Topic Area Architecture & Design: Content

- **VL 11**
  - Introduction and Vocabulary
  - Software Modelling
    - model; views / viewpoints: 4+1 view
  - Modelling structure
    - (simplified) class & object diagrams
    - (simplified) object constraint logic (OCL)

- **VL 12**
  - Principles of Design
    - modularity, separation of concerns
    - information hiding and data encapsulation
    - abstract data types, object orientation

- **VL 13**
  - Design Patterns
    - communicating finite automata (CFA)
    - Uppaal query language
    - CFA vs. Software
    - Unified Modelling Language (UML)
    - basic state-machines
    - an outlook on hierarchical state-machines

- **VL 14**
  - Model-driven/-based Software Engineering
Vocabulary

- System, Architecture, Design

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
Vocabulary

**system**—A collection of components organized to accomplish a specific function or set of functions.

IEEE 1471 (2000)

**software system**—A set of software units and their relations, if they together serve a common purpose.

This purpose is in general complex, it usually includes, next to providing one (or more) executable program(s), also the organisation, usage, maintenance, and further development.

Ludewig and Lichter, 2013

**component**—One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components.

IEEE 610.12 (1990)

**software component**—An architectural entity that

1. encapsulates a subset of the system's functionality and/or data,
2. restricts access to that subset via an explicitly defined interface, and
3. has explicitly defined dependencies on its required execution context.

Taylor et al., 2010
module – (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine. (2) A logically separable part of a program. IEEE 610.12 (1990)

module – A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers. (Ludewig and Lichter, 2013)

interface – A boundary across which two independent entities meet and interact or communicate with each other. (Bachmann et al., 2002)

interface (of component) – The boundary between two communicating components. The interface of a component provides the services of the component to the component’s environment and/or requires services needed by the component from the requirement. (Ludewig and Lichter, 2013)

**Even More Vocabulary**

design – (1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component. (2) The result of the process in (1). IEEE 610.12 (1990)

architecture – The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. IEEE 1471 (2000)

software architecture – The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass et al., 2003)

architectural description – A model – document, product or other artifact – to communicate and record a system’s architecture. An architectural description conveys a set of views each of which depicts the system by describing domain concerns. (Ellis et al., 1996)
Goals and Relevance of Design

- The **structure** of something is the set of **relations between its parts**.
- Something not built from (recognisable) parts is called **unstructured**.

**Design...**

(i) **structures** a system into **manageable** units (yields software architecture).
(ii) **determines** the approach for realising the required software.
(iii) provides **hierarchical structuring** into a **manageable** number of units at each hierarchy level.

Oversimplified process model "Design":

```
req  |  design  |  spec  |  code  
-----|----------|--------|--------
program design
implementation
```
Modelling
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**.

---

**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
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.

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
Examples for Software Models?

Example: Vending Machine

- Requirement: Buy Water
  - We (only) accept the software if,
    - (i) When we insert 0.50 €,
    - (ii) and press the water button (and no other button),
    - (iii) and there is water in stock,
    - (iv) Get a drink for free (and nothing else).

  then
  - (i) Insert one 1 euro coin.
  - (ii) Press the 'softdrink' button.
  - (iii) Do not insert any more money.
  - (iv) There is water in stock.

- Negative scenario: A Drink for Free
  - We (only) accept the software if,
    - (i) Insert one 50 cent coin.
    - (ii) Press the 'softdrink' button.
    - (iii) Do not insert any more money.
    - (iv) Get two softdrinks.
**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)*

![Diagram of the 4+1 View model](image)

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?
- **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 Σ (and actions in 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

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

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 $S$. 
Software is a finite description $S$ of a (possibly infinite) set $\llbracket S \rrbracket$ 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 $\llbracket \cdot \rrbracket : S \mapsto \llbracket S \rrbracket$ 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."
  - 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."
  - what should (or should not) be computed.

**Note**: No sharp boundaries! (would be too easy...)

---

**Views and Their Representation**

- Analyst

- Diagrams and representations of models and views.
- Introduction and Vocabulary
- Software Modelling
  - model views / viewpoints, 4+1 view
- Modelling structure
  - (simplified) class & object diagrams
  - (simplified) object constraint logic (OCL)
- Principles of Design
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
- Design Patterns
- Modelling behaviour
  - communicating finite automata (CFA)
  - Uppaal query language
- CFA vs. Software
- Unified Modelling Language (UML)
  - basic state-machines
  - an outlook on hierarchical state-machines
- Model-driven/-based Software Engineering
Class Diagrams: Concrete Syntax

where
- $T_1, \ldots, T_{m,0} \in \mathcal{T} \cup \{C_{0,1}, C_* | C \text{ a class name}\}$
- $\mathcal{T}$ 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

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

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 $C \in \mathcal{C}$.

Note: Inspired by OCL 2.0 standard OMG (2006), Annex A.
Definition. An (Object System) Signature is a 6-tuple \( S = (T, C, V, \text{atr}, F, \text{mth}) \) where

- \( T \) is a set of (basic) types,
- \( C \) is a finite set of classes,
- \( V \) is a finite set of typed attributes \( v : T \), i.e., each \( v \in V \) has type \( T \),
- \( \text{atr} : \mathbb{C} \to 2^V \) maps each class to its set of attributes,
- \( F \) is a finite set of typed behavioural features \( f : T_1, \ldots, T_n \to T \),
- \( \text{mth} : \mathbb{C} \to 2^F \) maps each class to its set of behavioural features.

A type can be a basic type \( \tau \in T_0 \), or \( C_0, C_1, C^* \), where \( C \in \mathbb{C} \).

\[ S_0 = (\{\text{Int, Bool}\}, q, T, C, V, \text{atr}, F, \text{mth}) \]

From Abstract to Concrete Syntax

\[ x : \text{Int}, p : C_0,1, n : C_1 \text{, get}_x : \text{Int} \]

\[ C \mapsto \{p, n\}, D \mapsto \{p, x\} \]

\[ \text{mth} = \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\} \]
\[ \mathcal{A}_0 = (\{\text{Int, Bool}\}, \\
(C, D), \\
\{x : \text{Int}, p : C, n : C\}, \\
\{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \\
\{f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int}\} \\
\{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}) \]

**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**

- 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#

Visualisation of Implementation: (Useful) Example

- 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.
Tell Them What You’ve Told Them...

- **Design** structures a system into **manageable units**.

- (Software) **Model**: a concrete or mental **image** or **archetype** with
  - **image** / **reduction** / **pragmatics** property.

- Towards **Software Modelling**:
  - Views and Viewpoints, e.g. 4+1,
  - **Structure** vs. **Behaviour**

- **Class Diagrams** can be used to **describe** system structures **graphically**
  - visualise code,
  - define an **object system structure** $\mathcal{S}$.

- An **Object System Structure** $\mathcal{S}$
  (together with a structure $\mathcal{S}$)
  - defines a set of **system states** $\Sigma_{\mathcal{S}}$;
  - a system state is structured according to $\mathcal{S}$.

- A **System State** $\sigma \in \Sigma_{\mathcal{S}}$
  - can be visualised by an **object diagram**.

(Ambler, 2005)
References


