Software Architecture

Bertrand Meyer, Carlo A. Furia, Martin Nordio
(Christian Estler)

ETH Zurich, February-May 2011

Lecture 16: UML - Unified Modeling Language
Why do we need models?

Ideal world...

Problem domain

Implementation

Code

Reality

Problem domain

Implementation

Code
Why do we need models?

- Models are abstractions of „the real thing“
- They hide complexity by looking at a problem from a certain perspective
  - Focus on relevant parts
  - Ignoring irrelevant details
  - What is relevant depends on the model
- Example: to model the main components of a car, we do not need internal details of the engine.
Why model software?

- Why is code itself not a good model?
- Software is getting increasingly more complex
  - Windows XP: ~40 millions lines of code
  - A single programmer cannot manage this amount of code in its entirety
- Code is not easily understandable by developers who did not write it
- We need simpler representations for complex systems
- Modeling is a means for dealing with complexity
Unified Modeling Language (UML)

- General purpose modeling language (for [OO software] systems)
- Today’s de-facto standard in Industry

- Sine ’97, UML is defined/evolved by the Object Management Group (OMG)
  - Founded 1989 by IBM, Apple, Sun, ...
  - Microsoft joined 2008
  - Today more than 800 members
Authors: The Three Amigos

Grady Booch  James Rumbaugh  Ivar Jacobson
Why “Unified” Modeling Language?
What is UML?

UML is a standardized language for specifying, visualizing, constructing and documenting (software) systems.

- **Specification:** the language is supposed to be simple enough to be understood by the clients.
- **Visualization:** models can be represented graphically.
- **Construction:** the language is supposed to be precise enough to make code generation possible.
- **Documentation:** the language is supposed to be widespread enough to make your models understandable by other developers.
What is UML?

- UML defines
  - Entities of models and their (possible) relations
  - Different **graphical notations** to visualize **structure** and **behavior**

- A model in UML consist of
  - Diagrams
  - Documentation which complements the diagrams
What UML is *not*!

- Programming language
  - this would bound the language to a specific computing architecture
  - however code generation is encouraged

- Software development process
  - Choose your own process, (e.g. Waterfall-model, V-model, ...)
  - Use UML to model & document

- CASE tool specification
  - however tools do exist: Sun, IBM Rose, Microsoft Visio, Borland Together etc.
Diagrams in UML

- UML currently defines 14 types of diagrams
  - 7 types of Structure Diagrams
  - 7 types of Behavior Diagrams

- Different diagrams provide different levels of abstraction
  - High-level structure vs. low-level structure
    - Example: components vs. objects
  - High-level behavior vs. low-level behavior
    - Example: use-case vs. feature-call sequence
Diagrams in UML

Diagram

Structure Diagram
- Class Diagram
- Component Diagram
- Object Diagram
  - Profile Diagram
  - Composite Structure Diagram
  - Deployment Diagram

Behavior Diagram
- Activity Diagram
- Use Case Diagram
  - Interaction Diagram
    - Interaction Overview Diagram
    - Timing Diagram

Notation: UML
Case study*

- ETH cafeteria wants to introduce a card-based payment system
- Students upload money to their card using a special automaton (similar to an ATM)
- Using their cards, students pay cashless in the cafeteria

* inspired by: http://www.fbi.h-da.de/labore/case/uml.html
Use Case diagram

- **Use Case diagrams**
- High-level abstraction of the system's external behavior
- From the client's perspective
- What the client plans to do with the system

Elements of a Use Case diagram:

- **System name**
- **Actor name**
- **Use Case name**
- **Boundary of the planned system**
- **Person or System that interacts with the planned system**
- **Task that should be executed by the planned system**
Use Case diagram for the payment system

- **Client**
- **Caf_Pay**
- **Log Transaction**
- **Load Card**
- **Accounting**
- **Teller**

- **Pay Food**
  - **<<include>>**
  - **<<include>>**

**Association between Actor and Use Case**
**Dependency between Use Cases**
**Stereotype**
Use Case diagrams: include-association

<<include>> stereotype to include use cases:
reusing common functionality, no duplicates

“Pay Food” and “Load Card” use the functionality provided by “Log Transaction”
Use Case diagrams: extend-association

<<extend>> stereotype to provide special case

Normal case specifies point at which the behavior may diverge (extension point)

Extending case specifies condition under which the special case applies (as entry condition)
Elements of Use Case diagrams

- **Entities:**
  - actors
  - use cases

- **Relations:**
  - association between an actor and a use case
  - generalization between actors
  - generalization between use cases
  - dependencies between use cases

- **Comments:**
  - system boundaries
Use Case specification

- Each Use Case shown in a diagram should be accompanied by a textual specification

- The specification should follow the scheme:
  - Use Case name
  - Actors
  - Entry Condition
  - Normal behavior
  - Exceptions
  - Exit Condition
  - Special Requirements (e.g. non-functional requirements)
Example for "Pay Food" Use Case

**Name:** Pay Food

**Actors:** Client, Teller

**Entry Condition:** Client has food and wants to pay it

**Normal behavior:** Teller types in food; Total amount is shown on display; Client puts card into reading device; Amount gets withdrawn; If not enough money on card, then an error message is shown; Return card to client

**Exceptions:** If card is not readable, then show error message and return card; If power failure while card in reading device, wait until power is back and return card - payment needs to be redone

**Exit Condition:** Client has paid the food and gets the card back
Diagrams in UML

- Structure Diagram
  - Class Diagram
  - Component Diagram
  - Object Diagram
  - Profile Diagram
  - Composite Structure Diagram
  - Deployment Diagram
  - Package Diagram

- Behavior Diagram
  - Activity Diagram
  - Use Case Diagram
  - Interaction Diagram
  - State Machine Diagram
  - Sequence Diagram
  - Communication Diagram
  - Interaction Overview Diagram
  - Timing Diagram

Notation: UML
Activity diagrams

- Activity diagrams are used to model (work)flows

- They are used to visualize complex behavior, e.g.
  - Business process
  - Algorithms (though less common)

- Tokens are used to determine the flow, similar to Petri-nets

- A common usage: detailed modeling of Use Cases
Elements of Activity diagrams

- **Action**: atomic element, no further splitting possible

- **Activity**: can contain activities, actions, control nodes

- **Control nodes**: used to denote control structure in the flowgraph

**Diagrams**:
- Initial node: start of a flow
- Decision- / Merging node
- End node: terminates the activity
- Splitting node
- Synchronization node
An activity diagram for the case study

Activity Pay_Food

Pay Desk
- Type in food
- Reverse entry

Display
- Display food costs
- Display card amount
- Display error

Card Reader
- Insert card
- Read card
- Compare
- Withdraw amount
- Remove card

- [insufficient amount]
- [sufficient amount]
Diagrams in UML

Diagram

Structure Diagram
- Class Diagram
- Composite Structure Diagram
- Deployment Diagram
- Package Diagram

Behavior Diagram
- Activity Diagram
- Use Case Diagram

Interaction Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

Notation: UML
UML Class diagrams

- Keep in mind:
  - Use Cases represent an external view of the system’s behavior
  - Classes represent inner structure of the system
  → No correlation between use cases and classes

- Class diagrams are used at different levels of abstraction with different levels of details
  - Early phase: identifying classes and their relations in the problem domain (high-level, no implementation details)
  - Implementation phase: high level of detail (attributes, visibility, ...), all classes relevant to implement the system
A class encapsulates **state** (attributes) and **behavior** (operations)

- Each attribute has a type
- Each operation has a signature

The class name is the only mandatory information.
More on classes

Valid UML class diagrams

<table>
<thead>
<tr>
<th>Cafeteria_Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount</td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>get_amount()</td>
</tr>
<tr>
<td>get_id()</td>
</tr>
</tbody>
</table>

Corresponding BON diagram

- No distinction between attributes and operations (uniform access principle)
More on classes

- **Abstract classes** have a *italicized* class name or `{abstract}` property (also applicable to operations)

<table>
<thead>
<tr>
<th>Card</th>
<th>Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td>get_id()</td>
<td>get_id()</td>
</tr>
</tbody>
</table>

OR

<table>
<thead>
<tr>
<th>Card {abstract}</th>
<th>Card {abstract}</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td>get_id()</td>
<td>get_id()</td>
</tr>
</tbody>
</table>

- **Parameterized classes**

<table>
<thead>
<tr>
<th>List {T, k: Integer}</th>
<th>&lt;&lt;bind&gt;&gt; &lt;T-&gt;Address, k-&gt; 250&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>items: T[0..k]</td>
<td>Address_Book</td>
</tr>
</tbody>
</table>
**Interface classes**

- Interface classes have a keyword `<<interface>>`
- Interfaces have **no attributes**
- Classes implement an interface using an implementation relation

![Diagram showing interface classes]

```plaintext
<<interface>>
ICard_Reader

read()
write()

ICard_Reader

read()
write()
```
Generalization and specialization

- Generalization expresses a kind-of ("is-a") relationship

- Generalization is implemented by inheritance
  - The child classes inherit the attributes and operations of the parent class

- Generalization simplifies the model by eliminating redundancy
Associations

- A line between two classes denotes an association
- An association is a type of relation between classes
- Objects of the classes can communicate using the association, e.g.
  - Class A has an attribute of type B
  - Class A creates instances of B
  - Class A receives a message with argument of type B
Association multiplicity

- **Multiplicity** denotes how many objects of the class take part in the relation
  - 1-to-1
    
    ![1-to-1 Diagram](Diagram)
  
  - 1-to-many
    
    ![1-to-many Diagram](Diagram)
  
  - many-to-many
    
    ![many-to-many Diagram](Diagram)
Association roles

- Different instances of a class can be differentiated using roles

- Example: Invoice and shipping address are both addresses

- Example: Position hierarchy
Special associations

- **Aggregation** - “part-of” relation between objects
  - Component can be part of multiple aggregates
  - Component can be created and destroyed independently of the aggregate

- **Composition** - strong aggregation
  - A component can only be part of a single aggregate
  - Exists only together with the aggregate
More on associations

- **Ordering** of an end - whether the objects at this end are ordered
- **Changeability** of an end - whether the set of objects at this end can be changed after creation
Navigability of association

- Associations can be **directed**
- Direction denotes whether objects can be accessed through this association

Card knows about Card_Reader

Card_Reader knows about Card

Card and Card_Reader know about each other
Class diagram for the case study

- **ETH_Card**
  - `id`

- **Pay_Automaton**

- **Cafeteria_Card**
  - `amount`
  - `read()`
  - `write()`

- **Pay_Desks**
  - `input_food()`
  - `compare()`

- **Display**
  - `show_Amount()`
  - `show_Error()`

- **Card_Reader**
  - `card_insert()`
  - `card_Amount()`

- **Money_Slot**
  - `money_In()`
  - `money_Out()`
  - `validate_Bill()`
UML Object diagrams

- An Object diagram is used to denote a snapshot of the system at runtime.
- It shows the existing objects, their attribute values and relations at that particular point of time.

**Object identifier**

<table>
<thead>
<tr>
<th>42: Cafeteria_Card</th>
<th>pd1: Pay_Desk</th>
</tr>
</thead>
<tbody>
<tr>
<td>id=235813</td>
<td></td>
</tr>
<tr>
<td>amount=25.50</td>
<td></td>
</tr>
</tbody>
</table>

**Link:** name or role-name are optional.
Diagrams in UML

Structure Diagram
- Class Diagram
- Component Diagram
- Object Diagram
- Deployment Diagram
- Package Diagram

Behavior Diagram
- Activity Diagram
- Use Case Diagram
- Interaction Diagram
- State Machine Diagram

Notation: UML
A package is a UML mechanism for **organizing elements** into groups

- Usually not an application domain concept
- Increase readability of UML models

**Decompose** complex systems into subsystems

- Each subsystem is modeled as a package
Diagrams in UML

Notation: UML
Component diagrams

- **Entities:**
  - components
    - programs
    - documents
    - files
    - libraries
    - DB tables
  - interfaces
  - classes
  - objects

- **Relations:**
  - dependency
  - association (composition)
  - implementation

```
<<component>>
DataBase

<<component>>
Business

<<provided interfaces>>
Interface_m
<<required interfaces>>
Interface_n

<<realization>>
Class_A
Class_B

<<artifact>>
library.jar
```
Diagrams in UML

- Structure Diagram
  - Class Diagram
  - Component Diagram
  - Object Diagram
  - Deployment Diagram
  - Package Diagram
- Behavior Diagram
  - Activity Diagram
  - Use Case Diagram
  - Interaction Diagram
  - State Machine Diagram
  - Collaboration Diagram
  - Communication Diagram
  - Interaction Overview Diagram
  - Timing Diagram

Notation: UML
Overview

- We will now look at two more diagrams which are used to model the **behavior** of a system.

- **Sequence diagrams**: used to describe the interaction of objects and show their “communication protocol”

- **State diagrams**: focus on the state of an object (or system) and how it changes due to events
Sequence diagrams

- **Entities:**
  - objects (including instances of actors)

- **Relations:**
  - message passing

- **Sugar:**
  - lifelines
  - activations
  - creations
  - destructions
  - frames

![Sequence Diagram](image)
Nested messages

The source of an arrow indicates the activation which sent the message.

An activation is as long as all nested activations.
Creation and destruction

Creation is denoted by a message arrow pointing to the object

In garbage collection environments, destruction can be used to denote the end of the useful life of an object
From Use Cases to Sequence diagrams

Sequence diagrams are derived from flows of events of use cases.

An event always has a sender and a receiver.
- Find the objects for each event.

Relation to object identification:
- Objects/classes have already been identified during object modeling.
- Additional objects are identified as a result of dynamic modeling.
**Example Sequence diagram**

- **Client**
- **Card Reader**
- **Caf Card**
- **Display**
- **Money Slot**

1. Insert card
2. Read ()
3. Show_Amount()
4. Insert money
5. Validate()
6. Draw card

Example Sequence:

**Insert card**

- **Read ()**
- **Show_Amount()**

**Insert money**

- Validate()
- Show_Amount()
Example Sequence diagram

- The diagram shows only the successful case

- Exceptional case could go either on another diagram or could be incorporated to this one

- Sequence diagrams show main scenario and “interesting” cases
  - interesting: exceptional or important variant behavior

- Need not draw diagram for every possible case

- would lead to too many diagrams
Interaction frames

:Container

:Processor

:Item

Loop
[for each item]

Process()

Alt
[value < 100]

[else]

Increase()

decrease()
Fork structure

The dynamic behavior is placed in a single object, usually a control object.

It knows all the other objects and often uses them for direct queries and commands.
Stair structure

The **dynamic behavior is distributed**

- Each object delegates some responsibility to other objects
- Each object knows only a few of the other objects and knows which objects can help with a specific behavior
Fork or stair?

Object-oriented supporters claim that the stair structure is better

- The more the responsibility is spread out, the better

Choose the **stair** (decentralized control) if

- The operations have a **strong connection**
- The operations will **always** be performed in the **same order**

Choose the **fork** (centralized control) if

- The operations can **change order**
- **New operations** are expected to be added as a result of new requirements
Diagrams in UML

- Structure Diagram
  - Class Diagram
  - Composite Structure Diagram
  - Deployment Diagram
  - Package Diagram

- Behavior Diagram
  - Activity Diagram
  - Use Case Diagram
  - Interaction Diagram
  - State Machine Diagram

Notation: UML
State Machine Diagrams

- UML State Machine Diagrams are a powerful notation to model finite automata.

- It shows the states which an object or a (sub)system - depending on the level of abstraction - can have at runtime.

- It also shows the events which trigger a change of state.
State Machine diagrams

- Entities:
  - states: name, activity, entry/exit action
- Relations:
  - transitions between states: event, condition, action
State Machine diagram for the case study

Remember: event (arg) [ condition ] / action

Request money

| do / increase amount
| entry / validate money

Card check

| Insert_Card               |
| [ Card = OK ]            |
| [Card = !OK ]            |
| / return_card            |

Cancel/ return_card

Insert_Money

Load card

Done
Activities in states can be composite items that denote other state diagrams

Sets of substates in a nested state diagram can be denoted with a superstate

- Avoid spaghetti models
- Reduce the number of lines in a state diagram
State diagrams: example composite state

TrafficLight

- Off
- On
- Blinking
- Working
- Green
- Red
- Yellow

TurnOn:
- after 3 sec

SwitchOn:
- after 30 sec

SwitchOff:
- after 45 sec

TurnOff:
- after 5 sec
Example: superstate

- **Idle**
  - entry / clear balance

- **CollectMoney**
  - insCoin(amount) / add to balance

- **TicketSelected**
  - selectTicket(tkt)

- **ExactlyPaid**
  - [ change = 0 ]
  - [ ticket dispensed ]

- **OverPaid**
  - [ change > 0 ]
  - do / dispense change

- **Superstate**
Expanding the superstate

Transitions from other states to the superstate enter the first substate of the superstate

Transitions to other states from a superstate are inherited by all the substates (state inheritance)
State diagram vs. Sequence diagram

**State diagrams** help to identify

- Changes to an individual object over time

**Sequence diagrams** help to identify

- The temporal relationship between objects
- Sequence of operations as a response to one or more events
Diagrams in UML
Practical tips

- Create **component** diagrams only for large, distributed systems
- Create **state** diagrams only for classes with complex, interesting behavior (usually classes representing entities from the problem domain or performing control)
- Create **activity** diagrams for complex algorithms and business processes (not for every operation)
- Create **sequence** diagrams for nontrivial collaborations and protocols (not for every scenario)
- Don’t put too much information on a diagram
- Choose the level of abstraction and maintain it