer`
- `+getOrderStatus():String`
- `+addEmail(email:String)`

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

Both representations are almost equivalent.

- **attributes**

- **role name**

- **directed association**

- {ordered} {unordered}
  - {unique} {nonunique}
  - Default is unordered, unique
Relationships (1/6) - overview and intuition
- Association

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 linked to **one** car instance

A wheel can be linked to **more than one** car instance
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) }
Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams

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

A

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

B

Aggregation

A

B
A major source of confusion
Common vague interpretations: "owns a" or "part of"

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

Vague definitions \[\rightarrow\] 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**
Any difference to association?

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...

Car

Wheel

MotorCycle

Ok for wheels to be part of mycar1 or mybike1

mycar1

wheel1 wheel2 wheel3 wheel4

mybike1

wheel5 wheel6

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
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.
Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams

---

Relationships (3/6) - overview and intuition
- Composition

Using associations...

Car

Wheel

MotorCycle

1 4

2 1

(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...

mycar1

wheel1 wheel2 wheel3 wheel4

mybike1

wheel5 wheel6

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
Instance models

Specific

Kristian : CoffeeCustomer  
buys  
Cup1 : CupOfCoffee  
makes  
IDA-B-house : Machine

Generic

aCoffeeCustomer : CoffeeCustomer  
buys  
aCup : CupOfCoffee  
makes  
aMachine : Machine

Short hand

: CoffeeCustomer  
buys  
: CupOfCoffee  
makes  
: Machine

Related: Roles

: CoffeeCustomer  
buys  
: CupOfCoffee  
makes  
: Machine
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**
**Part I**
Classes and Objects

1. **Inheritance**
~ relation implementation

- **Vehicle**
  - + drive()

  - **Car**
    - + reverse()

    - **MotorCycle**
      - + drive()

    Overrides drive()

  Inherits the code for drive(). New operation reverse()

2. **Subtyping**
~ relation on interfaces

- **Vehicle**
  - + drive()

  - **MotorCycle**
    - + drive()

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()? reverse() is not visible!

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

**Part II**
State Machines and Sequence Diagrams
Relationships - (5/6) overview and intuition - Realization

**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) "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" has a reference(s) to instance(s) of "B". Alternative: attributes
Relationships - (5/6) overview and intuition - Realization

Realization
~ provides a specified interface

Can we create an instance of Vehicle? Yes! It is concrete.

Can we create an instance of AnotherVehicle? No!

- Interface (no implementation)
  - Door
    - + open()
  - Vehicle
    - + drive()
  - Specifier

- Implementation
  - Car
    - + drive()
    - + reverse()
    - + open()
  - MotorCycle
    - + drive()
  - Must implement the interface

- Abstract class (Italic)
  - AnotherVehicle
    - + drive()
    - + open()
  - Abstract operation

Provides the Door interface

Must implement the interface

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
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.
Component diagram with interfaces

Example of a component diagram:

**Older notation:**

```
<<component>>
```

**Alternative notation:**

```
<<component>>
```
Relationships - (6/6) overview and intuition - Realization

**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) "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**
Relationships - (6/6) overview and intuition
- Dependency

```
ConfigDialog

ConfigManager
+ rememberPsw: Boolean
+ useMasterPsw: Boolean
+ changeWarningMsg()
```

Dependency

client

supplier
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".
Artifacts

- Components describe modular parts of the system with well-defined interfaces
- Artifacts are the pieces of information produced by software development

```
<<component>>
PurchaseOrder

<<manifest>>
PurchaseOrder.jar
```

```
<<artifact>>
PurchaseOrder.jar
```
Relationships - (6/6) overview and intuition - Dependency

UML as sketch
- Help to communicate *some important* aspect of system
- Common medium
- In documents, focus on communication compared to completeness

Reverse engineering can be very useful to see dependencies between classes and modules!

UML as blueprint

Forward engineering

Reverse engineering

UML as programming language

UML model

Compile

Executable Code

Programming Code

Round-trip engineering
Part III
State Machines and Sequence Diagrams
For class CoinHandler:

- State: checking
  - trigger event: falseCoin()/returnCoin(self)
  - action, reaction: transition

- State: idle
  - trigger event: insertCoin()/checkCoin(self)
  - action, reaction: transition

Start state marker

This object
Orthogonal, composite state

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
Explicit exit points

course attempt

- Studying
  - Lab 1 \(\text{lab1 done}\) → Lab 2 \(\text{lab2 done}\)
  - Project \(\text{project done}\)
  - Final exam \(\text{pass}\)
- Lab 1 \(\text{fail}\)
- Lab 2 \(\text{fail}\)
- Project \(\text{fail}\)
- Final exam \(\text{fail}\)

failed passed

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
Activity diagram ≠ State diagram

Initial node

- insert coin

<table>
<thead>
<tr>
<th>decision</th>
<th>coin accepted?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[no]</td>
</tr>
</tbody>
</table>

fork

- brew coffee
- add sugar/whitener

join

- add hot water to adjust strength
- pour coffee

final node

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
Sequence diagram

Role: CoffeeCustomer

Message:
- insertCoin
- machineReady
- pressButton(b1)
- pourCoffee

Life line of object:
Procedure is active

Part I
Classes and Objects

Part II
State Machines and Sequence Diagrams
Sequence diagram with several objects

- : CoffeeCustomer
  - insertCoin
  - pressButton(b1)
  - pourCoffee

- : Interface
  - coinAccepted

- : CoinHandler
  - warmUp
  - makeOrder(o1)

- : Brewer
  - transport
  - pourCoffee

{C-A < 5s}
Combining fragments of interaction diagrams

SD processOrder

:Order

:TicketDB

:Account

create

ref

Get existing customer data

[get next item]

reserve(date,no)

add(seats)

answer

destruction

loop

loop condition

loop

Part I

Classes and Objects

Part II

State Machines and Sequence Diagrams

krisitan.sandahl@liu.se
More fragments of interaction diagrams

```
:Order

loop
[get next item]
reserve(date,no)
alt
[available]
add(seats)
reject
[unavailable]

:TicketDB
```

- **guard condition**
- **nested conditional**
- **alternate branches**