Softwaretechnik / Software-Engineering

Lecture 10: Structural Software Modelling

2019-06-17

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

- 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
Content

- **Vocabulary**
  - System, Architecture, Design
- **Modelling**
  - Software Modelling
    - views & viewpoints
    - the 4+1 view
  - Class Diagrams
    - concrete syntax,
    - abstract syntax,
    - semantics: system states.
    - class diagrams at work,
  - Object Diagrams
    - concrete syntax,
    - dangling references,
    - partial vs. complete,
    - object diagrams at work.

Vocabulary
**Vocabulary**

**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)

**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)

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)

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)

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)
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)
**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 requested. (Ludewig and Lichter, 2013)

---

**Once Again, Please**

System consists of 1 or more Component has Component Interface

Software System Software Component may be a Module

System Architecture Software Architecture is the result of Design is described by Architectural Description

**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)
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":

![Process Model Diagram]
Goals and Relevance of Design: An Analogy

Design...
(i) structures a system into manageable units [...],
(ii) determines the approach for realising the [system],
(iii) provides hierarchical structuring into a manageable number of units at each hierarchy level.

Regional Planning: Design a Quarter.

Building Engineering: Design a House.

Topic Area Architecture & Design: Content

VL 10
- Introduction and Vocabulary
- Software Modelling
  - model views / viewpoints: 4+1 view
- Modelling structure
  - (simplified) Class & Object diagrams
  - (simplified) Object Constraint Logic (OCL)

VL 11
- Principles of Design
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation

VL 12
- 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

VL 13
- Model-driven/-based Software Engineering
Modelling

- Vocabulary
  - System, Architecture, Design

- Modelling
  - Software Modelling
    - views & viewpoints
    - the 4+1 view
  - Class Diagrams
    - concrete syntax,
    - abstract syntax,
    - semantics: system states.
    - class diagrams at work,
  - Object Diagrams
    - concrete syntax,
    - dangling references,
    - partial vs. complete,
    - object diagrams at work.
**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*.
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
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),

→ architects can efficiently work on appropriate level of abstraction

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

Software Modelling
Examples for (Software) Models?

From Process Model to Concrete Process

compose

Building Blocks

concretise

Plan

Process

...
Examples for (Software) Models?

Decision Tables as Specification Language

- Decision Tables can be used to objectively describe desired software behaviour.

- Example: Over developers, please provide a program such that
  - in each situation (button pressed, ventilation on/off),
  - whatever the software does (action start/stop)
  - is shown by decision table 2.

<table>
<thead>
<tr>
<th>Action</th>
<th>Next Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>button pressed</td>
<td>X</td>
</tr>
<tr>
<td>ventilation on/off</td>
<td>X</td>
</tr>
<tr>
<td>start ventilation</td>
<td>X</td>
</tr>
<tr>
<td>stop ventilation</td>
<td>X</td>
</tr>
</tbody>
</table>
Examples for (Software) Models?

Example: Vending Machine

- Requirement: Buy Water
  - We only accept the software if:
  - (i) We insert 0.50 €.
  - (ii) Press the 'water' button (and no other button).
  - (iii) Do not insert any more money.
  - (iv) Get water for free.
  - We accept the software if:
    - (i) Insert one 1 euro coin
    - (ii) and press the 'water' button
    - (iii) Get water
    - (iv) Get two softdrinks.

- Negative scenario: A Drink for Free
  - We don’t accept the software if:
    - (i) It is possible to get a drink for free.
    - (ii) Insert one 1 euro coin
    - (iii) Press the 'softdrink' button.
    - (iv) Do not insert any more money.
    - (v) Get two softdrinks.
views — A representation of a whole system from the perspective of a related set of concerns.  

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.
**Views and Viewpoints**

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

**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)*

- Logical View
- Development View
- Process View
- Physical View

- end-user functionality
- implementation, management
- integrators, performance, scalability
- system engineers, topology, communication
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.
Definition. Software is a finite description $\mathcal{S}$ of a (possibly infinite) set $[\mathcal{S}]$ 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 $S$. 

- Form of the states in $\Sigma$ (and actions in $A$): structure of $\mathcal{S}$
- Computation paths $\pi$ of $\mathcal{S}$: behaviour of $\mathcal{S}$
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 reflective and constructive descriptions of behaviour:

- **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.

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

- **Form of the states** in $\Sigma$ (and actions in $A$):
  - structure of $S$
- **Computation paths** $\pi$ of $S$:
  - behaviour of $S$

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

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

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

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

---

**Content**

- **Vocabulary**
  - System, Architecture, Design
- **Modelling**
- **Software Modelling**
  - views & viewpoints
  - the 4+1 view
- **Class Diagrams**
  - concrete syntax,
  - abstract syntax,
  - semantics: system states.
  - class diagrams at work,
- **Object Diagrams**
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
Class Diagrams
Class Diagrams: Concrete Syntax

where

- $T_1, \ldots, T_{m,0} \in \mathcal{T} \cup \{C_0, 1, 0 | C \text{ a class name}\}$
- $\mathcal{T}$ is a set of basic types, e.g. $\text{Int, Bool, ...}$

Concrete Syntax: Example

```
C
n : C
p : C_0
```

```
D
x : Int
p : C_0
f(Int) : Bool
get_x() : Int
```
Concrete Syntax: Example

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

Alternative lazy notation for alternative notation:
**Concrete Syntax: Example**

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>n: C*</td>
<td></td>
</tr>
<tr>
<td>p: C_{0,1}</td>
<td></td>
</tr>
<tr>
<td>x: Int</td>
<td></td>
</tr>
<tr>
<td>p: C_{0,1}</td>
<td></td>
</tr>
<tr>
<td>f(Int): Bool</td>
<td></td>
</tr>
<tr>
<td>get_x(): Int</td>
<td></td>
</tr>
</tbody>
</table>

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

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

where

- $\mathcal{T}$ is a set of (basic) types,
- $\mathcal{C}$ is a finite set of classes,
- $\mathcal{V}$ is a finite set of typed attributes $v : T$, i.e., each $v \in \mathcal{V}$ has type $T$,
- $\text{atr} : \mathcal{C} \rightarrow 2^\mathcal{V}$ maps each class to its set of attributes.
- $\mathcal{F}$ is a finite set of typed behavioural features $f : T_1, \ldots, T_n \rightarrow T$,
- $\text{mth} : \mathcal{C} \rightarrow 2^\mathcal{F}$ 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 \( T \),
- \( \text{atr} : C \rightarrow 2^V \) maps each class to its set of attributes,
- \( F \) is a finite set of typed behavioural features \( f : T_1, \ldots, T_n \rightarrow T \),
- \( \text{mth} : C \rightarrow 2^F \) maps each class to its set of behavioural features.
- A type can be a basic type \( \tau \in T \), or \( C_0, 1, \) or \( C^\ast \), where \( C \in C \).

\[ S_0 = (\{\text{Int, Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_0, n : C^\ast\}, \{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}) \]
From Abstract to Concrete Syntax

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

- \( \mathcal{T} = \{ \text{Int, Bool} \} \)
- \( \mathcal{C} = \{ \text{C, D} \} \)
- \( \mathcal{V} = \{ x : \text{Int}, p : \mathcal{C}_{\text{D}}, n : \mathcal{C}_{\times} \} \)
- \( \mathcal{atr} = \{ \text{C} \mapsto \{ n, p \}, \text{D} \mapsto \{ x, p \} \} \)
- \( \mathcal{F} = \{ f : \text{Int} \rightarrow \text{Bool}, \text{get}_x() : \text{Int} \} \)
- \( \mathcal{mth} = \{ \text{C} \mapsto \emptyset, \text{D} \mapsto \{ f, \text{get}_x() \} \} \)

Once Again: Concrete vs. Abstract Syntax
\[ S_0 = (\{ \text{Int, Bool} \}, \{ C, D \}, \{ x : \text{Int}, p : C_0, n : C_1 \}, \{ C \mapsto \{ p, n \}, D \mapsto \{ p, x \} \}, \{ f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int} \}, \{ C \mapsto \emptyset, D \mapsto \{ f, \text{get}_x \} \}) \]
\[ S_0 = \left( \{ \text{Int, Bool} \}, \{ C, D \}, \{ x : \text{Int}, p : C_0, n : C_1 \}, \{ C \mapsto \{ p, n \}, D \mapsto \{ p, x \} \}, \{ f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int} \}, \{ C \mapsto \emptyset, D \mapsto \{ f, \text{get}_x \} \} \right) \]
The class diagram syntax can be used to **visualise code**:

Provide rules which map (parts of) the code to class diagram elements.
• open favourite IDE,
• open favourite project,
• press "generate class diagram"
• wait... wait...
Visualisation of Implementation: (Useless) Example

- open favourite IDE,
- open favourite project,
- press "generate class diagram"
- wait... wait... wait...
• A diagram is a good diagram if (and only if) it serves its purpose!
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

The Elements of UML 2.0 Style
Scott W. Ambler

(Ambler, 2005)
A More Abstract Class Diagram Semantics
Definition. An Object System Structure of signature
\[ \mathcal{S} = (T, C, V, atr, F, mth) \]
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 "\( \approx \)" is defined.
- \( C_0, C_1 \) for \( C \in \mathcal{C} \) are mapped to \( D(C_0,1) = 2^{D(C)} \).

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

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
\[ \mathcal{S}_0 = (\{ \text{Int}, \text{Bool} \}, \{ C, D \}, \{ x : \text{Int}, p : C_0,1, n : C_1 \}, \{ C \mapsto \{ p, n \}, D \mapsto \{ p, x \} \}, \{ f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int} \}, \{ C \mapsto \emptyset, D \mapsto \{ f, \text{get}_x \} \}) \]

A structure \( D \) maps

- \( \tau \in T \) to some \( D(\tau), C \in C \) to some identities \( D(C) \) (infinite, pairwise disjoint),
- \( C_0, C_1 \) for \( C \in \mathcal{C} \) to \( D(C_0,1) = 2^{D(C)} \).

\[
\begin{align*}
D(\text{Int}) &= \mathbb{Z} \\
D(C) &= \mathbb{N} \times \{ C \} = \{ c_1, z_1, z_2, \ldots \} \\
D(D) &= \mathbb{N} \times \{ D \} = \{ d_0, z_0, z_1, \ldots \} \\
D(C_0,1) &= 2^{D(C)} \\
D(D_0,1) &= 2^{D(D)}
\end{align*}
\]

\( D' \) maps

\[
\begin{align*}
D'(\text{Int}) &= \{ 3, 7, 11, \ldots \} \\
D'(C) &= \{ 1, 4, 5, 6, \ldots \} \\
D'(D) &= \{ 0, 1, 2, 3, \ldots \} \\
D'(C_0,1) &= 2^{D(C)} \\
D'(D_0,1) &= 2^{D(D)}
\end{align*}
\]
Definition. Let $D$ be a structure of $S = (T, C, V, atr, F, mth)$. A system state of $S$ wrt. $D$ is a type-consistent mapping $\sigma : D(C) \nrightarrow (V \nrightarrow (D(T) \cup D(C^*))).$

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

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

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

We use $\Sigma^D_S$ to denote the set of all system states of $S$ wrt. $D$.

System State Examples

$S_0 = (\{\text{Int}, \text{Bool}\}, \{C, D\}, \{x : \text{Int}, \, p : C_{0,1}, \, n : C_*\}, \{C \rightarrow \{p, n\}, D \rightarrow \{p, x\}\},$

$\{f : \text{Int} \rightarrow \text{Bool}, \, \text{get}_x : \text{Int}\}, \{C \rightarrow \emptyset, D \rightarrow \{f, \text{get}_x\}\})$

$D(\text{Int}) = \mathbb{Z}, \quad D(C) = \{1C, 2C, 3C, \ldots\}, \quad D(D) = \{1D, 2D, 3D, \ldots\}$

A system state is a partial function $\sigma : D(C) \rightarrow (V \rightarrow (D(T) \cup D(C^*)))$ 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 $D \in C$. 

$\sigma_1 = \{2D \mapsto \{p \mapsto \{1C\}, \, n \mapsto \emptyset\}, \quad 1D \mapsto \{p \mapsto \{2C\}, \, x \mapsto 2D\}\}$

$\sigma_2 = \emptyset$

$\sigma_3 = \{S_{\emptyset} \mapsto \{p \mapsto \{C_{0,1}\}, \, n \mapsto \emptyset\}\}$
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() {};
}
```
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...
• open favourite IDE,
• open favourite project,
• press "generate class diagram"
• wait... wait...
• 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.
Content

- Vocabulary
  - System, Architecture, Design

- Modelling
  - Software Modelling
    - views & viewpoints
    - the 4+1 view
  - Class Diagrams
    - concrete syntax,
    - abstract syntax,
    - semantics: system states.
    - class diagrams at work,
  - Object Diagrams
    - concrete syntax,
    - dangling references,
    - partial vs. complete,
    - object diagrams at work.
Object Diagrams

\[ \mathcal{X}_0 = (\{\text{Int, Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_0, 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}\}\}, \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:

- $1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}$
- $5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}$
- $1_D \mapsto \{p \mapsto \emptyset, n \mapsto 23\}$

This is an object diagram.

- Alternative notation:
\[ S_0 = (\{ \text{Int, Bool} \}, \{ C, D \}, \{ x : \text{Int}, p : C_{0,1}, n : C_* \}, \{ C \mapsto \{ p, n \}, D \mapsto \{ p, x \} \}, \{ f : \text{Int} \rightarrow \text{Bool}, \text{get}_{-x} : \text{Int} \}, \{ C \mapsto \emptyset, D \mapsto \{ f, \text{get}_{-x} \} \}), \quad \mathcal{D}(\text{Int}) = \mathbb{Z} \]

\[ \sigma = \{ 1_C \mapsto \{ p \mapsto 0, n \mapsto \{ 5_C \} \}, 5_C \mapsto \{ p \mapsto 0, n \mapsto 5 \}, 1_D \mapsto \{ p \mapsto 5_C, x \mapsto 23 \} \} \]

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

This is an object diagram.

- Alternative notation:

- Alternative non-standard notation:

This is an object diagram.
Definition. Let $\sigma \in \Sigma^D$ 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:

```
1_C: C  n
   \mapsto  p \mapsto \emptyset, \quad \mapsto 5_C
1_D: D  x
   \mapsto  p \mapsto \{5_C\}, \quad \mapsto x \mapsto 23
```
• By now we discussed “object diagram represents system state”:

\[
\begin{align*}
1_C &: \{p \mapsto 0, n \mapsto \{5_C\}\}, \\
5_C &: \{p \mapsto 0, n \mapsto \emptyset\}, \\
1_D &: \{p \mapsto \{5_C\}, x \mapsto 23\}
\end{align*}
\]

What about the other way round...?

• Object diagrams can be partial, e.g.

\[
\begin{align*}
1_C &: \emptyset \\
5_C &: \emptyset \\
1_D &: D x = 23
\end{align*}
\]

→ we may omit information.

• Is the following object diagram partial or complete?

\[
\begin{align*}
1_C &: \emptyset \\
1_D &: D x = 23
\end{align*}
\]
• By now we discussed “object diagram represents system state“:

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

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

What about the other way round...?

• Object diagrams can be partial, e.g.

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

→ we may omit information.

• Is the following object diagram partial or complete?

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

• 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 σ.

\[
\text{Special Case: Anonymous Objects}
\]

If the object diagram

\[
\begin{array}{c}
\begin{array}{c}
\text{1}_C \\
\text{5}_C \\
\text{1}_D
\end{array}
\end{array}
\begin{array}{c}
\cdots
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{p} \\
\text{n}
\end{array}
\end{array}
\end{array}
\]

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

\text{Intuition: different boxes represent different objects.}
Example: Data Structure (Schumann et al., 2008)
Example: Illustrative Object Diagram (Schumann et al., 2008)
Content

- Vocabulary
  - System, Architecture, Design

- Modelling

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

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

- Object Diagrams
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
• **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 $\mathcal{D}$)
  • defines a set of *system states* $\Sigma_{\mathcal{D}}$;
  • a system state is *structured* according to $\mathcal{S}$.

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

---

References


