

Softwaretechnik / Software-Engineering

Lecture 10: Structural Software Modelling

2019-06-17

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

- 10 - 2019-06-17 - main -

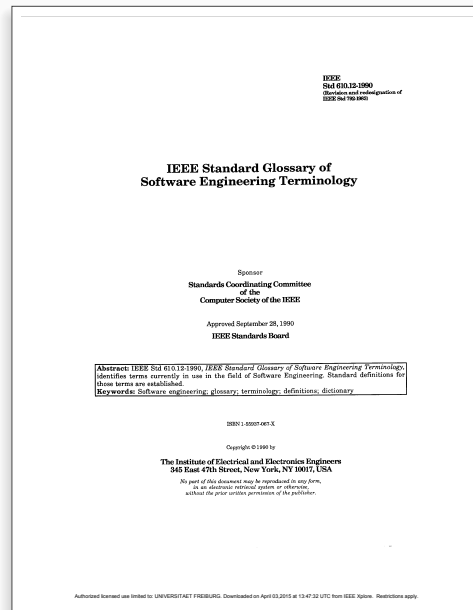
Topic Area Architecture & Design: Content

VL 10	<ul style="list-style-type: none">● Introduction and Vocabulary● Software Modelling<ul style="list-style-type: none">● model; views / viewpoints; 4+1 view
⋮	
VL 11	<ul style="list-style-type: none">● Modelling structure<ul style="list-style-type: none">● (simplified) Class & Object diagrams● (simplified) Object Constraint Logic (OCL)
⋮	
VL 12	<ul style="list-style-type: none">● Principles of Design<ul style="list-style-type: none">● modularity, separation of concerns● information hiding and data encapsulation● abstract data types, object orientation
⋮	
VL 13	<ul style="list-style-type: none">● Design Patterns● Modelling behaviour<ul style="list-style-type: none">● Communicating Finite Automata (CFA)● Uppaal query language● CFA vs. Software● Unified Modelling Language (UML)<ul style="list-style-type: none">● basic state-machines● an outlook on hierarchical state-machines
⋮	
	● Model-driven/-based Software Engineering

- 10 - 2019-06-17 - Slidecontent -

- 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)

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)

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)

Vocabulary Cont'd

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)

Vocabulary Cont'd

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)

Even More Vocabulary

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)

Even More Vocabulary

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)

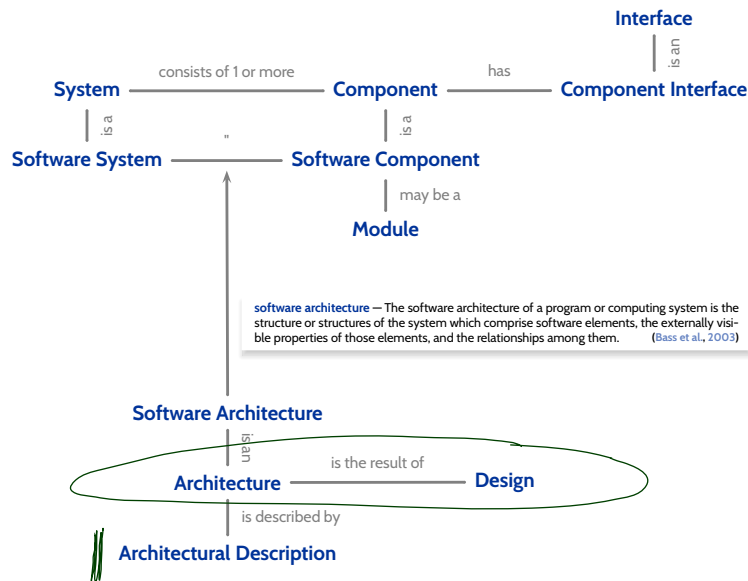
interface— A boundary across which two independent entities meet and interact or communicate with each other. (Bachmann et al., 2002)

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 requirements. (Ludewig and Lichter, 2013)

— 10 - 2019-04-17 - Software -

8/61

Once Again, Please



— 10 - 2019-04-17 - Software -

9/61

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**.

– 10 – 2019-Q4-17 – Software –

10/61

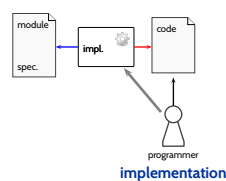
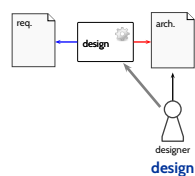
Goals and Relevance of Design

- 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”:



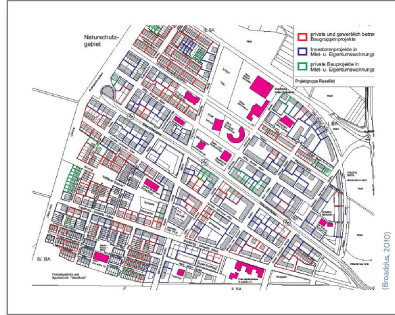
– 10 – 2019-Q4-17 – Software –

10/61

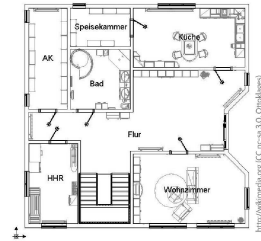
Goals and Relevance of Design: An Analogy

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.

–10– 2019-04-17 – Scheinero –

11/61

Topic Area Architecture & Design: Content

VL 10	<ul style="list-style-type: none">• Introduction and Vocabulary• Software Modelling<ul style="list-style-type: none">• <u>model, views / viewpoints; 4+1 view</u>• Modelling structure<ul style="list-style-type: none">• (simplified) Class & Object diagrams
VL 11	<ul style="list-style-type: none">• (simplified) Object Constraint Logic (OCL)• Principles of Design<ul style="list-style-type: none">• modularity, separation of concerns• information hiding and data encapsulation• abstract data types, object orientation
VL 12	<ul style="list-style-type: none">• Design Patterns• Modelling behaviour<ul style="list-style-type: none">• Communicating Finite Automata (CFA)• Uppaal query language
VL 13	<ul style="list-style-type: none">• CFA vs. Software<ul style="list-style-type: none">• Unified Modelling Language (UML)<ul style="list-style-type: none">• basic state-machines• an outlook on hierarchical state-machines• Model-driven/-based Software Engineering

–10– 2019-04-17 – Stückelmeier –

12/61

- **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.

Modelling

Definition. (Folk) A **model** is an abstract, formal, mathematical representation or description of structure or behaviour of a (software) system.

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.

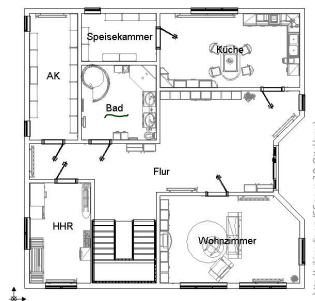
3. System



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

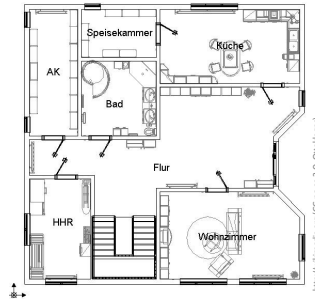


Example: Design-Models in Construction Engineering

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



<http://wikimedia.org>
(CC BY-SA 3.0, Bildhauerei)

Observation (1): Floorplan **abstracts** from certain system properties, e.g. ...

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

→ architects can efficiently work on appropriate level of abstraction

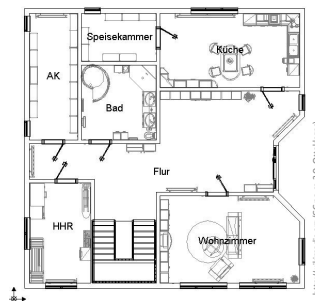
16/61

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



<http://wikimedia.org>
(CC BY-SA 3.0, Bildhauerei)

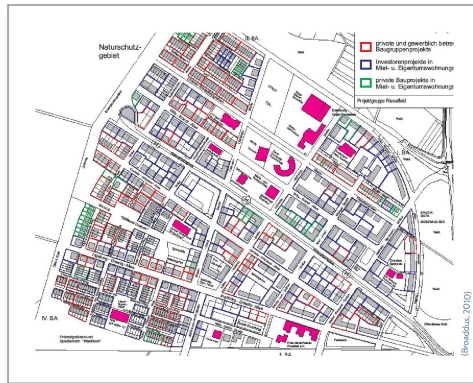
Observation (2): Floorplan **preserves/determines** certain system properties, e.g.,

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

→ find design errors before building the system (e.g. bathroom windows)

16/61

A Better Analogy is Maybe Regional Planning



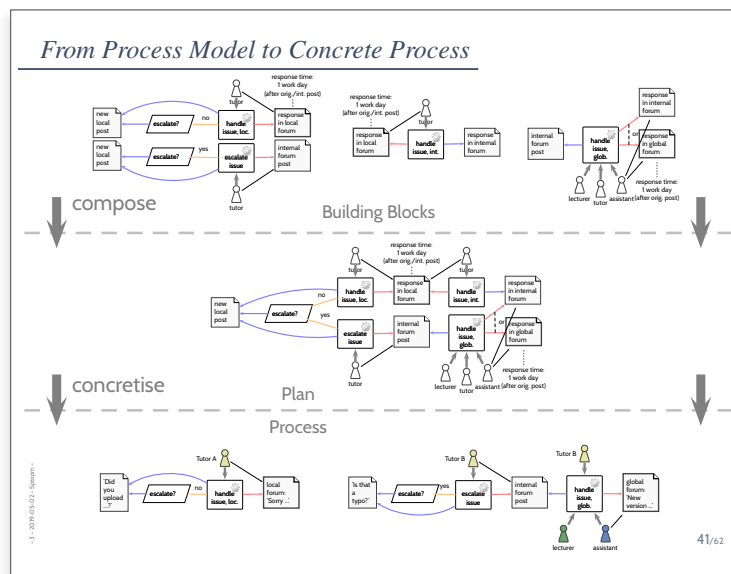
-10-2018-04-17-Smodel-

17/61

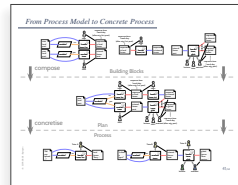
Software Modelling

-10-2018-04-17-main-

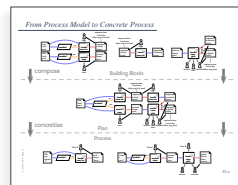
18/61



Examples for (Software) Models?



Examples for (Software) Models?



Decision Tables as Specification Language

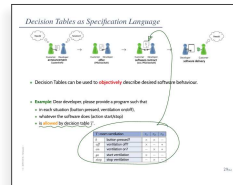
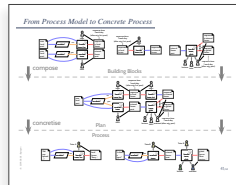


- Decision Tables can be used to **objectively** describe desired software behaviour.

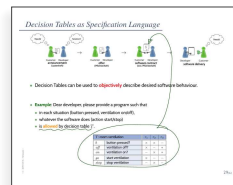
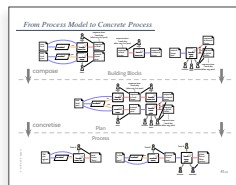
- Example:** Dear developer, please provide a program such that
 - in each situation (button pressed, ventilation on/off),
 - whatever the software does (action start/stop)
 - is allowed by decision table T .

T : room ventilation		r_1	r_2	r_3
b	button pressed?	x	x	—
off	ventilation off?	x	—	+
on	ventilation on?	—	x	+
go	start ventilation	x	—	—
stop	stop ventilation	—	x	—

Examples for (Software) Models?



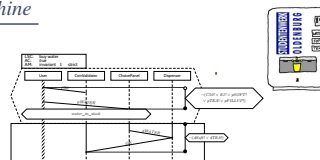
Examples for (Software) Models?



Example: Vending Machine

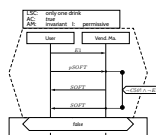
Requirement: Buy Water

- We (only) accept the software if,
- Whenever we insert 0.50 €,
 - and press the 'water' button (and no other button),
 - and there is water in stock,
 - then we get water (and nothing else).



Negative scenario: A Drink for Free

- We don't accept the software if it is possible to get a drink for free.
- Insert one 1 euro coin.
 - Press the 'softdrink' button.
 - Do not insert any more money.
 - 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)

Views and Viewpoints

view — A representation of a whole system from the perspective of a related set of concerns.
IEEE 1471 (2000)

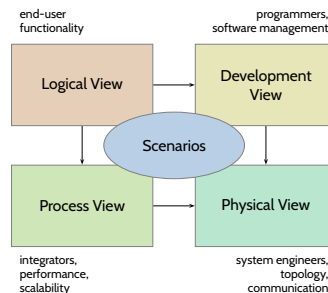
viewpoint — A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.
IEEE 1471 (2000)

- 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.

— 10 — 2018-Q4-17 — Swamodell —

20/61

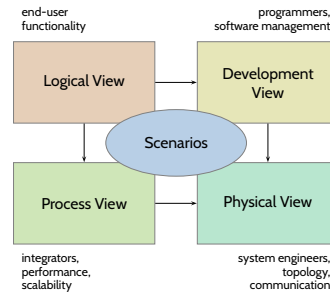
An Early Proposal: The 4+1 View (Kruchten, 1995)



— 10 — 2018-Q4-17 — Swamodell —

21/61

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.

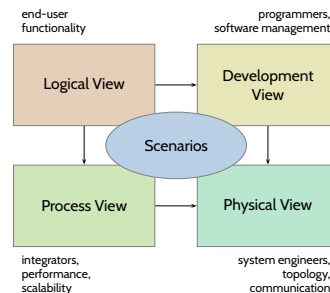
dynamic view (~ process view): how and when are components instantiated and how do they work together at runtime.

deployment view (~ physical view): How are component instances mapped onto infrastructure and hardware units?

– 10 – 2019-04-17 – Svenmodel –

21/61

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.

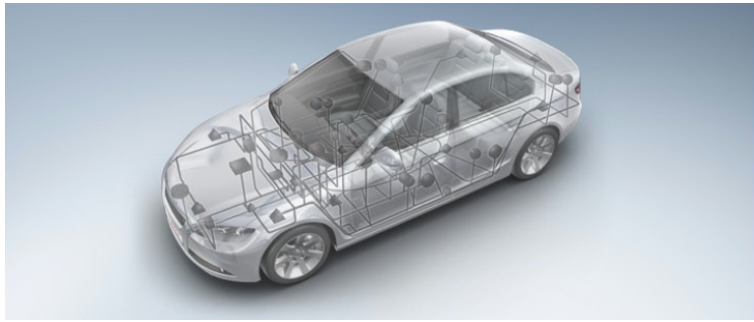
dynamic view (~ process view): how and when are components instantiated and how do they work together at runtime.

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.” ?!)

– 10 – 2019-04-17 – Svenmodel –

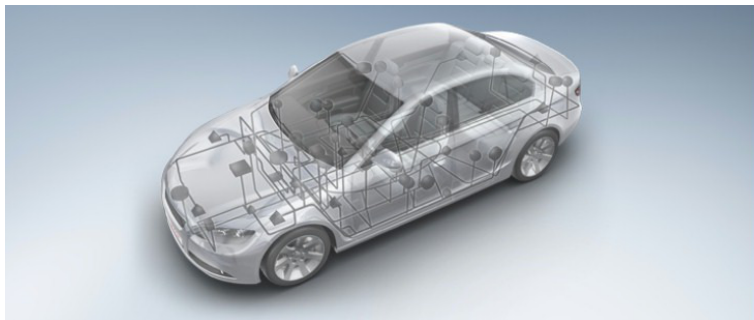
21/61



http://products.bosch-mobility-solutions.com/en/driving_safety/driver_safety_systems_for_commercial_vehicles/electronic_systems_4/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.?

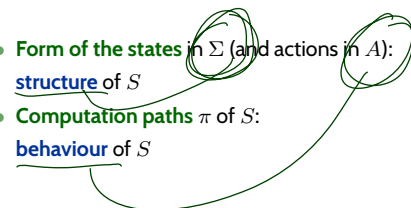


http://products.bosch-mobility-solutions.com/en/driving_safety/driver_safety_systems_for_commercial_vehicles/electronic_systems_4/electronic_systems_3.html — Robert Bosch GmbH

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.

- **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 $\llbracket S \rrbracket$ of (finite or infinite) **computation paths** of the form

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

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 .

- **Form of the states** in Σ (and actions in A):
structure of S
- **Computation paths** π of S :
behaviour of S

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

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

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

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 .

- **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 $\llbracket S \rrbracket$ of (finite or infinite) **computation paths** of the form

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

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
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.*”
→ **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.*”
→ **how things are computed.**

- **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 $\llbracket S \rrbracket$ of (finite or infinite) **computation paths** of the form

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

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 **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.”
 → **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.”
 → **how things are computed**.

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

– 10 – 2019-04-17 – Semmodel –

23/61

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.

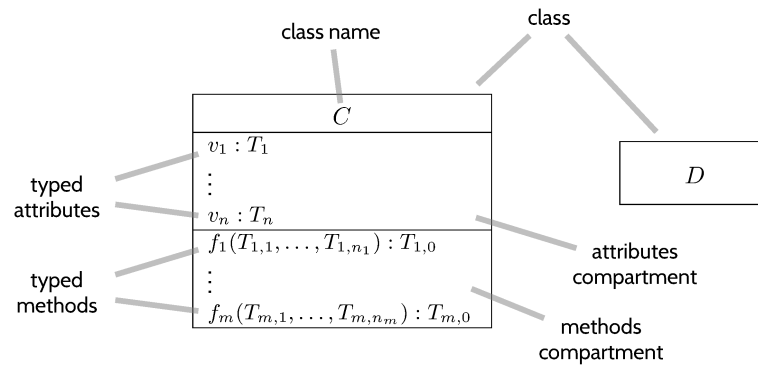
– 10 – 2019-04-17 – Semant –

24/61



Class Diagrams

Class Diagrams: Concrete Syntax



where

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

–10– 2018-Oct-17 – Sunday –

27/61

Concrete Syntax: Example

C
$n : C_*$
$p : C_{0,1}$

D
$x : Int$
$p : C_{0,1}$
$f(Int) : Bool$
$get_x() : Int$

–10– 2018-Oct-17 – Sunday –

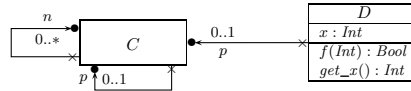
28/61

Concrete Syntax: Example

C
$n : C_*$
$p : C_{0,1}$

D
$x : Int$
$p : C_{0,1}$
$f(Int) : Bool$
$get_x() : Int$

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

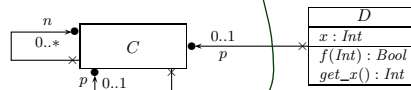


Concrete Syntax: Example

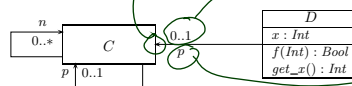
C
$n : C_*$
$p : C_{0,1}$

D
$x : Int$
$p : C_{0,1}$
$f(Int) : Bool$
$get_x() : Int$

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



Alternative lazy notation for alternative notation:

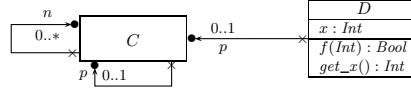


Concrete Syntax: Example

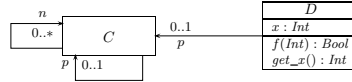
C
$n : C_*$
$p : C_{0,1}$

D
$x : Int$
$p : C_{0,1}$
$f(Int) : Bool$
$get_x() : Int$

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.

28/61

Abstract Syntax: Object System Signature

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

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

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 ,
- $atr : \mathcal{C} \rightarrow 2^V$ maps each class to its set of attributes.
- F is a finite set of typed behavioural features $f : T_1, \dots, T_n \rightarrow T$,
- $mth : \mathcal{C} \rightarrow 2^F$ 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_* , where $C \in \mathcal{C}$.

Note: Inspired by OCL 2.0 standard [OMG \(2006\)](#), Annex A.

29/61

Object System Signature Example

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

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

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 ,
- $atr : \mathcal{C} \rightarrow 2^V$ maps each class to its set of attributes.
- F is a finite set of typed behavioural features $f : T_1, \dots, T_n \rightarrow T$,
- $mth : \mathcal{C} \rightarrow 2^F$ 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_* , where $C \in \mathcal{C}$.

Object System Signature Example

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

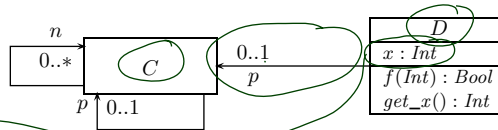
$$\mathcal{S} = (\mathcal{T}, \mathcal{C}, V, atr, F, mth)$$

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 ,
- $atr : \mathcal{C} \rightarrow 2^V$ maps each class to its set of attributes.
- F is a finite set of typed behavioural features $f : T_1, \dots, T_n \rightarrow T$,
- $mth : \mathcal{C} \rightarrow 2^F$ 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_* , where $C \in \mathcal{C}$.

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

From Abstract to Concrete Syntax



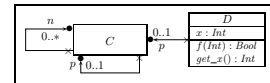
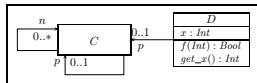
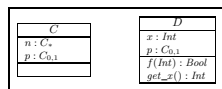
$$\mathcal{S} = (\mathcal{T}, \mathcal{C}, V, atr, F, mth)$$

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

–10–2019-04-17–Samstag–

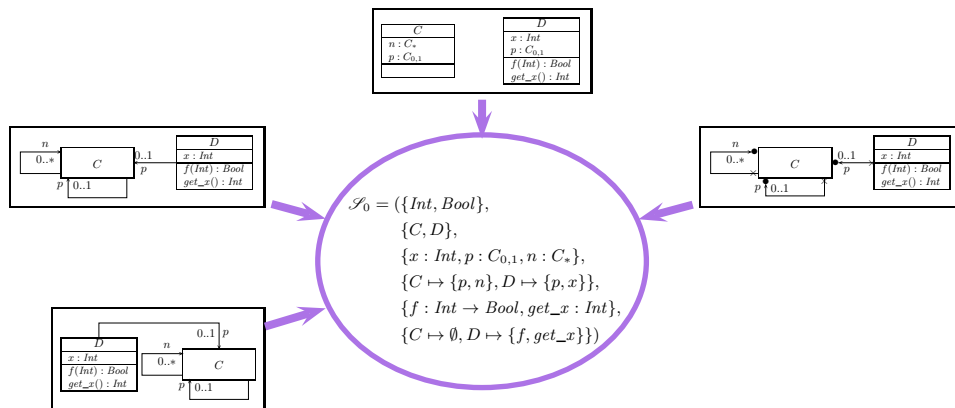
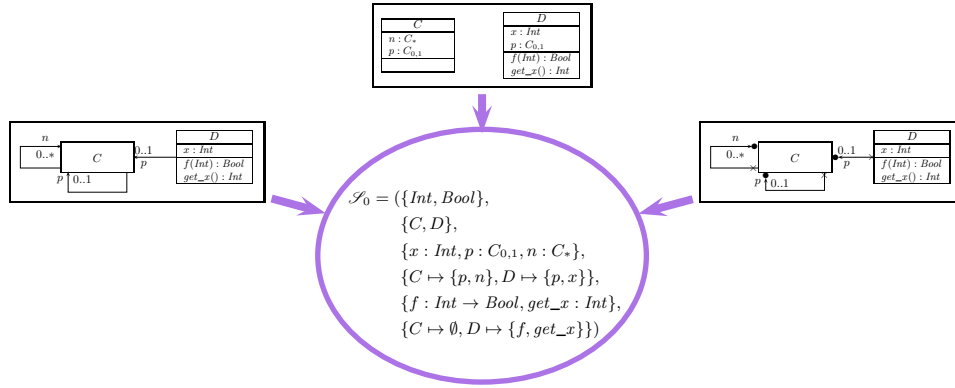
31/61

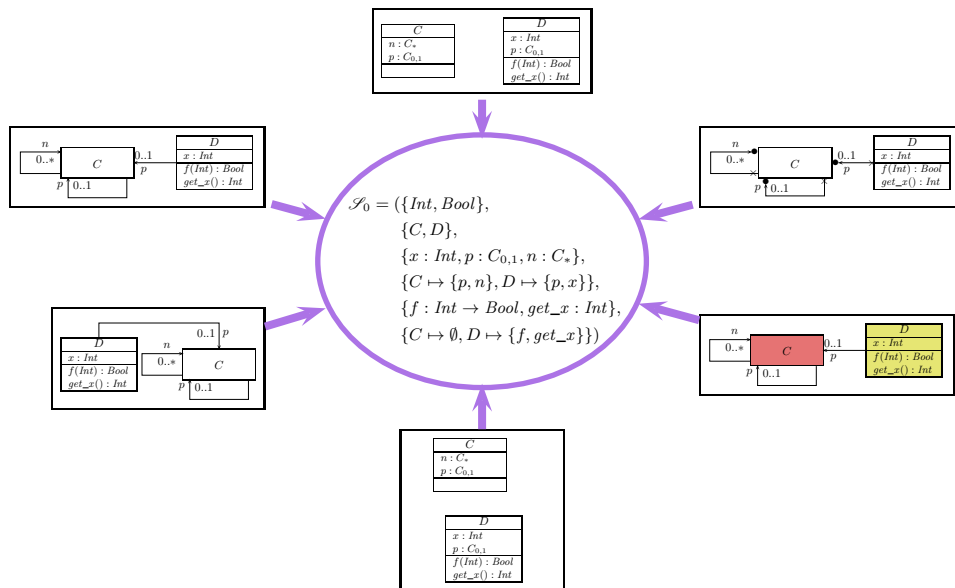
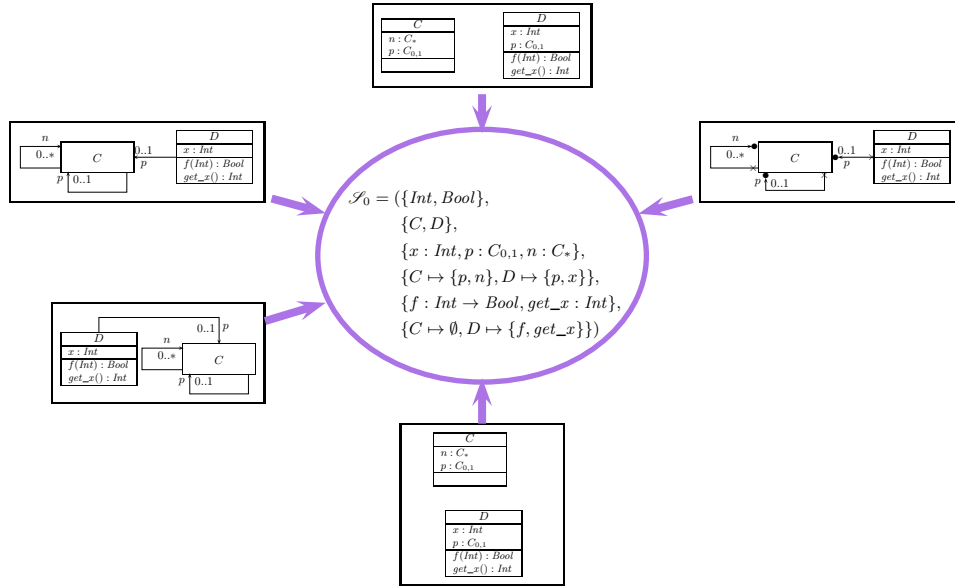
Once Again: Concrete vs. Abstract Syntax



–10–2019-04-17–Samstag–

32/61





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.

```
1 package pac;  
2  
3 import pac.D;  
4  
5 public class C {  
6  
7     public D n;  
8  
9     public void print_nx() {  
10         System.out.printf(  
11             "%i\n", n.get_x() );  
12     };  
13  
14     public C() {};  
15 }
```

```
1 package pac;  
2  
3 import pac.C;  
4  
5 public class D {  
6  
7     private int x;  
8  
9     public int get_x() {  
10         return x;  
11     };  
12  
13     public D() {};  
14 }
```

– 10 – 2019-04-17 – Software review –

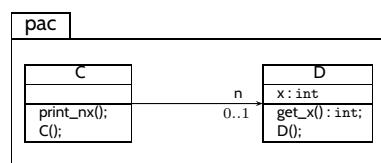
33/61

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.

```
1 package pac;  
2  
3 import pac.D;  
4  
5 public class C {  
6  
7     public D n;  
8  
9     public void print_nx() {  
10         System.out.printf(  
11             "%i\n", n.get_x() );  
12     };  
13  
14     public C() {};  
15 }
```

```
1 package pac;  
2  
3 import pac.C;  
4  
5 public class D {  
6  
7     private int x;  
8  
9     public int get_x() {  
10         return x;  
11     };  
12  
13     public D() {};  
14 }
```



– 10 – 2019-04-17 – Software review –

33/61

Visualisation of Implementation: (Useless) Example

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

Visualisation of Implementation: (Useless) Example

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

Visualisation of Implementation: (Useless) Example

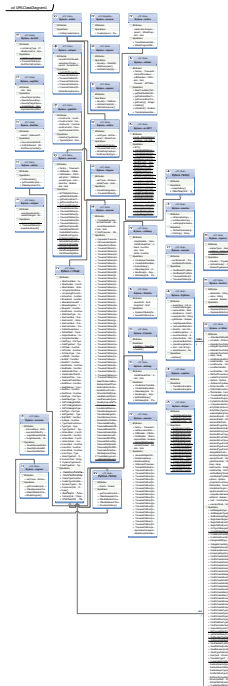
- open favourite IDE,
- open favourite **project**,
- press “**generate class diagram**”
- **wait...wait...wait...**

– 10 – 2018-Q4-17 – Software review –

34/61

Visualisation of Implementation: (Useless) Example

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

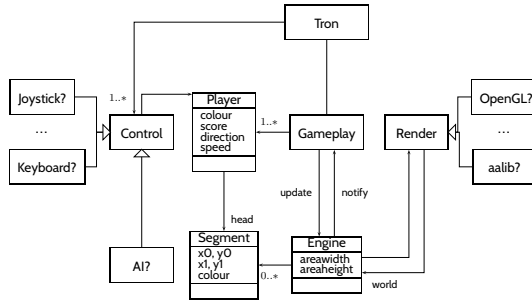
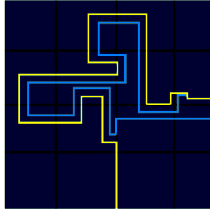


- ca. 35 classes,
- ca. 5,000 LOC C#

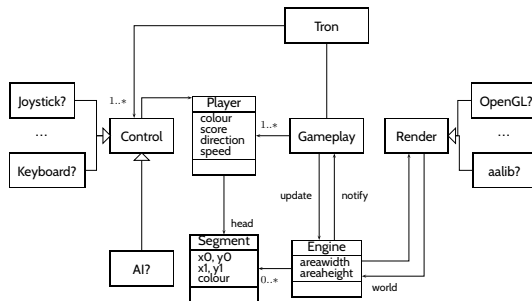
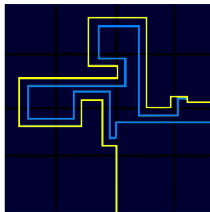
– 10 – 2018-Q4-17 – Software review –

34/61

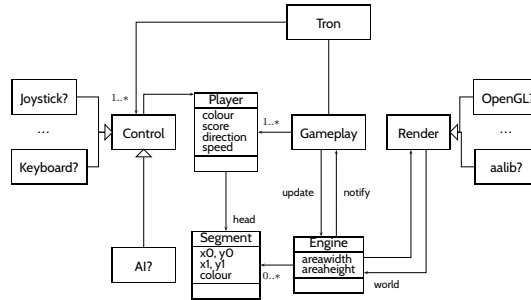
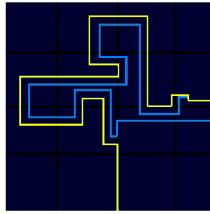
Visualisation of Implementation: (Useful) Example



Visualisation of Implementation: (Useful) Example



- A diagram is a **good diagram** if (and only if?) it serves its **purpose**!

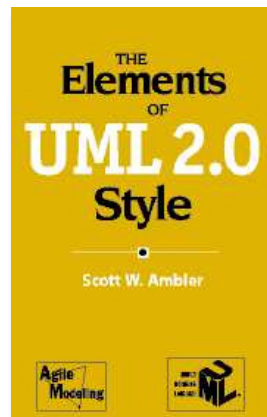


- A diagram is a **good diagram** if (and only if?) it serves its **purpose**!
- **Note**: a class **diagram** for visualisation may be partial.
 - show only the **most relevant** classes and attributes (for the given purpose).
- **Note**: a signature can be defined by a **set of** class diagrams.
 - use multiple class diagrams with a **manageable** number of classes for different purposes.

– 10 – 2019-06-17 – SoftwareReview –

35/61

Literature Recommendation



(Ambler, 2005)

– 10 – 2019-06-17 – SoftwareReview –

36/61

- **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.

A More Abstract Class Diagram Semantics

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

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

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

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

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

-10-2019-04-17 - Sumitrac -

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

39/61

Basic Object System Structure Example

Wanted: a structure for signature

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

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 $\mathcal{D}(C_{0,1}) = \mathcal{D}(C_*) = 2^{\mathcal{D}(C)}$.

$$\begin{aligned} \mathcal{D}(Int) &= \mathbb{Z} \\ \mathcal{D}(C) &= \mathbb{N} \times \{C\} = \{1_C, 2_C, 3_C, \dots\} \\ \mathcal{D}(D) &= \mathbb{N} \times \{D\} = \{1_D, 2_D, 3_D, \dots\} \\ \mathcal{D}(C_{0,1}) = \mathcal{D}(C_*) &= 2^{\mathcal{D}(C)} \\ \mathcal{D}(D_{0,1}) = \mathcal{D}(D_*) &= 2^{\mathcal{D}(D)} \end{aligned} \quad \left| \quad \begin{aligned} \mathcal{D}': \quad &\{3, 17, 25\} \\ &\{0, 1, 2, \dots\} \\ &\{a, aa, aaa, \dots\} \end{aligned}$$

-10-2019-04-17 - Sumitrac -

40/61

System State

Definition. Let \mathcal{D} be a structure of $\mathcal{S} = (\mathcal{T}, \mathcal{C}, V, \text{atr}, F, \text{mth})$.

A **system state** of \mathcal{S} 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 $u \in \mathcal{D}(C)$, $C \in \mathcal{C}$, if $u \in \text{dom}(\sigma)$

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

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

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

System State Examples

$$\begin{aligned} \mathcal{S}_0 &= (\{Int, Bool\}, \{C, D\}, \{x : Int, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \\ &\quad \{f : Int \rightarrow Bool, get_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get_x\}\}) \\ \mathcal{D}(Int) &= \mathbb{Z}, \quad \mathcal{D}(C) = \{1_C, 2_C, 3_C, \dots\}, \quad \mathcal{D}(D) = \{1_D, 2_D, 3_D, \dots\} \end{aligned}$$

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

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

$$\sigma_1 = \left\{ \underbrace{2_C \mapsto \{p \mapsto \{2_C\}, n \mapsto \emptyset\}}_{\mathcal{D}(C)}, \underbrace{1_D \mapsto \{p \mapsto \{2_C\}, x \mapsto 2_D\}}_{\text{link}} \right\}$$

$$\sigma_2 = \emptyset$$

$$\sigma_3 = \left\{ \underbrace{5_C \mapsto \{p \mapsto \{3_C\}, n \mapsto \emptyset\}}_{\text{link}} \right\} \checkmark$$

Class Diagrams at Work

- 10 - 2019-04-17 - main -

43/61

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.

```
1 package pac;  
2  
3 import pac.D;  
4  
5 public class C {  
6  
7     public D n;  
8  
9     public void print_nx() {  
10         System.out.printf(  
11             "%i\n", n.get_x() );  
12     };  
13     public C() {};  
14 }
```

```
1 package pac;  
2  
3 import pac.C;  
4  
5 public class D {  
6  
7     private int x;  
8  
9     public int get_x() {  
10         return x; }  
11  
12     public D() {};  
13 }
```

- 10 - 2019-04-17 - Subwork -

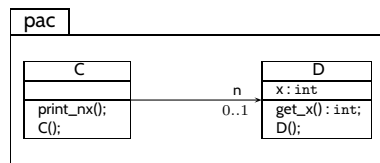
44/61

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.

```
1 package pac;  
2  
3 import pac.D;  
4  
5 public class C {  
6  
7     public D n;  
8  
9     public void print_nx() {  
10         System.out.printf(  
11             "%i\n", n.get_x() );  
12     };  
13     public C() {};  
14 }
```

```
1 package pac;  
2  
3 import pac.C;  
4  
5 public class D {  
6  
7     private int x;  
8  
9     public int get_x()  
10     { return x; };  
11  
12     public D() {};  
13 }
```



-10 - 2019-04-17 - ScalaWeek -

44/61

Visualisation of Implementation: (Useless) Example

- open favourite IDE,
- open favourite **project**,
- press **"generate class diagram"**
- wait...**

-10 - 2019-04-17 - ScalaWeek -

45/61

Visualisation of Implementation: (Useless) Example

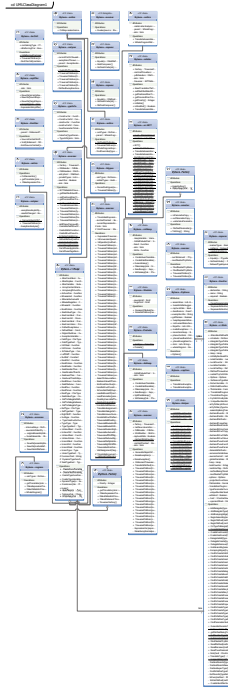
- open favourite IDE,
- open favourite **project**,
- press “**generate class diagram**”
- **wait...wait...**

Visualisation of Implementation: (Useless) Example

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

Visualisation of Implementation: (Useless) Example

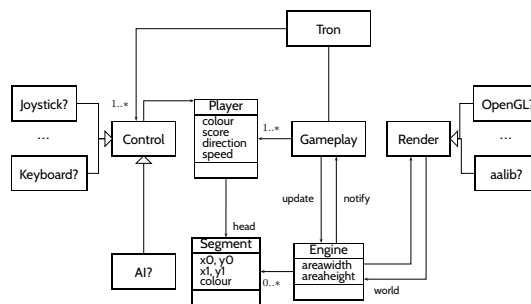
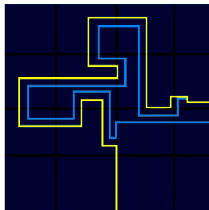
- open favourite IDE,
- open favourite **project**,
- press “**generate class diagram**”
- **wait...wait...wait...**



- ca. 35 classes,
- ca. 5,000 LOC C#

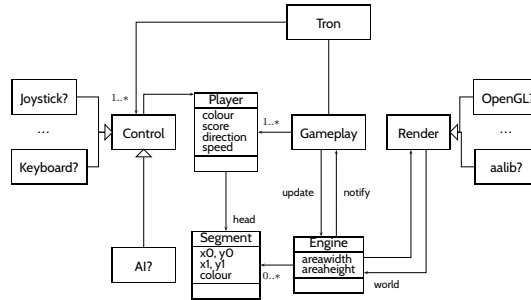
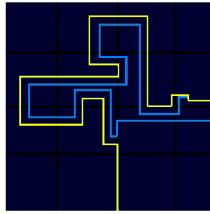
45/61

Visualisation of Implementation: (Useful) Example



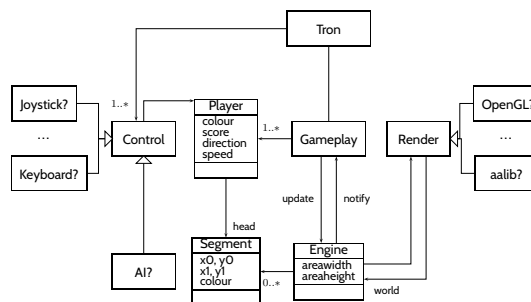
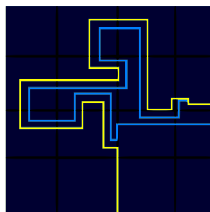
46/61

Visualisation of Implementation: (Useful) Example

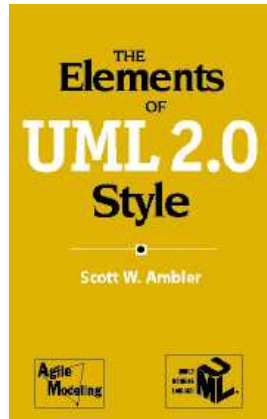


- A diagram is **a good diagram** if (and only if?) it serves its **purpose!**

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.



(Ambler, 2005)

– 10 – 2018-Q4-17 – Scitwork –

47/61

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.



– 10 – 2018-Q4-17 – Scitwork –

48/61

Object Diagrams

-10-2019-04-17-main-

49/61

Object Diagrams

$$\begin{aligned} \mathcal{S}_0 &= (\{Int, Bool\}, \{C, D\}, \{x : Int, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \\ &\quad \{f : Int \rightarrow Bool, get_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get_x\}\}), \quad \mathcal{D}(Int) = \mathbb{Z} \\ \sigma &= \{1_C \mapsto \underbrace{\{p \mapsto \emptyset, n \mapsto \{5_C\}\}}, 5_C \mapsto \underbrace{\{p \mapsto \emptyset, n \mapsto \emptyset\}}, 1_D \mapsto \underbrace{\{p \mapsto \{5_C\}, x \mapsto 23\}}\} \end{aligned}$$

-10-2019-04-17-5ed-

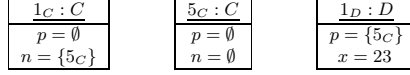
50/61

Object Diagrams

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

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

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



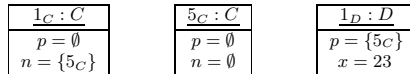
This is an **object diagram**.

Object Diagrams

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

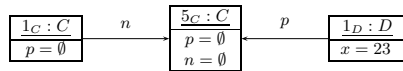
$$\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$$

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



This is an **object diagram**.

- Alternative notation:

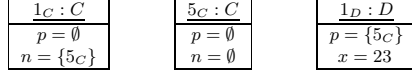


Object Diagrams

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

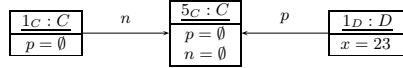
$$\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$$

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

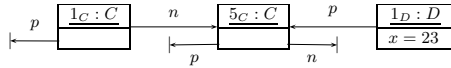


This is an **object diagram**.

- Alternative notation:



- Alternative **non-standard** notation:



-10-2019-04-17-5ed-

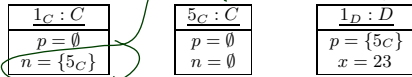
50/61

Object Diagrams

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

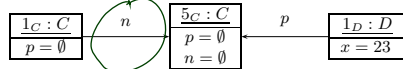
$$\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$$

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

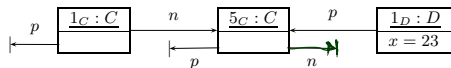


This is an **object diagram**.

- Alternative notation:

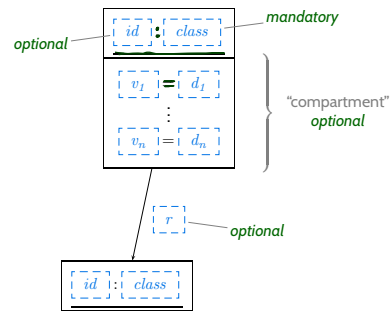


- Alternative **non-standard** notation:



-10-2019-04-17-5ed-

Concrete Syntax:



50/61

Special Case: Dangling Reference

Definition.

Let $\sigma \in \Sigma_{\mathcal{G}}$ be a system state and $u \in \text{dom}(\sigma)$ an alive object of class C in σ .

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.

$$\sigma(u)(r) \notin \text{dom}(\sigma).$$

Example:

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

–10–2019-04-17–5ed–

51/61

Special Case: Dangling Reference

Definition.

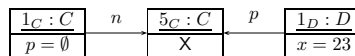
Let $\sigma \in \Sigma_{\mathcal{G}}$ be a system state and $u \in \text{dom}(\sigma)$ an alive object of class C in σ .

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.

$$\sigma(u)(r) \notin \text{dom}(\sigma).$$

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:

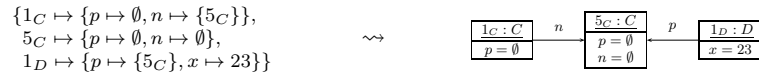


–10–2019-04-17–5ed–

51/61

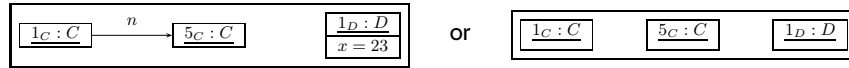
Partial vs. Complete Object Diagrams

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



What about the other way round...?

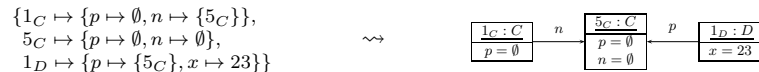
- Object diagrams can be **partial**, e.g.



→ we may omit information.

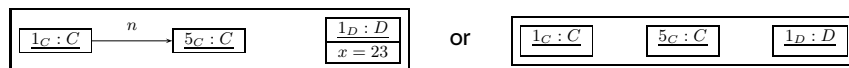
Partial vs. Complete Object Diagrams

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



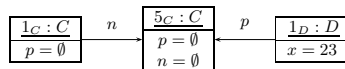
What about the other way round...?

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



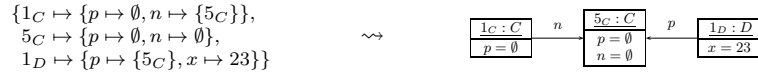
→ we may omit information.

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



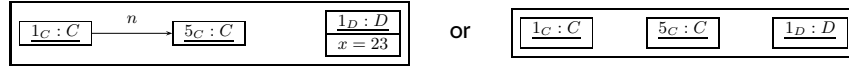
Partial vs. Complete Object Diagrams

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



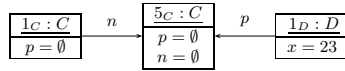
What about the other way round...?

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



→ we may omit information.

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



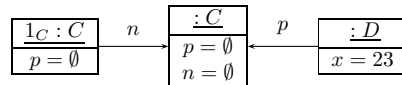
- If an object diagram
 - has values for **all** attributes of **all** objects in the diagram, and
 - if we **say that** it is meant to be complete

then we can **uniquely** reconstruct a system state σ .

52/61

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

Intuition: different boxes represent different objects.

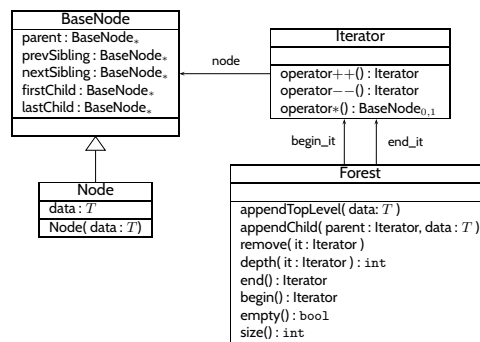
53/61

Object Diagrams at Work

-10-2019-04-17-main-

54/61

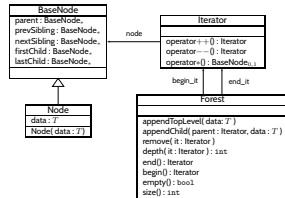
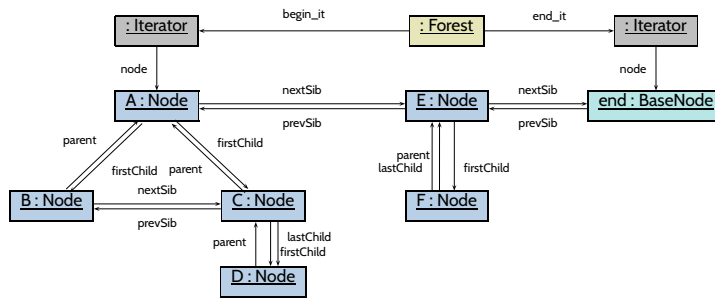
Example: Data Structure (Schumann et al., 2008)



-10-2019-04-17-Schumann-

55/61

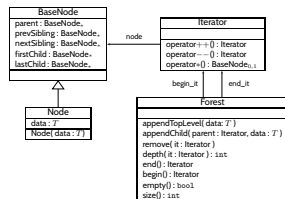
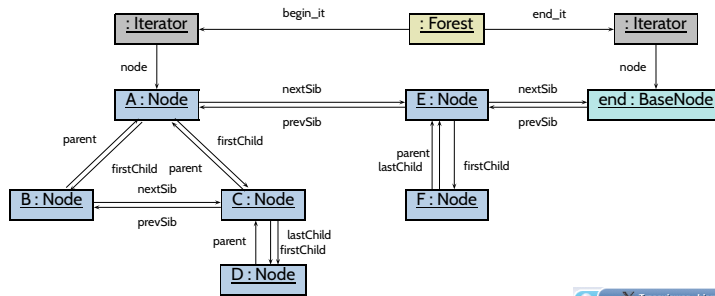
Example: Illustrative Object Diagram (Schumann et al., 2008)



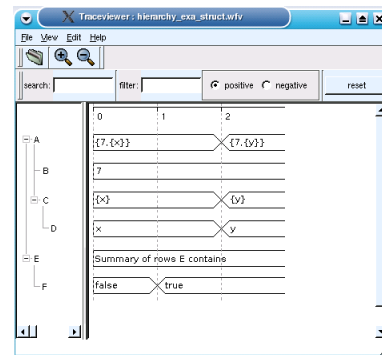
– 10 – 2019-04-17 – Solution –

56/61

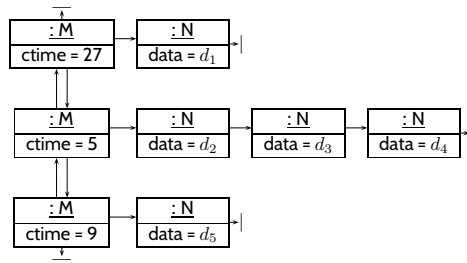
Example: Illustrative Object Diagram (Schumann et al., 2008)



– 10 – 2019-04-17 – Solution –



56/61



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** $\Sigma_{\mathcal{S}}$;
 - a **system state** is **structured** according to \mathcal{S} .
- A **System State** $\sigma \in \Sigma_{\mathcal{S}}$
 - can be **visualised** by an **object diagram**.

59/61

References

60/61

References

- Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press.
- Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002). Documenting software architecture: Documenting interfaces. Technical Report 2002-TN-015, CMU/SEI.
- Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley, 2nd edition.
- Broaddus, A. (2010). A tale of two eco-suburbs in Freiburg, Germany: Parking provision and car use. *Transportation Research Record*, 2187:114–122.
- Ellis, W. J., II, R. F. H., Saunders, T. F., Poon, P. T., Rayford, D., Sherlund, B., and Wade, R. L. (1996). Toward a recommended practice for architectural description. In *ICECCS*, pages 408–413. IEEE Computer Society.
- Glinz, M. (2008). Modellierung in der Lehre an Hochschulen: Thesen und Erfahrungen. *Informatik Spektrum*, 31(5):425–434.
- Harel, D. (1997). Some thoughts on statecharts, 13 years later. In Grumberg, O., editor, *CAV*, volume 1254 of *LNCS*, pages 226–231. Springer-Verlag.
- IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.
- IEEE (2000). *Recommended Practice for Architectural Description of Software-Intensive Systems*. Std 1471.
- Kruchten, P. (1995). The “4+1” view model of software architecture. *IEEE Software*, 12(6):42–50.
- 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.
- 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.
- Taylor, R. N., Medvidovic, N., and Dahofy, E. M. (2010). *Software Architecture Foundations, Theory, and Practice*. John Wiley and Sons.