Software Design, Modelling and Analysis in UML

Lecture 1: Introduction

2014-10-21

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
This Lecture:

- **Educational Objectives:** After this lecture you should
  - be able to explain the term *model*.
  - know the idea (and hopes and promises) of *model-based* SW development.
  - be able to explain how *UML* fits into this general picture.
  - know what we’ll do in the course, and why.
  - thus be able to decide whether you want to stay with us...

- **Content:**
  - Analogy: Model-based/-driven development by construction engineers.
  - Software engineers: “me too” – Model-based/-driven Software Engineering.
  - UML Mode of the Lecture: Blueprint.
  - Contents of the course
  - Formalia
Modelling
The following slides may raise thoughts such as:

- “everybody knows this”,
- “completely obvious”,
- “trivial”,
- “clear”,
- “irrelevant”,
- “oversimplified”
- ...

Which is true, in some sense,

- but: “everybody” is a strong claim, and I want to be sure that this holds for the audience from now on.

In other words: that we’re talking about the same things.
An Analogy: The House-Building Problem (Oversimplified)

**Given** a set of **Requirements**, such as:
- The house shall fit on the given piece of land.
- Each room shall have a door, the doors shall open.
- The given furniture shall fit into the living room.
- The bathroom shall have a window.
- The cost shall be in budget.

**Wanted**: a house which satisfies the requirements.

Now, strictly speaking, a house is a **complex system**:
- Consists of a huge number of bricks.
- Consists of subsystems, such as windows.
- Water pipes and wirings have to be in place.
- Doors have to open consistently.
- Floors depend on each other (load-bearing walls).
- ... 

How do construction engineers **handle** this complexity...?
1. Requirements

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

2. Design

3. System

Observation: Floorplan abstracts from certain system properties, e.g.

- kind, number, and placement of bricks,
- subsystem details (e.g., window style),
- water pipes/wiring, and
- wall decoration

→ architects can efficiently work on appropriate level of abstraction
Approach: Floorplan

1. Requirements

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

2. Design

3. System

Observation: Floorplan preserves/determines certain system properties, e.g.,

- house and room extensions (to scale),
- placement of subsystems (such as windows),
- presence/absence of windows and doors,

→ find design errors before building the system (e.g. bathroom windows)
Important Ingredient: Floorplans Have Precise Meaning

- Let floorplan $F$ be an element of a floorplanning language $\mathcal{F}$.
- Let $\phi$ be a property in some requirement specification language $\Phi$.

**Wanted**: notions of

(i) $H \models F$ — house $H$ is **built according to plan** $F$,
(ii) $F \models \phi$ — **plan** $F$ **has property** $\phi$,
(iii) plan $F$ **preserves/determines property** $\phi$, i.e.

\[
\forall H \models F \bullet F \models \phi \iff H \models \phi.
\]

(i)–(iii) [more or less] given for floorplans and houses; $F \models \phi$ avoids $H \not\models \phi$.

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.
Floorplan $F$ denotes a set $\gamma(F)$ of houses (concretisations of $F$), which differ, e.g.
in colour of bricks, or making of windows.

Floorplan $F$ represents house $H$ according to abstraction $\alpha$.

- Note: by adding information to $F$ (such as making of windows),
  we can narrow down $\gamma(F)$.
Given: a house.
Wanted: a concise description for potential buyers.
Approach: draw a floorplan.

Distinguish:

• Sometimes the plan $F$ is first, and the realisation $H \in \gamma(F)$ comes later.
• Sometimes the realisation $H$ is first, and the “plan” $F = \alpha(H)$ comes later.
Note: Requirements Should be Consistent.

- If the requirements are already **contradictory** (or **inconsistent**), then there is **no sense** in drawing a floorplan.

**Example:**

- The house shall fit on the given piece of land.
- The given furniture shall fit into the living room.

What if the land is 10m narrow, the couch is 11m × 11m, and the rooms (as usual) 2.5m high...?
**Definition. [Folk]** A **model** is an abstract, formal, mathematical representation or description of structure or behaviour of a (software) system.

**Definition. [Glinz, 2008, 425]**
A **model** is a concrete or mental **image** (Abbildung) of something or a concrete or mental **archetype** (Vorbild) for something.

Three properties are constituent:

(i) the **image attribute** (Abbildungsmerkmal), i.e. there is an entity (called original) whose image or archetype the model is,

(ii) the **reduction attribute** (Verkürzungsmerkmal), i.e. only those attributes of the original that are relevant in the modelling context are represented,

(iii) the **pragmatic attribute**, i.e. the model is built in a specific context for a specific **purpose**.
Model-Based/-Driven Software Engineering
Software System (Very Abstract View)

We see software $M$ as a transition system.

- It has a (possibly infinite) set of states $S$, (structure)
- an initial state $s_0$, and
- a (possibly $L$-labelled) transition relation

$$\rightarrow \subseteq S \times L \times S.$$ (behaviour)

Software may have infinite and finite runs, i.e. initial and consecutive sequences of states $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots$.

The software engineering problem:

- **Given:** informal requirements $\varphi$.
- **Desired:** correct software, i.e. software $M$ such that $M$ satisfies $\varphi$.

Two prominent obstacles:

- Getting $\varphi$ formal in order to reason about $\varphi$ and $M$, e.g. prove $M$ correct.
- $M$ typically too large to “write it down” at once.
**Model-Driven Software Engineering**

```
Idea

Structure → Declarative Behaviour

Structure' → Constructive Behaviour

Structure'' → Constructive Behaviour'

Declarative Behaviour

Requirements model

Requirements/ constraints

Design

System model

Implementation

Elicit

Refine

Refine

Refine

Generate/ program
```
What would be a “from scratch” approach?

- Define a **formal** language to define requirements and designs.
- Equip it with a **formal** semantics.
- Define consistency/satisfaction relation in terms of semantics.

The approach in this course:

(i) Introduce a common semantical domain — what is a very abstract mathematical characterisation of **object based transitions systems**?
   *Why? Because in the end SW-Engineering is about the creation of (object based) transitions systems and Modeling is about describing them.*

(ii) Take (a fragment of) the visual formal language **UML** as **syntax**.

(iii) Introduce an abstract mathematical **representation of diagrams**.
   *Why? Because easier to handle than “pictures”; abstracts from details such as graphical layout (which don’t contribute to the semantics — note: in floor plans it does).*

(iv) Study the **UML** standard documents for the **informal semantics**.

(v) Define a mapping from (abstract representations of) diagrams to the semantical domain: **assign meaning to diagrams**.

(vi) Define (in terms of the meaning) when a diagram is, e.g., **consistent**.
Model-Driven Software Engineering

Idea

Structure

Declarative Behaviour

requirements model

requirements/constraints
design

Constructive Behaviour

System model

Structure'

Declarative Behaviour'

Structure''

Constructive Behaviour'

Implementation

elicit
refine
refine
refine
refine

generate/program
Idea

Class Diagram

Sequence Diagram

Sequence Diagram

State Machine

State Machine'

Class Diagram

Class Diagram '

Class Diagram''

Implementation

requirements model

requirements/constraints

design

system model

elicit
refine
refine
refine
refine

| =
| =
| |

generate/program

model

constraints
Idea

Class Diagram

Sequence Diagram

requirements model

requirements/constraints

elicit

refine

Class Diagram'

refine

Class Diagram''

[OMG, 2007a, 135]
Idea

Class Diagram

Class A
name: String
shape: Rectangle
+ size: Integer [0..1]
/ area: Integer (readOnly)
height: Integer = 7
width: Integer

Class B
id (redefines name)
shape: Square
height = 7
/ width

[OMG, 2007a, 135]

Class Diagram'

Class Diagram''

Sequence Diagram

Sequence Diagram'

State Machine

State Machine'

requirements model

requirements/constraints

design

system model

generate/program

Implementation
Idea

Class Diagram

Sequence Diagram

Sequence Diagram'

State Machine

Class Diagram'

State Machine'

requirements model

requirements/
constraints

design

system model

[OMG, 2007a, 135]

ClassA
name: String
shape: Rectangle
+ size: Integer [0..1]
/ area: Integer (readOnly)
height: Integer = 5
width: Integer

ClassB
id (redefines name)
shape: Square
height = 7
/ width

[OMG, 2007b, 44]
Idea

Class Diagram

Class Diagram'

Class Diagram''

Sequence Diagram

Sequence Diagram'

State Machine

State Machine'

Implementation

elicit

requirements model

requirements/
constraints

design

system model

[OMG, 2007a, 135]

[OMG, 2007b, 44]
Idea

Class Diagram

Sequence Diagram

Requirements Model

Requirements/Constraints

Design

System Model

Implementation

\[ \text{OMG, 2007b, 44} \]

\[ \text{OMG, 2007a, 135} \]
Idea

Class Diagram

Class A
- name: String
- shape: Rectangle
- area: Integer (read-only)
- height: Integer = 5
- width: Integer

[OMG, 2007a, 135]

Class B
- id (redefines name)
- shape: Square
- height = 7
- width

[OMG, 2007b, 44]

Sequence Diagram

state machine

State Machine

Player PlayedInYear

Year

- season

Team

- team

- goal

OMG, 2007b, 51

success

OK

Unlock

duration

card out { 0..13 }

duration constraint

time observation

duration observation

time constraint

time observation

OMG, 2007b, 513

requirements

refine

requirements model

refine

requirements/

constraints

refine

design

refine

system model

generate/

program

Implementation

elicit

refine

refine
Idea

Class Diagram

Sequence Diagram

State Machine

Implementation

requirements model

requirements/constraints

design

system model

OMG, 2007a, 135

OMG, 2007b, 44

OMG, 2007b, 513

OMG, 2007b, 567

Cl as sB

{re definition}

shape: Square

height = 7

/ width

Cl as sA

name: String

+ size: Integer

/ area: Integer

height: Integer

width: Integer

Player PlayedInYear

Year

Team

T eam

Ye Y

T eam

Ye

[ OMG, 2007a, 135 ]

[ OMG, 2007b, 44 ]

[ OMG, 2007b, 513 ]

[ OMG, 2007b, 567 ]
Idea

Class Diagram

ClassA
name: String
shape: Rectangle
+ size: Integer [0..1]
| area: Integer (readOnly)
height: Integer = 5
width: Integer

[OMG, 2007a, 135]

ClassB
id (redefines name)
shape: Square
height = 7
width

[OMG, 2007b, 44]

Class Diagram'

Year
season
PlayedInYear

Team
* team
* goalie

Player

[OMG, 2007b, 51]

Sequence Diagram

Ye Y
Player PlayedInY ear Y
Ye T eam
Y ear
P layer
P layedI nY ear

[OMG, 2007b, 44]

refine

Class Diagram''

Implementation

---

[OMG, 2007a, 135]

[OMG, 2007b, 44]

[OMG, 2007b, 51]

[OMG, 2007b, 56]
\( \mathcal{I} = (T, C, V, \text{atr}), SM \)

\( M = (\Sigma_\mathcal{I}, A_\mathcal{I}, \rightarrow_{SM}) \)

\( \varphi \in \text{OCL} \)

\( \mathcal{I} \), \( SD \)

\( B = (Q_{SD}, q_0, A_\mathcal{I}, \rightarrow_{SD}, F_{SD}) \)

\( \pi = (\sigma_0, \varepsilon_0) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} (\sigma_1, \varepsilon_1) \cdots \)

\( w_\pi = ((\sigma_i, \text{cons}_i, \text{Snd}_i))_{i \in \mathbb{N}} \)

\( G = (N, E, f) \)

\( \mathcal{O}_D \)
UML Mode
Consequences of the Pragmatic Attribute

Recall [Glinz, 2008, 425]:

[...](iii) the pragmatic attribute, i.e. the model is built in a specific context for a specific purpose.

Examples for context/purpose:

Floorplan as sketch: Floorplan as blueprint: Floorplan as program:
Actually, the last slide is inspired by Martin Fowler, who puts it like this:

“[…] people differ about what should be in the UML because there are differing fundamental views about what the UML should be.

I came up with three primary classifications for thinking about the UML: UmlAsSketch, UmlAsBlueprint, and UmlAsProgrammingLanguage. ([…] S. Mellor independently came up with the same classifications.)

So when someone else’s view of the UML seems rather different to yours, it may be because they use a different UmlMode to you.”

Claim:
- And this not only applies to UML as a language (what should be in it?)
- but at least as well to individual UML models.
**Actually, the last slide is inspired by Martin Fowler, who puts it like this:**

"[...] people differ about what should be in the UML because there are differing fundamental views about what the UML should be. I came up with three primary classifications for thinking about the UML: **UmlAsSketch**, **UmlAsBlueprint**, and **UmlAsProgrammingLanguage**. (S. Mellor independently came up with the same classifications.) So when someone else’s view of the UML seems rather different to yours, it may be because they use a different **UmlMode** to you."

**Claim:**

- And this not only applies to UML as a language (what should be in it?)
- but at least as well to individual UML models.

<table>
<thead>
<tr>
<th>Sketch</th>
<th>Blueprint</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>In this <strong>UmlMode</strong> developers use the UML to help communicate some aspects of a system. [...] Sketches are also useful in documents, in which case the focus is communication rather than completeness. [...] The tools used for sketching are lightweight drawing tools and often people aren’t too particular about keeping to every strict rule of the UML. Most UML diagrams shown in books, such as mine, are sketches. Their emphasis is on selective communication rather than complete specification. Hence my sound-bite “comprehensiveness is the enemy...”</td>
<td>[...] In forward engineering the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up. That design should be sufficiently complete that all design decisions are laid out and the programming should follow as a pretty straightforward activity that requires little thought. [...] Blueprints require much more sophisticated tools than sketches in order to handle the details required for the task. [...] Forward engineering tools support diagram drawing and back it up with a repository to hold the information. [...]</td>
<td>If you can detail the UML enough, and provide semantics for everything you need in software, you can make the UML be your programming language. Tools can take the UML diagrams you draw and compile them into executable code. The promise of this is that UML is a higher level language and thus more productive than current programming languages. The question, of course, is whether this promise is true. I don’t believe that graphical programming will succeed just because it’s graphical. [...]</td>
</tr>
</tbody>
</table>
The “mode” fitting the lecture best is **AsBlueprint**.

The purpose of the lecture’s formal semantics is:
- to be precise to **avoid misunderstandings**.
- to allow formal **analysis of consistency/implication** on the **design level** — find errors early.

while being consistent with the (informal semantics) from the standard [OMG, 2007a, OMG, 2007b] as far as possible.

Plus:
- Being precise also helps for mode **AsSketch**: it should be easier to “fill in” missing parts or resolve inconsistencies.
- Lecture serves as a starting point to define **your** semantics for **your** context/purpose (maybe obtaining a **Domain Specific Language**).
- Lecture could be worked out into mode **AsProgrammingLanguage**.
Course Overview
Table of Contents

- Motivation and Overview (VL 01)
- Semantical Domain (VL 02)
- OCL (VL 03)
- Object Diagrams (VL 04)
- Modelling Structure: Class Diagrams (VL 05–09)
- Modelling Behaviour
  - Constructive: State Machines (VL 10–15)
  - Reflective: Live Sequence Charts (VL 16–18)
- Inheritance (VL 19–20)
- Meta-Modeling (VL 21)
- Putting it all together: MDA, MDSE (VL 22)
Course Path: Over Map

- Motivation
- Semantical Domain
- OCL
- Object Diagrams
- Class Diagrams
- State Machines
- Live Sequence Charts
- Inheritance
- Meta-Modeling
- MDA, MDSE
- Components
- Real-Time
Course Path: Over Time

Diagram with handwritten notes and symbols.
Everything else, including

- **Development Process**
  UML is only the language for artefacts. But: we’ll discuss exemplarily, where in an abstract development process which means could be used.

- **How to come up with a good design**
  UML is only the language to write down designs. But: we’ll have a couple of examples.

- **Requirements Management**
  Versioning, Traceability, Propagation of Changes.

- **Every little bit and piece of UML**
  Boring. Instead we learn how to read the standard.

- **Object Oriented Programming**
  Interesting: inheritance is one of the last lectures.
Formalia
Formalia: Event

- **Lecturer:** Dr. Bernd Westphal
- **Support:** ./. 
- **Homepage:**
  
  http://swt.informatik.uni-freiburg.de/teaching/WS2014-15/sdmauml

- **Questions:**
  - **“online”:**
    1. ask immediately or in the break
  - **“offline”:**
    1. try to solve yourself
    2. discuss with colleagues
    3. Exercises: contact tutor by mail (cf. homepage)
    4. Rest: contact lecturer by mail (cf. homepage) or just drop by: Building 52, Room 00-020
Course language: **English**  
(slides/writing, presentation, questions/discussions)

**Presentation:**  
half slides/half on-screen **hand-writing** — for reasons

**Script/Media:**  
- slides with annotations on **homepage**, 2-up for printing, typically soon **after** the lecture  
- recording on eLectures portal with max. 1 week delay (link on **homepage**)

**Interaction:**  
absence often moaned but **it takes two**, so please ask/comment immediately.
Formalia: Dates/Times

- **Location:**
  - Tuesday, Thursday: here (building 51, room 03-026)

- **Schedule:**

  Week $N$, Thursday, 8–10 **lecture A1** (exercise sheet $A$ [online])
  Week $N + 1$, Tuesday 8–10 **lecture A2**
  Thursday 8–10 **lecture A3**
  Week $N + 2$, Monday, 12:00 (exercises $A$ [early submission])
  Tuesday, 8:00 (exercises $A$ [late submission])
  8–10 **tutorial A**
  Thursday 8–10 **lecture B1** (exercise sheet $B$ [online])

With a prefix of lectures, see homepage for details.
Formalia: Exercises and Tutorials

- **Schedule/Submission:**
  - **hand-out/online** on Thursday after tutorial,
    - **early turn in** on following Monday by 12:00 local time
    - **regular turn in** on following Tuesday by 8:00 local time
  - should work in groups of **approx. 3**, clearly give **names** on submission
  - please submit **electronically** by Mail to B. Westphal (cf. homepage); **paper submissions** are **tolerated**

- **Rating system:** “most complicated rating system ever”
  - **Admission points** (good-will rating, upper bound)
    (“reasonable proposal given student’s knowledge **before** tutorial”)
  - **Exam-like points** (evil rating, lower bound)
    (“reasonable proposal given student’s knowledge **after** tutorial”)

10% **bonus** for **early** submission.

- **Tutorial:** Plenary.
  - Together develop **one** good proposal,
    starting from discussion of the early submissions (anonymous).
• **Break:**
  
  • We’ll have a **10 min. break** in the middle of each event from now on, unless a majority objects **now**.
Exam Admission:

Achieving 50% of the regular admission points in total is sufficient for admission to exam.

Typically, 20 regular admission points per exercise sheet.

Exam Form:

- oral for BSc and on special demand,
- written for everybody else (if sufficiently many candidates remain).

Scores from the exercises do not contribute to the final grade.
Formalia: Evaluation

• Mid-term Evaluation:

  • We will have a mid-term evaluation (early December, roughly 1/3 of the course’s time).

  • If you decide to leave the course earlier you may want to do us a favour and tell us the reasons – by participating in the mid-term evaluation (will be announced on homepage).

  • Note: we’re always interested in comments/hints/proposals/wishes/...

    concerning form or content.

Feel free to approach us (tutors, me) in any form. We don’t bite.
Literature
Literature: UML

- OMG: Unified Modeling Language Specification, Superstructure, 2.1.2
- OMG: Object Constraint Language Specification, 2.0
  All three: http://www.omg.org (cf. hyperlinks on course homepage)

Literature: Modelling


http://www.springerlink.com/content/0170-6012

Questions?

