Course "Softwaretechnik"
Book Chapter 2

Modeling with UML

Lutz Prechelt, Bernd Bruegge, Allen H. Dutoit
Freie Universität Berlin, Institut für Informatik

- Modeling, models and UML
- Static view:
  - Class diagrams
- Dynamic view:
  - Sequence diagrams
  - State machine diagrams
  - Activity diagrams
- Other UML diagram types
  - component d., collaboration
  - use d., deployment d.,
  - communication d., interaction
  - overview d.
- UML Metamodel, Profiles
- Some notation details
  - Classes, associations,
  - interfaces, states
What is modeling?

- Modeling consists of building an abstraction of reality
  - Models ignore irrelevant details (i.e., they simplify)
  - and only represent the relevant details
- What is *relevant* or *irrelevant* depends on the purpose of the model. We typically want to
  - draw complicated conclusions about reality
    - with simple steps in the model in order to
  - get insights into the past or presence or make predictions

- Reality R:
  - Real things, people, etc.
  - Processes happening during some time
  - Relationships between things etc.
- Model M:
  - Abstractions of any or all of the above
What is a "good" model?

- In a good model, relationships which are valid in reality $R$ are also valid in model $M$ (if they exist in $M$ at all).
  - $I$: Mapping of reality $R$ to the model $M$ (abstraction)
  - $f_M$: relationship between abstractions in $M$
  - $f_R$: equivalent relationship between real things in $R$

- In a good model, the following diagram is commutative:
Models of models of models…

- We can think of a model as reality and can build another model from it (with additional abstractions)
  - The development of software systems is a transformation of models:
    Requirements elicitation  $\rightarrow$ req's document $M_1$,
    Requirements analysis  $\rightarrow$ analysis model $M_2$,
    Design  $\rightarrow$ design model $M_3$,
    Implementation  $\rightarrow$ source code $M_4$

"Model-Driven Development" uses this idea for its engineering approach.
Systems, models and views

- A **model** is an abstraction describing relevant aspects of a system
- A **view** ("Sicht") depicts selected aspects of a model
  - Any view is a model itself
  - Calling a model a view makes clear it is part of a larger model
  - Complex models are often shown as many views only
    - never as a whole
- A **notation** is a set of rules for depicting models
  - graphically or textually

Example:
- System: Aircraft
- Models: Flight simulator, scale model, construction plan, ...
- Views: All blueprints (e.g. electrical wiring, fuel system)
What is UML?

UML (Unified Modeling Language):
- The most-used standard for software modeling
  - For both requirements modeling (application domain)
  - and software modeling (solution domain)
- A set of related notations
  - Quite complex, we will use a subset only
- Resulted from the convergence of notations from three leading object-oriented methods:
  - OMT (James Rumbaugh)
  - OOSE (Ivar Jacobson)
  - Booch method (Grady Booch)
  - The authors are known as "The Three Amigos"

- Supported by CASE tools
Common UML diagram types

- **Use Case diagrams** (functional view)
  - Catalog scenarios that describe the functional behavior of the system as seen by the user [see lecture "use cases"]

- **Class diagrams / Object diagr.** (static view and examples)
  - Describe the static structure of the system: Classes, attributes, object associations (class diagram) or snapshots of possible resulting configurations (object diagram)

- **Sequence diagrams** (dynamic view examples)
  - Describe examples of the dynamic behavior between objects of the system (and possibly actors)

- **State machine diagrams** (dynamic view)
  - Describe some aspects of the dynamic behavior of the individual object of a class by a finite state automaton

- **Activity diagrams** (dynamic view)
  - Model the dynamic behavior of a system, in particular the workflow (essentially a flowchart, but with concurrency)
Less common UML diagram types

Hardly covered in this course:

- Implementation diagrams
  - Component diagrams
  - Deployment diagrams
- Communication diagrams
  - Equivalent to sequence diagrams, but embedded in an object diagram (shows both static structure and dynamic interaction)
- Interaction overview diagrams
  - Related to activity diagrams, for describing control flow

There is also a non-graphical language for expressing conditions:

- Object constraint language (OCL)
  - Introduced in lecture on Object Design
UML core conventions

• Diagrams are mostly graphs
  • Nodes are entities
  • Edges are relationships between entities

• Rectangles are classes or instances
• Ovals are functionalities or use cases

• An instance is denoted with an underlined name
  • `myWatch:SimpleWatch` or with no classifier: `myWatch`
  • `Jane:Firefighter` or with no name: `:Firefighter`
  • (Anonymous instance of unnamed classifier: `:`
    • Please don’t use this ...)

• A classifier is denoted with a non-underlined name
  • SimpleWatch
  • Firefighter
Class diagrams represent the structure of the system.
A class has a name and represents a concept
- The class name is the only mandatory information in a UML class description

A class encapsulates state (attributes) and behavior (operations)
- Each attribute has a type
- Each operation has a signature
Instances ("Exemplare", "Objekte")

- An instance represents a phenomenon
- The name of an instance is underlined and may indicate the class of the instance
  - May indicate instance name or class or both
- Attributes may be represented with their values

```
tariff1974 : TariffSchedule
zone2price = {
    {'1', 0.20},
    {'2', 0.40},
    {'3', 0.60}
}
```
Associations denote relationships between classes.

The multiplicity of an association end denotes how many objects the source object can legitimately reference:

- Any one TariffSchedule object is associated with at least one TripLeg object.
- Any one TripLeg object is associated with at least one TariffSchedule object.
- \( n \) and \( m \) can be numbers ("5") or ranges (closed/open: "1..5" or "2..*")
- A missing annotation can mean "don't know yet", "is specified elsewhere" or (by special convention) "1"
- "*" alone means "arbitrarily many" (zero, one, or several)
1-to-1 and 1-to-many associations

### One-to-one association

Country
- name : string

Capital

City
- name : string

(Too restrictive?: Some countries have a separate seat of government)

### One-to-many association

Polygon
- draw()

Point
- x : int
- y : int

(Too flexible?: Does a Polygon with 0, 1, or 2 Points really make sense?)
Many-to-many associations

Problem Statement: "A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol."

(Now a Company could have different tickerSymbols at each StockExchange)
Aggregation

- An **aggregation** is a special case of association denoting a "consists of"/"is part of" hierarchy.
- The object representing the whole is called the **aggregate**, the objects representing the parts are called **components**.

A solid diamond denotes **composition**, a strong form of aggregation where the parts never exist without the composite.
- The association is in force throughout the life of the parts objects.

Lutz Prechelt, prechelt@inf.fu-berlin.de
Inheritance (Java: "extends")

- The **children classes** inherit the attributes and operations of the **parent class**

- Read the triangle as an arrowhead, meaning "inherits from" (just like "extends")
  - `CancelButton` inherits from `Button`
  - `ZoneButton` inherits from `Button`
Realization (Java: "implements")
Example: Plato’s and Aristotle’s world views

**Plato**

- Reality
  - Thing
    - Particular
    - Form

**Aristotle**

- Reality
  - Substance
    - Form
    - Matter
Association classes

- Individual associations between objects can have attributes
  - Described by an association class
Association constraints

- Associations can be described by further details:

![Diagram showing association between Person, Address, and TransactionEntry with details like address, visibility, history, ordering property, and changeability constraint.]
Class diagrams: theater example

- ..and some more notation details:
  - role name
  - XOR constraint
  - static operation
Packages

- A package is a UML mechanism for organizing elements (e.g. classes or whole class diagrams) into groups
  - to simplify model readability and handling

- A complex system can be decomposed into subsystems, where each subsystem is modeled as a package
UML sequence diagrams

- Used during requirements analysis and system design
  - to refine the model
- Used for explaining behavior to other stakeholders

Objects are represented by columns (objname:classname)
Messages are represented by arrows
Activations are represented by narrow rectangles
Lifelines are represented by dashed lines
Nested messages

- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations (for normal calls)
- Horizontal dashed arrows indicate data flow
- Vertical dashed lines indicate lifelines

...to be continued...
Sequence diagram: theater example

external call, external return

gate (external call)

create()

:Order

message
reserve (date, count)
synchronous call

:TicketDB

role

lifeline
debit (cost)

:Account

bonus (date, count)

second call to object

ongoing objects

destruction

gate (external return)

execution specification

creation

return
Advanced features

- creation
- nesting

- iteration

- conditions, branching

- destruction
Sequence diagram summary

- UML sequence diagrams represent behavior in terms of object interactions
  - Useful to find missing objects
  - Useful for explaining design ideas
    - Describes examples only, no general specification
- Time-consuming to build, but may be worth it
- Complement the class diagrams (which represent structure)
State machine diagrams

**State**

- Zeit anzeigen
  - do / zeitAnzeigen()

- Minuten einstellen
  - do / blinkMinuten()

**Initial state**

**Event**

- Knopf A

**Transition**

- Knopf A

**Transition Effect**

- Knopf B / stundenErhöhen()
Transitions can be subject to guard conditions.

- **Waiting**
  - Transition: receive order [amount > $25]
  - Transition: receive order [amount < $25] (guard condition)

- **Confirm Credit**
  - Transition: approved/debit account()
  - Transition: rejected
  - Transition: trigger event

- **Process Order**
  - Action: trigger event
Parallel (orthogonal) states, explicit exits
A transition is the consequence of an event

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>call event</td>
<td>Receipt of an explicit synchronous call request by an object</td>
<td>\texttt{op (a:T)}</td>
</tr>
<tr>
<td>change event</td>
<td>A change in value of a Boolean expression</td>
<td>\texttt{when (exp)}</td>
</tr>
<tr>
<td>signal event</td>
<td>Receipt of an explicit, named, asynchronous communication among objects</td>
<td>\texttt{sname (a:T)}</td>
</tr>
<tr>
<td>time event</td>
<td>The arrival of an absolute time or the passage of a relative amount of time</td>
<td>\texttt{after (time)}</td>
</tr>
</tbody>
</table>
There can be multiple transitions at one state

- Internal transitions don’t leave the state
- Entry and Exit Activities can be annotated inside the state box
  - to avoid redundancy and encapsulate the state

also: do / some_activity
  - for a concurrent, abortable, potentially long-running activity occurring throughout the state
Activity Diagrams

- An activity diagram shows flow control (and optionally data flow) within a system

- Two types of (executable) nodes:
  - Action node:
    - Basic activity, cannot be decomposed any further
    - Predefined in UML, e.g. object creation/destruction, accessing/modifying attributes or links, calling operations
  - Activity node:
    - Can be decomposed further
    - The activity is modeled by another activity diagram
State machine diagram vs. activity diagram

State machine diagram for Incident
(Node represents some set of attribute values)

Activity diagram for Incident handling
(Node represents some collection of operations)
Activity diagram: decisions

OpenIncident -> Decision

- [lowPriority] -> AllocateResources
- [fire & highPriority] -> NotifyFireChief
- [not fire & highPriority] -> NotifyPoliceChief

AllocateResources
Activity diagrams: concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads
Activity diagrams: theater example

(ohne neue Konstrukte)
Further UML diagram types

Static view:

- Component diagrams, internal structure diagrams
  - Subsystems (components) and their interfaces
- Collaboration use diagram
  - A part of a structure that collaborates for a specific purpose
- Deployment diagrams
  - Computers and which part of the system runs on which

Dynamic view:

- Communication diagrams
  - Equivalent to sequence diagrams, but embedded in an object diagram (shows both static structure and dynamic interaction)
- Interaction overview diagrams
  - Related to activity diagrams, for describing control flow
Components

- Components represent classes or subsystems (multiple classes)
  - The focus is on their interfaces
  - All implementation details are ignored
Component diagram, internal structure diagram

- Compositions of components
  - Component diagram: relationships between components
  - Internal structure diagram: structure of a component (as below) or any other classifier

```
Booking

Scheduler

MakeReservations

UpdatePlans

TravelSystem

Planner

TripGUI

User Console
```
Collaboration use diagram

- A view describing the roles different parts play for one specific purpose
  - Can be on class level or on instance level
Deployment diagram

for distributed systems: describes which code runs on which computer ("node")
Communication diagram

- An object diagram with interaction annotations
  - Indicates interactions (like a sequence diagram) as well as object relationships (by the object diagram)

```
comm takeOrder

request(order, customer) → :OrderTaker

1: checkCredit(customer) ↓

: CreditBureau

credit

2: cost = reserve(order) → :TicketDB

3: debit(customer, cost)

message flow

sequence number

connector: tickets

role

one-way navigation on underlying association
```
Interaction overview diagram

- A combination of activity diagram and sequence diagram:
  - activities may be sequence diagram fragments
Diagram types overview (UML 2.2)

Notation: UML

Diagram

Structure Diagram

Class Diagram
Profile Diagram
Composite Structure Diagram

Component Diagram
Deployment Diagram

Object Diagram
Package Diagram

Behaviour Diagram

Activity Diagram
Use Case Diagram

Interaction Diagram

Interaction Overview Diagram

Sequence Diagram
Communication Diagram

Timing Diagram

source: Wikimedia Commons
UML is described in UML itself

- The UML model describing UML is called the **UML metamodel**
  - It consists of UML class diagrams plus descriptive text

- Class level: Every kind of UML element (e.g. "association") is a class in that metamodel
  - The characteristics are described by attributes or associated classes
  - e.g. the UML metamodel contains a class `Association`

- Instance level: Every association in a specific UML model can be interpreted as an instance of the `Association` class in the UML metamodel
  - But actually there is much more than just one class:
The UML Metamodel of associations

Source:
UML 2.4.1, section 7.2
http://www.omg.org
UML is extensible

- **Profiles** add elements to the UML metamodel
  - A profile is a package that defines «stereotypes» and constraints that can be applied to certain metamodel elements.
UML is fairly precise

• In this course, we will be using UML in a rather informal and imprecise manner
  • Our models are usually not very detailed
  • They leave many things unspecified (i.e., they are incomplete)

• However, one can produce fairly precise UML models
  • Such models have a reasonably well-defined meaning, as UML itself is specified in a semi-formal manner
    • No complete semantics have been specified for UML overall, though
  • There is much more to UML than can be said here
    • UML 2.4 Infrastructure + Superstructure: 200 + 800 pages
    • UML 2.5, rewritten in one document: 800 pages

• Precise UML usage is relevant for automatic code generation from the UML model
  • In some domains, such as telecommunication, complete subsystems are sometimes code-generated from UML models today
What should you know about UML?

• For all application domains:
  • Learn as much as you can about class diagrams (object diagrams help in doing this)
    • (soon maybe also component diagrams)
  • Learn the basics of use case, sequence, communication, state machine, and activity diagrams

• For realtime and formally specifiable (sub)domains:
  • Also learn a lot about state machine diagrams

• If you want to make full use of UML CASE tools:
  • Learn a lot about packages and about profiles

• If you want to build UML CASE tools:
  • Learn about the UML metamodel (Warning: tough!)
UML summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful, but complex language
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features
  - Many people who say they "know UML" actually know very little

- We will concentrate on a few notations:
  - Functional model: use case diagram
  - Object model: class diagram
  - Dynamic model: sequence diagrams, state machine and activity diagrams
Literature

  - this is also the source of the figures with blue annotations

  - actually teaches how to use the UML
    - this lecture did not do this, but some of the rest of the course will
  - less misleading than some other books on the topic

The current version of UML is 2.5.1 (December 2017).
Thank you!

and now: some bonus slides
UML language elements details

• The next few slides present a number of details in the notation of
  • Classes (Class diagrams)
  • Associations (Class diagrams)
  • Interfaces (Class diagrams)
  • States (State machine diagrams)
Details: Class

```mermaid
classDiagram
  Cname
  + attrName: Cname = expression
  # attrName: Cname
  - attrName: Cname [*]
  + opName (p:C1,q:C2): C3
  «constructor»
  opName (v:Cname=value)
  Responsibilities
  text description

  «stereotypeName»
  tagName = value
```

- **class**
- **visibility**
- **optional named compartment**
- **stereotype icon**
- **stereotype name**
- **class name (italics for abstract)**
- **public attribute with initial value**
- **protected attribute**
- **private attribute with multiplicity many**
- **public concrete operation with return type**
- **stereotype on subsequent operations**
- **abstract operation with default value**
- **compartment name**
- **compartment list element**
- **stereotype application**
- **tagged value**
Details: Association
Details: Interfaces

- realization
- usage
- provided interface
- required interface
- dependency
- supplier
- client

explicit style
implicit style
Details: States

- **StateA**: e1 (p:C) [cond] / action1; action2
- **StateB**: entry / action3, exit / action4, e1 / action5, do / activity1
- **StateC**: orthogonal state
- **StateD**: e3

**Event Components**:
- Event name
- Event parameters
- Guard condition
- Effects

**State Transition**:
- Transition between states

**Activity Components**:
- Entry activity
- Exit activity
- Internal transition
- Do activity

**Region**
- Initial state
- Substate
- Final state

**Completion Transition**
- Fires on completion of activity

**Explicit Transition**
- Aborts nested activity