Softwaretechnik / Software-Engineering

Lecture 11: Structural Software Modelling

2017-06-26

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

VL 10

- **Introduction and Vocabulary**
- **Software Modelling**
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)

VL 11

  - (iv) **Modelling structure**
    - a) (simplified) class diagrams
    - b) (simplified) object diagrams
    - c) (simplified) object constraint logic (OCL)

VL 12

  - (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

VL 13

- **Design Patterns**

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

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

http://wikimedia.org (CC nc-sa 3.0, Ottoklages)

3. System

http://wikimedia.org (CC nc-sa 3.0, Bobthebuilder82)

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. Designmodel

http://wikimedia.org (CC nc-sa 3.0, Ottoklages)

3. System

http://wikimedia.org (CC nc-sa 3.0, Bobthebuilder82)

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

(Broaddus, 2010)

http://commons.wikimedia.org/wiki/File:Freiburg_Rieselfeld_2007.jpg,
Norbert Blau, CC BY-SA 3.0

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." ?!

Deployment / Physical View

http://products.bosch-mobility-solutions.com/en/de/driving_safety/driving_safety_systems_for_commercial_vehicles/electronic_systems_1/electronic_systems_3.html – Robert Bosch GmbH

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 Σ (also actions A):
  structure of S
- **Computation paths** π 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

Topic Area Architecture & Design: Content

VL 10

- **Introduction and Vocabulary**
- **Software Modelling**
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)

VL 11

  - (iv) **Modelling structure**
    - a) (simplified) class diagrams
    - b) (simplified) object diagrams
    - c) (simplified) object constraint logic (OCL)

VL 12

  - (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

VL 13

- **Design Patterns**

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

... so, off to "'technological paradise' where [...] everything happens according to the blueprints".

(Kopetz, 2011; Lovins and Lovins, 2001)

Class Diagrams

Class Diagrams: Concrete Syntax

where

- $T_1, \ldots, T_{m,0} \in \mathscr{T} \cup \{\underbrace{C_{0,1}}, \underbrace{C_*} \mid C$ a class name$\}$
- $\mathscr{T}$ is a set of **basic types**, e.g. $Int, Bool, \ldots$.

Concrete Syntax: Example

Alternative notation for $C_{0,1}$ and C_* 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**.

Abstract Syntax: Object System Signature

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

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

where

- $\mathscr{T}$ is a set of (basic) **types**,
- $\mathscr{C}$ 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,
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

powerset of V

- F is a finite set of **typed behavioural features** $f : T_1, \ldots, T_n \to T$,
- $mth : \mathscr{C} \to 2^F$ maps each class to its set of behavioural features.
- A type can be a basic type $\tau \in \mathscr{T}$, or $C_{0,1}$, or C_*, where $C \in \mathscr{C}$.

Note: Inspired by OCL 2.0 standard OMG (2006), Annex A.

Object System Signature Example

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

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

where

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

$$\begin{aligned}
\mathscr{S}_0 = (&\overbrace{\{Int, Bool\}}^{\mathscr{T}}, \\
&\overbrace{\{C, D\}}^{\mathscr{C}}, \\
&\overbrace{\{x : Int, p : C_{0,1}, n : C_*\}}^{V}, \\
&\overbrace{\{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}}^{atr}, \\
&\overbrace{\{f : Int \to Bool, get_x : Int\}}^{F}, \\
&\overbrace{\{C \mapsto \emptyset, D \mapsto \{f, get_x\}\}}^{mth})
\end{aligned}$$

From Abstract to Concrete Syntax

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

- $\mathscr{T} = \{Int, Bool\}$
- $\mathscr{C} = \{C, D\}$
- $V = \{n : C_*, \; p : C_{0,1}, \; x : Int\}$
- $atr = \{C \mapsto \{n, p\}, \; D \mapsto \{p, x\}\}$
- $F = \{f : Int \to Bool, \; get_x : Int\}$
- $mth = \{C \mapsto \emptyset, \; D \mapsto \{f, get_x\}\}$

Once Again: Concrete vs. Abstract Syntax

$$
\begin{aligned}
\mathscr{S}_0 = (&\{Int, Bool\},\\
&\{C, D\},\\
&\{x : Int, p : C_{0,1}, n : C_*\},\\
&\{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},\\
&\{f : Int \to Bool, get_x : Int\},\\
&\{C \mapsto \emptyset, D \mapsto \{f, get_x\}\})
\end{aligned}
$$

Class Diagrams at Work

Visualisation of Implementation

- The class diagram syntax can be used to **visualise code**:
  provide rules which map (parts of) the code to class diagram elements.

```java
package pac;

import pac.D;

public class C {

  public D n;

  public void print_nx() {
    System.out.printf(
      "%i\n", n.get_x() ); };

  public C() {};
}
```

```java
package pac;

import pac.C;

public class D {

  private int x;

  public int get_x()
    { return x; };

  public D() {};
}
```

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#

Visualisation of Implementation: (Useful) Example

- 4, 7..19, 23..* : NOT
- each Tron instance is linked to at least 1 Control instance

- 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.

Literature Recommendation

(Ambler, 2005)

A More Abstract Class Diagram Semantics

Object System Structure

Definition. An Object System **Structure** of signature

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

is a domain function $\mathscr{D}$ which assigns to each type a domain, i.e.

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

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

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

Basic Object System Structure Example

Wanted: a structure for signature

$$\mathscr{S}_0 = (\{Int, Bool\}, \{C, D\}, \{x : Int, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},$$
$$\{f : Int \to Bool, get_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get_x\}\})$$

A structure $\mathscr{D}$ maps

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

$$\begin{aligned}
\mathscr{D}(Bool) &= \{true, false\} \\
\mathscr{D}(Int) &= \mathbb{Z} \\
\mathscr{D}(C) &= \mathbb{N}^+ \times \{C\} = \{1_C, 2_C, 3_C, \dots\} \\
\mathscr{D}(D) &= \mathbb{N}^+ \times \{D\} = \{1_D, 2_D, 3_D, \dots\} \\
\mathscr{D}(C_{0,1}) = \mathscr{D}(C_*) &= 2^{\mathscr{D}(C)} \\
\mathscr{D}(D_{0,1}) = \mathscr{D}(D_*) &= 2^{\mathscr{D}(D)}
\end{aligned}$$

System State

object identities

partial function

attributes

Definition. Let $\mathscr{D}$ be a structure of $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$.
A **system state** of $\mathscr{S}$ wrt. $\mathscr{D}$ is a **type-consistent** mapping

$$\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow \underbrace{(\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_*))}_{\text{values}}).$$

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

- $\mathrm{dom}(\underbrace{\sigma(u)}_{: V \nrightarrow (\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_*))}) = atr(C)$
- $(\sigma(u))(v) \in \mathscr{D}(\tau)$ if $v : \tau, \tau \in \mathscr{T}$
- $\sigma(u)(v) \in \mathscr{D}(D_*)$ if $v : D_{0,1}$ or $v : D_*$ with $D \in \mathscr{C}$

We call $u \in \mathscr{D}(\mathscr{C})$ **alive** in σ if and only if $u \in \mathrm{dom}(\sigma)$.

We use $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ to denote the set of all system states of $\mathscr{S}$ wrt. $\mathscr{D}$.

System State Examples

$$\mathscr{S}_0 = (\{Int, Bool\}, \{C, D\}, \{x : Int, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},$$
$$\{f : Int \to Bool, get_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get_x\}\})$$
$$\mathscr{D}(Int) = \mathbb{Z}, \quad \mathscr{D}(C) = \{1_C, 2_C, 3_C, ...\}, \quad \mathscr{D}(D) = \{1_D, 2_D, 3_D, ...\}$$

A system state is a partial function $\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow (\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_*)))$ such that

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

$\sigma_1 = \emptyset$

$\sigma_2 = \{ 1_C \mapsto \{ p \mapsto \{5_C\}, \; n \mapsto \emptyset \}, \; 5_C \mapsto \{ p \mapsto \emptyset, \; n \mapsto \{1_C\} \}, \; 1_D \mapsto \{ p \mapsto \{5_C\}, \; x \mapsto 27 \} \}$

object

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

Object Diagrams

$$\mathscr{S}_0 = (\{Int, Bool\}, \{C, D\}, \{x : Int, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},$$
$$\{f : Int \to Bool, get_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get_x\}\}), \qquad \mathscr{D}(Int) = \mathbb{Z}$$

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

- We may **represent** σ graphically as follows:

This is an **object diagram**.

- Alternative notation:

- Alternative **non-standard** notation:

Special Case: Dangling Reference

Definition.
Let $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$ be a system state and $u \in \mathrm{dom}(\sigma)$ an alive object of class C in σ.

We say $r \in atr(C)$ **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.

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

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:

Partial vs. Complete Object Diagrams

- By now we discussed "**object diagram represents system state**":

$$\begin{aligned}&\{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\},\\ &\ 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\},\\ &\ 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}\end{aligned} \quad \rightsquigarrow$$

What about the other way round…?

- **Object diagrams** can be **partial**, e.g.

or

$\longrightarrow$ we may omit information.

- Is the following object diagram **partial** or **complete**?

- 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 σ.

Special Case: Anonymous Objects

If the object diagram

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

Intuition: different boxes represent different objects.

Object Diagrams at Work

Example: Data Structure (Schumann et al., 2008)

Example: Illustrative Object Diagram (Schumann et al., 2008)

Object Diagrams for Analysis

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** $\mathscr{S}$.
- An **Object System Structure** $\mathscr{S}$ (together with a structure $\mathscr{D}$)
  - defines a set of **system states** $\Sigma_{\mathscr{S}}^{\mathscr{D}}$.
- A **System State** $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$
  - can be **visualised** by an **object diagram**.

References

References

Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press.

Booch, G. (1993). *Object-oriented Analysis and Design with Applications*. Prentice-Hall.

Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–114.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D. (1997). Some thoughts on statecharts, 13 years later. In Grumberg, O., editor, *CAV*, volume 1254 of *LNCS*, pages 226–231. Springer-Verlag.

Harel, D., Lachover, H., et al. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.

IEEE (2000). *Recommended Practice for Architectural Description of Software-Intensive Systems*. Std 1471.

Jacobson, I., Christerson, M., and Jonsson, P. (1992). *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley.

Kopetz, H. (2011). What I learned from Brian. In Jones, C. B. et al., editors, *Dependable and Historic Computing*, volume 6875 of *LNCS*. Springer.

Kruchten, P. (1995). The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50.

Lovins, A. B. and Lovins, L. H. (2001). *Brittle Power - Energy Strategy for National Security*. Rocky Mountain Institute.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

OMG (2006). Object Constraint Language, version 2.0. Technical Report formal/06-05-01.

OMG (2007). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1990). *Object-Oriented Modeling and Design*. Prentice Hall.

Schumann, M., Steinke, J., Deck, A., and Westphal, B. (2008). Traceviewer technical documentation, version 1.0. Technical report, Carl von Ossietzky Universität Oldenburg und OFFIS.