Softwaretechnik / Software-Engineering

Lecture 11: Structural Software Modelling

2017-06-26

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
Introduction and Vocabulary

Software Modelling

(i) views and viewpoints, the 4+1 view
(ii) model-driven/-based software engineering
(iii) Unified Modelling Language (UML)
(iv) Modelling structure
  a) (simplified) class diagrams
  b) (simplified) object diagrams
  c) (simplified) object constraint logic (OCL)

Principles of Design

a) modularity
b) separation of concerns
c) information hiding and data encapsulation
d) abstract data types, object orientation

Modelling behaviour

a) communicating finite automata
b) Uppaal query language
c) basic state-machines
d) an outlook on hierarchical state-machines

Design Patterns
Software Modelling
- views & viewpoints
- the 4+1 view

Class Diagrams
- concrete syntax,
- abstract syntax,
- class diagrams at work,
- semantics: system states.

Object Diagrams
- concrete syntax,
- dangling references,
- partial vs. complete,
- object diagrams at work.

Software Modelling Cont’d
- An outlook on UML
- model-driven software engineering
Example: Design-Models in Construction Engineering

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. Designmodel

3. System

Observation (1): 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
Example: Design-Models in Construction Engineering

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 model

3. System

Observation (2): Floorplan preserves/determines certain system properties, e.g.,

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

→ find design errors before building the system (e.g. bathroom windows)
A Better Analogy is Maybe Regional Planning
**Model**

**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** (Abbild) 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.
Software Modelling
Views and Viewpoints

view – A representation of a whole system from the perspective of a related set of concerns.

IEEE 1471 (2000)

viewpoint – A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

IEEE 1471 (2000)

- A perspective is determined by concerns and information needs:
  - team leader, e.g., needs to know which team is working on what component,
  - operator, e.g., needs to know which component is running on which host,
  - developer, e.g., needs to know interfaces of other components.
  - etc.
**An Early Proposal: The 4+1 View** *(Kruchten, 1995)*

Newer proposals *(Ludewig and Lichter, 2013)*:

**system view**: how is the system under development integrated into (or seen by) its **environment**; with which other systems (including users) does it **interact** how.

**static view** *(~ developer view)*: components of the architecture, their interfaces and relations. Possibly: assignment of development, test, etc. onto teams.

**dynamic view** *(~ process view)*: how and when are components instantiated and how do they work together at runtime.

**deployment view** *(~ physical view)*: how are component instances mapped onto infrastructure and hardware units.

“Purpose of architecture: **support** functionality; functionality is **not part** of the architecture.” ?!
**Example**: modern cars

- large number of electronic control units (ECUs) spread all over the car,
- which part of the overall software is running on which ECU?
- which function is used when? Event triggered, time triggered, continuous, etc.?

For, e.g., a simple **smartphone app**, process and physical view may be trivial or determined by the employed framework (→ later) – so no need for (extensive) particular documentation.
**Structure vs. Behaviour / Constructive vs. Reflective**

- **Form of the states** in $\Sigma$ (also actions $A$):
  - structure of $S$

- **Computation paths** $\pi$ of $S$:
  - behaviour of $S$

---

**Definition.** Software is a finite description $S$ of a (possibly infinite) set $[S]$ of (finite or infinite) computation paths of the form

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$$

where

- $\sigma_i \in \Sigma, i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A, i \in \mathbb{N}_0$, is called action (or event).

The (possibly partial) function $[\cdot] : S \mapsto [S]$ is called interpretation of $S$.

---

(Harel, 1997) proposes to distinguish **constructive** and **reflective** descriptions of behaviour:

- **constructive**:
  - "constructs [of description] contain information needed in executing the model or in translating it into executable code."
  - $\rightarrow$ **how things are computed**.

- **reflective** (or **assertive**):
  - "[description used] to derive and present views of the model, statically or during execution, or to set constraints on behavior in preparation for verification."
  - $\rightarrow$ **what should (or should not) be computed**.

**Note:** No sharp boundaries! (would be too easy…)
Views and Their Representation

![Diagram of game development process]

- **External inputs**
  - Keyboard
  - Joystick

- **Game Logic**
  - Player scores
  - Interface
  - Physical engine
  - Collision notification

- **Output**
  - Graphics (from ASCII to bitmap; native or via API)
  - Sound

**Gameplay**

- Keyboard
- Joystick

**Engine**

- Physical objects
- Collision notification

**Main External inputs**

- Keyboard
- Joystick

**Game Logic**

- Player scores
- Interface
- Physical engine
- Collision notification

**Output**

- Graphics (from ASCII to bitmap; native or via API)
- Sound

**Notify**

- Update

**Console**

- User
- Game
| VL 10 | • Introduction and Vocabulary
| VL 11 | • Software Modelling
| VL 12 |   (i) views and viewpoints, the 4+1 view
| VL 13 |   (ii) model-driven/-based software engineering
|       |   (iii) Unified Modelling Language (UML)
|       |   (iv) **Modelling structure**
|       |     a) (simplified) class diagrams
|       |     b) (simplified) object diagrams
|       |     c) (simplified) object constraint logic (OCL)
|       | (v) **Principles of Design**
|       |     a) modularity
|       |     b) separation of concerns
|       |     c) information hiding and data encapsulation
|       |     d) abstract data types, object orientation
|       | (vi) **Modelling behaviour**
|       |     a) communicating finite automata
|       |     b) Uppaal query language
|       |     c) basic state-machines
|       |     d) an outlook on hierarchical state-machines
|       | • **Design Patterns**
Software Modelling
- views & viewpoints
- the 4+1 view

Class Diagrams
- concrete syntax,
- abstract syntax,
- class diagrams at work,
- semantics: system states.

Object Diagrams
- concrete syntax,
- dangling references,
- partial vs. complete,
- object diagrams at work.

Software Modelling Cont’d
- An outlook on UML
- model-driven software engineering
... so, off to “technological paradise where [...] everything happens according to the blueprints”.

(Kopetz, 2011; Lovins and Lovins, 2001)
Class Diagrams
where

- \( T_1, \ldots, T_m, 0 \in \mathcal{T} \cup \{ C_{0,1}, C_* \mid C \text{ a class name} \} \)

- \( \mathcal{T} \) is a set of basic types, e.g. \( \text{Int}, \text{Bool}, \ldots \)
Concrete Syntax: Example

Alternative notation for $C_{0,1}$ and $C_\ast$ typed attributes:

Alternative lazy notation for alternative notation:

And nothing else! This is the concrete syntax of **class diagrams** for the **scope of the course**.
Definition. An **Object System Signature** is a 6-tuple

\[ \mathcal{I} = (\mathcal{T}, \mathcal{C}, V, \text{atr}, F, mth) \]

where

- \( \mathcal{T} \) is a set of (basic) **types**, 
- \( \mathcal{C} \) is a finite set of **classes**, 
- \( V \) is a finite set of **typed attributes** \( v : T \), i.e., each \( v \in V \) has type \( T \), 
- \( \text{atr} : \mathcal{C} \rightarrow 2^V \) maps each class to its set of attributes. 
- \( F \) is a finite set of **typed behavioural features** \( f : T_1, \ldots, T_n \rightarrow T \), 
- \( mth : \mathcal{C} \rightarrow 2^F \) maps each class to its set of behavioural features. 
- A type can be a basic type \( \tau \in \mathcal{T} \), or \( C_{0,1} \), or \( C_\ast \), where \( C \in \mathcal{C} \).

**Note:** Inspired by OCL 2.0 standard **OMG (2006)**, Annex A.
**Object System Signature Example**

**Definition.** An *(Object System) Signature* is a 6-tuple

\[ \mathcal{S} = (\mathcal{T}, \mathcal{C}, V, \text{atr}, F, \text{mth}) \]

where

- \( \mathcal{T} \) is a set of (basic) types,
- \( \mathcal{C} \) is a finite set of *classes*,
- \( V \) is a finite set of **typed attributes** \( v : T \), i.e., each \( v \in V \) has type \( T \),
- \( \text{atr} : \mathcal{C} \rightarrow 2^V \) maps each class to its set of attributes,
- \( F \) is a finite set of **typed behavioural features** \( f : T_1, \ldots, T_n \rightarrow T \),
- \( \text{mth} : \mathcal{C} \rightarrow 2^F \) maps each class to its set of behavioural features,
- A type can be a basic type \( \tau \in \mathcal{T} \), or \( C_{0,1} \), or \( C_* \), where \( C \in \mathcal{C} \).

\[
\mathcal{S}_0 = (\{\text{Int, Bool}\}, \\
\{C, D\}, \\
\{x : \text{Int}, p : C_{0,1}, n : C_*\}, \\
\{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \\
\{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int}\}, \\
\{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\})
\]
From Abstract to Concrete Syntax

\[ \mathcal{I} = (\mathcal{T}, C, V, atr, F, mth) \]

- \( \mathcal{T} = \{ \text{Int, Bool} \} \)
- \( C = \{ C, D \} \)
- \( V = \{ n: C_0..1, p: C_0..1, x: \text{Int} \} \)
- \( atr = \{ C \mapsto \{ n, p \}, D \mapsto \{ p, x \} \} \)
- \( F = \{ f: \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int} \} \)
- \( mth = \{ C \mapsto \emptyset, D \mapsto \{ f, \text{get}_x \} \} \)
\[ \mathcal{A}_0 = (\{ \text{Int}, \text{Bool} \}, \{ C, D \}, \{ x : \text{Int}, p : C_{0,1}, n : C\ast \}, \{ C \mapsto \{ p, n \}, D \mapsto \{ p, x \} \}, \{ f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int} \}, \{ C \mapsto \emptyset, D \mapsto \{ f, \text{get}_x \} \}) \]
Class Diagrams at Work
The class diagram syntax can be used to **visualise code**: provide rules which map (parts of) the code to class diagram elements.
Visualisation of Implementation: (Useless) Example

- open favourite IDE,
- open favourite project,
- press “generate class diagram”
- wait... wait... wait...

- ca. 35 classes,
- ca. 5,000 LOC C#
A diagram is a **good diagram** if (and only if?) it serves its **purpose**!

**Note**: a class **diagram** for visualisation may be **partial**.

→ show only the **most relevant** classes and attributes (for the given **purpose**).

**Note**: a signature can be defined by a **set of** class diagrams.

→ use multiple class diagrams with a **manageable** number of classes for different purposes.
(Ambler, 2005)
A More Abstract Class Diagram Semantics
**Definition.** An Object System **Structure** of signature

\[ I = (T, C, V, atr, F, mth) \]

is a domain function \( D \) which assigns to each type a domain, i.e.

- \( \tau \in T \) is mapped to \( D(\tau) \),
- \( C \in C \) is mapped to an infinite set \( D(C) \) of **(object) identities**.
  - object identities of different classes are disjoint, i.e.
    \[ \forall C, D \in C : C \neq D \rightarrow D(C) \cap D(D) = \emptyset, \]
  - on object identities, (only) comparison for equality “\( = \)” is defined.
- \( C_* \) and \( C_{0,1} \) for \( C \in C \) are mapped to \( 2 \cdot D(C) \).

We use \( D(C) \) to denote \( \bigcup_{C \in C} D(C) \); analogously \( D(C_*) \).

**Note:** We identify **objects** and **object identities**, because both uniquely determine each other (cf. OCL 2.0 standard).
Wanted: a structure for signature

\[ S_0 = (\{\text{Int}, \text{Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}) \]

A structure \( \mathcal{D} \) maps

- \( \tau \in \mathcal{T} \) to some \( \mathcal{D}(\tau) \), \( C \in \mathcal{C} \) to some identities \( \mathcal{D}(C) \) (infinite, pairwise disjoint),
- \( C_* \) and \( C_{0,1} \) for \( C \in \mathcal{C} \) to \( \mathcal{D}(C_{0,1}) = \mathcal{D}(C_*) = 2^{\mathcal{D}(C)} \).

\[
\begin{align*}
\mathcal{D}(\text{bool}) &= \{ \text{true, false} \} \\
\mathcal{D}(\text{Int}) &= \mathbb{Z} \\
\mathcal{D}(C) &= \mathbb{N}^+ \times \{C\} = \{1_C, 2_C, 3_C, \ldots \} \\
\mathcal{D}(D) &= \mathbb{N}^+ \times \{D\} = \{1_D, 2_D, 3_D, \ldots \} \\
\mathcal{D}(C_{0,1}) &= \mathcal{D}(C_*) = 2^{\mathcal{D}(C)} \\
\mathcal{D}(D_{0,1}) &= \mathcal{D}(D_*) = 2^{\mathcal{D}(D)}
\end{align*}
\]
Definition. Let $\mathcal{D}$ be a structure of $\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, \text{atr}, F, \text{mth})$. A system state of $\mathcal{I}$ wrt. $\mathcal{D}$ is a type-consistent mapping

$$\sigma : \mathcal{D}(\mathcal{C}) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{T}) \cup \mathcal{D}(\mathcal{C}^*))))$$

That is, for each $u \in \mathcal{D}(\mathcal{C})$, $C \in \mathcal{C}$, if $u \in \text{dom}(\sigma)$

- $\text{dom}(\sigma(u)) = \text{atr}(C)$
- $\sigma(u)(v) \in \mathcal{D}(\mathcal{T})$ if $v : \tau, \tau \in \mathcal{T}$
- $\sigma(u)(v) \in \mathcal{D}(\mathcal{D}^*)$ if $v : D_{0,1}$ or $v : D^*$ with $D \in \mathcal{C}$

We call $u \in \mathcal{D}(\mathcal{C})$ alive in $\sigma$ if and only if $u \in \text{dom}(\sigma)$.

We use $\Sigma_{\mathcal{D}}$ to denote the set of all system states of $\mathcal{I}$ wrt. $\mathcal{D}$. 
System State Examples

\[ S_0 = (\{\text{Int}, \text{Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},
\{f : \text{Int} \mapsto \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\})\]

\[ D(\text{Int}) = \mathbb{Z}, \quad D(C) = \{1_C, 2_C, 3_C, \ldots\}, \quad D(D) = \{1_D, 2_D, 3_D, \ldots\} \]

A system state is a partial function \( \sigma : D(C) \mapsto (V \mapsto (D(T) \cup D(C_*))) \) such that

- \( \text{dom}(\sigma(u)) = \text{atr}(C) \),
- \( \sigma(u)(v) \in D(\tau) \) if \( v : \tau, \tau \in T \),
- \( \sigma(u)(v) \in D(C_*) \) if \( v : D_* \) or \( v : D_{0,1} \) with \( D \in C \).

\[ \sigma_1 = \emptyset \]

\[ \sigma_2 = \{1_C \mapsto \{p \mapsto \{SC_3^n\}, n \mapsto \emptyset\},
\{SC \mapsto \{p \mapsto \emptyset, n \mapsto \{CC_1^n\}\},
\{TD \mapsto \{p \mapsto \{SC_3^n\}, n \mapsto \mathbb{Z}\}\} \} \]
Content

- Software Modelling
  - views & viewpoints
  - the 4+1 view

- Class Diagrams
  - concrete syntax,
  - abstract syntax,
  - class diagrams at work,
  - semantics: system states.

- Object Diagrams
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.

- Software Modelling Cont’d
  - An outlook on UML
  - model-driven software engineering
Object Diagrams
\[
\mathcal{S}_0 = (\{\text{Int}, \text{Bool}\}, \{C, D\}, \{x: \text{Int}, p: C_{0,1}, n: C^*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f: \text{Int} \to \text{Bool}, \text{get}_x: \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}), \quad \mathcal{D}(\text{Int}) = \mathbb{Z}
\]

\[
\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}
\]

- We may represent \(\sigma\) graphically as follows:

This is an object diagram.

- Alternative notation:

- Alternative non-standard notation:
Special Case: Dangling Reference

**Definition.**
Let $\sigma \in \Sigma^\mathcal{D}$ be a system state and $u \in \text{dom}(\sigma)$ an alive object of class $C$ in $\sigma$.

We say $r \in \text{atr}(C)$ is a dangling reference in $u$ if and only if $r : C_{0,1}$ or $r : C_*$ and $u$ refers to a non-alive object via $v$, i.e.

$$\sigma(u)(r) \not\subset \text{dom}(\sigma).$$

**Example:**
- $\sigma = \{ 1_C \mapsto \{ p \mapsto \emptyset, n \mapsto \{ 5_C \} \}, 1_D \mapsto \{ p \mapsto \{ 5_C \}, x \mapsto 23 \} \}$

- Object diagram representation:

```
  \[
  \begin{array}{c}
  1_C : C \\
  p = \emptyset \\
  \end{array}
  \xrightarrow{n} 
  \begin{array}{c}
  5_C : C \\
  X \\
  \end{array}
  \xrightarrow{p} 
  \begin{array}{c}
  1_D : D \\
  x = 23 \\
  \end{array}
  \]
```
Partial vs. Complete Object Diagrams

- By now we discussed “object diagram represents system state”:

\[
\begin{align*}
&\{\{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\},
5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\},
1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\} \\
\end{align*}
\]

What about the other way round…?

- Object diagrams can be partial, e.g.

\[
\begin{array}{c}
1C : C \\
\hline
p = \emptyset \\
n = \emptyset
\end{array}
\quad
\begin{array}{c}
5C : C \\
\hline
p = \emptyset \\
n = \emptyset
\end{array}
\quad
\begin{array}{c}
1D : D \\
\hline
x = 23
\end{array}
\]

→ we may omit information.

- Is the following object diagram partial or complete?

\[
\begin{array}{c}
1C : C \\
\hline
p = \emptyset \\
n = \emptyset
\end{array}
\quad
\begin{array}{c}
5C : C \\
\hline
p = \emptyset \\
n = \emptyset
\end{array}
\quad
\begin{array}{c}
1D : D \\
\hline
x = 23
\end{array}
\]

- If an object diagram
  - has values for all attributes of all objects in the diagram, and
  - if we say that it is meant to be complete
then we can uniquely reconstruct a system state \( \sigma \).
Special Case: Anonymous Objects

If the object diagram

\[
\begin{array}{c}
1_C : C \\
p = \emptyset
\end{array}
\]

\[
\begin{array}{c}
\vdash C \\
p = \emptyset \\
n = \emptyset
\end{array}
\]

\[
\begin{array}{c}
\vdash D \\
x = 23
\end{array}
\]

is considered as complete, then it denotes the set of all system states

\[
\{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{c\}\}\}, c \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, d \mapsto \{p \mapsto \{c\}, x \mapsto 23\}\}
\]

where \( c \in \mathcal{D}(C), \quad d \in \mathcal{D}(D), \quad c \neq 1_C. \)

**Intuition:** different boxes represent different objects.
Object Diagrams at Work
Example: Data Structure (Schumann et al., 2008)

```
BaseNode
parent : BaseNode.
prevSibling : BaseNode.
nextSibling : BaseNode.
firstChild : BaseNode.
lastChild : BaseNode.

Node
    data : T

Node(data : T)

Iterator
    operator++(): Iterator
    operator--(): Iterator
    operator*(): BaseNode

Iterator
    begin_it
    end_it

Forest
    appendTopLevel(data : T)
    appendChild(parent : Iterator, data : T)
    remove(it : Iterator)
    depth(it : Iterator) : int
    end() : Iterator
    begin() : Iterator
    empty() : bool
    size() : int
```
Object Diagrams for Analysis

```
: M
ctime = 27
data = d_1

: M
ctime = 5
data = d_2

: M
ctime = 9
data = d_3

: N
data = d_4

: N
data = d_5
```

```
M
ctime : int?

N
data : Data

prevM
\text{0..1}

nextM
\text{0..1}

prev\text{N}\text{0..1}

next\text{N}\text{0..1}
```
Content

- Software Modelling
  - views & viewpoints
  - the 4+1 view

- Class Diagrams
  - concrete syntax,
  - abstract syntax,
  - class diagrams at work,
  - semantics: system states.

- Object Diagrams
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.

- Software Modelling Cont’d
  - An outlook on UML
  - model-driven software engineering
Tell Them What You’ve Told Them...

- **Software Modelling**: views and viewpoints, e.g. 4+1

- **Class Diagrams** can be used to **graphically**
  - visualise code,
  - define an **object system structure** \( S \).

- An **Object System Structure** \( S \)
  (together with a structure \( D \))
  - defines a set of **system states** \( \Sigma^{D} \).

- A **System State** \( \sigma \in \Sigma^{D} \)
  - can be **visualised** by an **object diagram**.
References
References


