Softwaretechnik / Software-Engineering

Lecture 10: Structural Software Modelling

2019-06-17

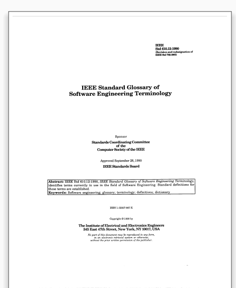
Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

Content

Vocabulary


IEEE Standard Glossary of
Software Engineering Terminology

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

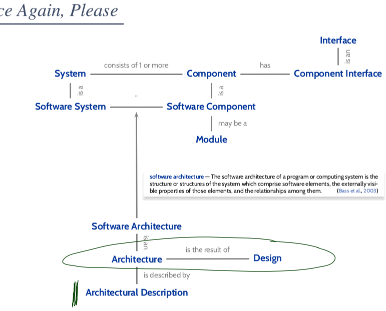
module— (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine.
(2) A logically separable part of a program. **IEEE 610.12 (1990)**

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 requirement. (Ludewig and Lichter, 2013)

Once Again, Please



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**.

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":

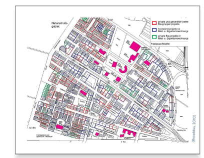
Goals and Relevance of Design: An Analogy

Design…

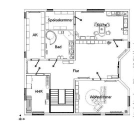
(i) **structures** a system into **manageable** units […],

(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
VL 11	• (simplified) Object Constraint Logic (OCL)

⋮	• **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

VL 13	• 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.

Modelling

Model

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

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.

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.

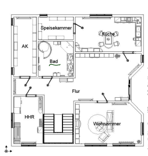
3. System

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



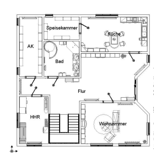
3. System

Example: Design-Models in Construction Engineering

1. Requirements

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

2. Designmodel



3. System

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

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

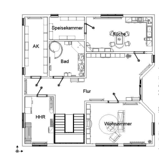
→ architects can efficiently work on appropriate level of abstraction

Example: Design-Models in Construction Engineering

1. Requirements

- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

2. Designmodel



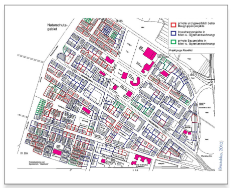
3. System

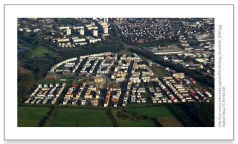
Observation (2): Floorplan **preserves/determines** certain system properties, e.g.,

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

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

A Better Analogy is Maybe Regional Planning

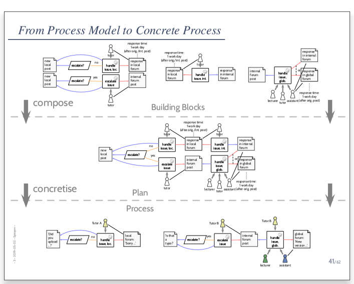




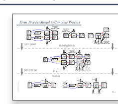
Software Modelling

Examples for (Software) Models?

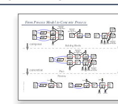
Examples for (Software) Models?

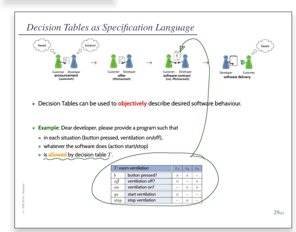


Examples for (Software) Models?

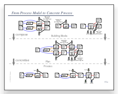


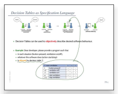
Examples for (Software) Models?



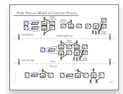


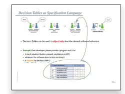
Examples for (Software) Models?

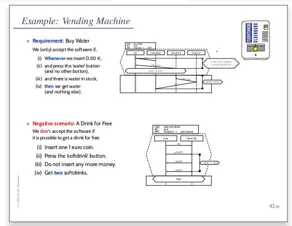




Examples for (Software) Models?







Views and Viewpoints

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)**

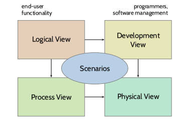
Views and Viewpoints

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

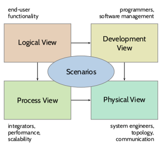
viewpoint — A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis. **IEEE 1471 (2000)**

- 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.

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



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



Newer proposals (Ludewig and Lichter, 2013):

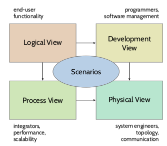
system view: How is the system under development integrated into (or seen by) its **environment**? With which other systems (including users) does it **interact** how?

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?

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



Newer proposals (Ludewig and Lichter, 2013):

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

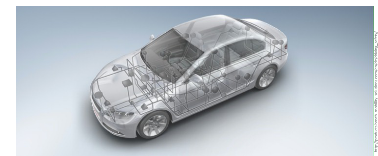
static view (∼ **developer view**): Components of the architecture, their interfaces and relations. Possibly: assignment of development, test, etc. onto teams.

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

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

("Purpose of architecture: **support** functionality; functionality is **not part** of the architecture." ?!)

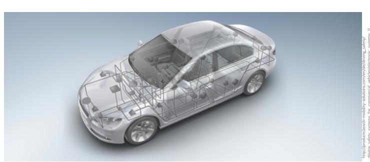
Deployment / Physical View



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.?

Deployment / Physical View



Example: modern cars

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

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

Structure vs. Behaviour / Constructive vs. Reflective

Structure vs. Behaviour / Constructive vs. Reflective

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

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

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

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.

Structure vs. Behaviour / Constructive vs. Reflective

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

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

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

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:

Structure vs. Behaviour / Constructive vs. Reflective

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

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

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

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."
  → **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**.

Structure vs. Behaviour / Constructive vs. Reflective

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

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

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

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."
  → **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**.

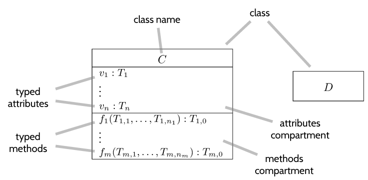
Note: No sharp boundaries! (would be too easy...)

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



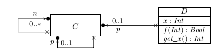
where

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

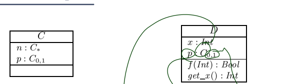
Concrete Syntax: Example

Concrete Syntax: Example

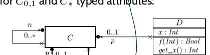
Alternative notation for $C_{0,1}$ and C_* typed attributes:



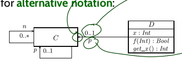
Concrete Syntax: Example



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

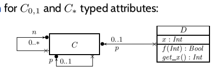


Alternative lazy notation for **alternative notation**:

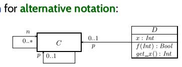


Concrete Syntax: Example

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



Alternative lazy notation for **alternative notation**:



And nothing else! This is **the** concrete syntax of **class diagrams** for the **scope of the course**.

Abstract Syntax: Object System Signature

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

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

where

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

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

Object System Signature Example

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

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

where

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

Object System Signature Example

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

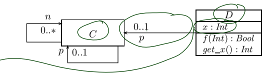
$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

where

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

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

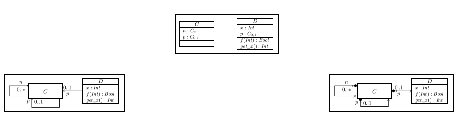
From Abstract to Concrete Syntax



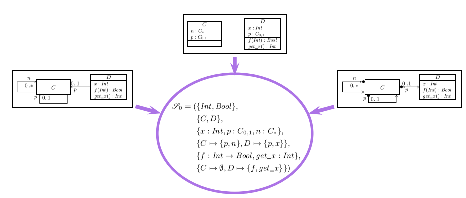
$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

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

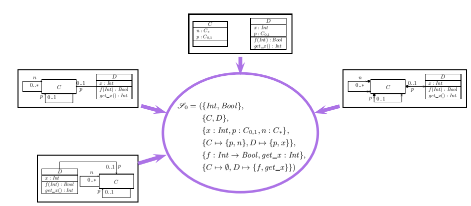
Once Again: Concrete vs. Abstract Syntax



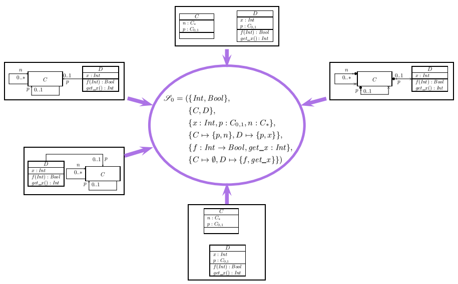
Once Again: Concrete vs. Abstract Syntax



Once Again: Concrete vs. Abstract Syntax

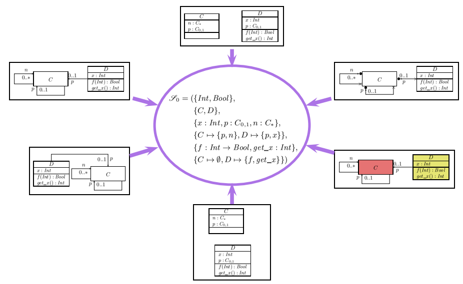


Once Again: Concrete vs. Abstract Syntax



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

Once Again: Concrete vs. Abstract Syntax



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

Visualisation of Implementation

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

```java
package pac;

import pac.D;

public class C {

  public D n;

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

  public C() {};
}
```

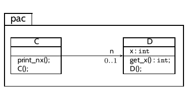
```java
package pac;

import pac.C;

public class D {

  private int x;

  public int get_x()
    { return x; };

  public D() {};
}
```


Visualisation of Implementation

- 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...

Visualisation of Implementation: (Useless) Example

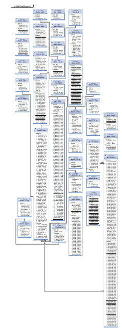
- open favourite IDE,
- open favourite **project**,
- press "**generate class diagram**"
- wait... wait...

Visualisation of Implementation: (Useless) Example

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

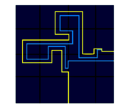
Visualisation of Implementation: (Useless) Example

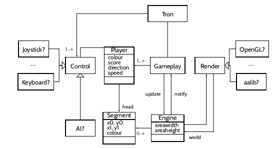
- open favourite IDE,
- open favourite **project**,
- press "**generate class diagram**"
- **wait**...**wait**...**wait**...



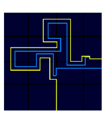
- ca. 35 classes,
- ca. 5,000 LOC C#

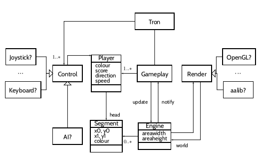
Visualisation of Implementation: (Useful) Example





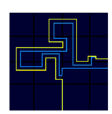
Visualisation of Implementation: (Useful) Example

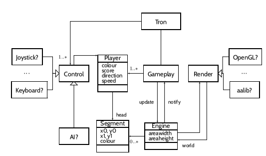




- A diagram is **a good diagram** if (and only if?) it serves its **purpose**!

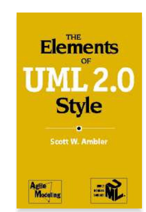
Visualisation of Implementation: (Useful) Example





- 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.

Literature Recommendation



(Ambler, 2005)

Content

A More Abstract Class Diagram Semantics

Object System Structure

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

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

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

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

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

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

Basic Object System Structure Example

Wanted: a structure for signature

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

A structure $\mathscr{D}$ maps

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

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

$\mathscr{D}'$: $\{3, 17, 25\}$

System State

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

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

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

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

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

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

System State Examples

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

$\mathscr{D}(Int) = \mathbb{Z}$, $\quad \mathscr{D}(C) = \{1_C, 2_C, 3_C, \dots\}$, $\quad \mathscr{D}(D) = \{1_D, 2_D, 3_D, \dots\}$

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

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

$\sigma_1 = \{ 2_C \mapsto \{p \mapsto \{1_C\}, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{2_C\}, x \mapsto 27\} \}$

$\sigma_2 = \emptyset$

$\sigma_3 = \{ 5_C \mapsto \{p \mapsto \{13_C\}, n \mapsto \emptyset\} \}$

Class Diagrams at Work

Visualisation of Implementation

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

```java
package pac;

import pac.D;

public class C {

  public D n;

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

  public C() {};
}
```

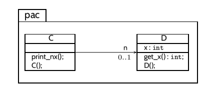
```java
package pac;

import pac.C;

public class D {

  private int x;

  public int get_x()
    { return x; };

  public D() {};
}
```


Visualisation of Implementation

- 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**…

Visualisation of Implementation: (Useless) Example

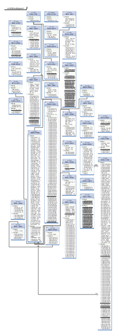
- open favourite IDE,
- open favourite **project**,
- press "**generate class diagram**"
- **wait**…**wait**…

Visualisation of Implementation: (Useless) Example

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

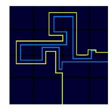
Visualisation of Implementation: (Useless) Example

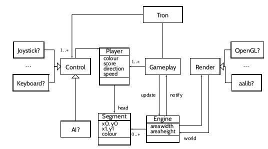
- open favourite IDE,
- open favourite **project**,
- press "**generate class diagram**"
- **wait**...**wait**...**wait**...



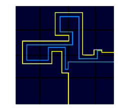
- ca. 35 classes,
- ca. 5,000 LOC C#

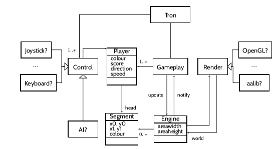
Visualisation of Implementation: (Useful) Example





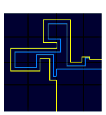
Visualisation of Implementation: (Useful) Example

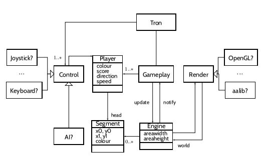




- A diagram is **a good diagram** if (and only if?) it serves its **purpose**!

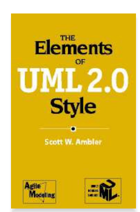
Visualisation of Implementation: (Useful) Example





- 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.

Literature Recommendation



(Ambler, 2005)

Content

Object Diagrams

Object Diagrams

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

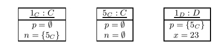
$$\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$$

Object Diagrams

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

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

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



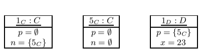
This is an **object diagram**.

Object Diagrams

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

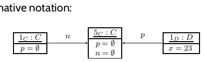
$$\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$$

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



This is an **object diagram**.

- Alternative notation:

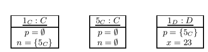


Object Diagrams

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

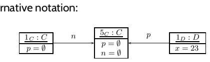
$$\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$$

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

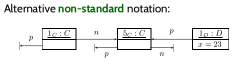


This is an **object diagram**.

- Alternative notation:



- Alternative **non-standard** notation:

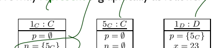


Object Diagrams

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

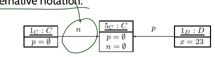
$$\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$$

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

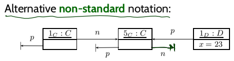


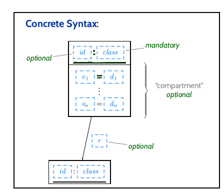
This is an **object diagram**.

- Alternative notation:



- Alternative **non-standard** notation:





Special Case: Dangling Reference

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

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

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

Example:

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

Special Case: Dangling Reference

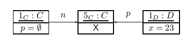
Definition.
Let $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$ be a system state and $u \in \mathrm{dom}(\sigma)$ an alive object of class C in σ.

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

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

Example:

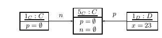
- $\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$
- Object diagram representation:



Partial vs. Complete Object Diagrams

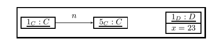
- By now we discussed "**object diagram represents system state**":

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

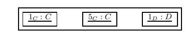


What about the other way round…?

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



or

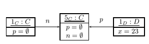


→ we may omit information.

Partial vs. Complete Object Diagrams

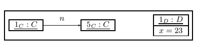
- By now we discussed "**object diagram represents system state**":

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

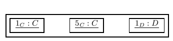


What about the other way round…?

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

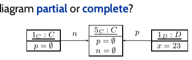


or



→ we may omit information.

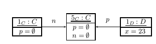
- Is the following object diagram **partial** or **complete**?



Partial vs. Complete Object Diagrams

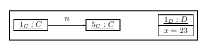
- By now we discussed "**object diagram represents system state**":

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

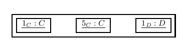


What about the other way round…?

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

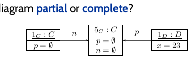


or



→ we may omit information.

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

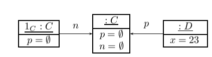


- If an object diagram
  - has values for **all** attributes of **all** objects in the diagram, and
  - if we **say that** it is meant to be complete

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

Special Case: Anonymous Objects

If the object diagram



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

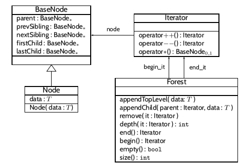
$$\{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{c\}\}, c \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, d \mapsto \{p \mapsto \{c\}, x \mapsto 23\}\}$$

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

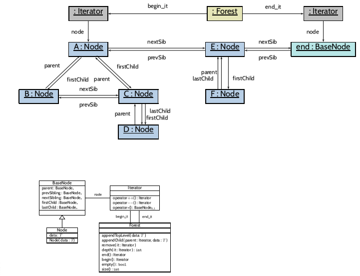
Intuition: different boxes represent different objects.

Object Diagrams at Work

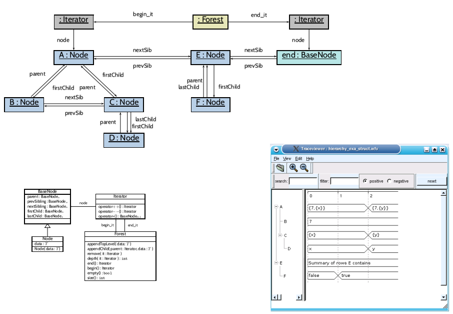
Example: Data Structure (Schumann et al., 2008)



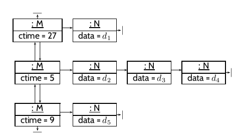
Example: Illustrative Object Diagram (Schumann et al., 2008)



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



Object Diagrams for Analysis



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.

Tell Them What You've Told Them...

- **Design** structures a system into **manageable units**.
- (Software) **Model**: a concrete or mental **image** or **archetype** with
  - **image** / **reduction** / **pragmatics** property.
- Towards **Software Modelling**:
  - Views and Viewpoints, e.g. 4+1,
  - **Structure** vs. **Behaviour**
- **Class Diagrams** can be used to **describe** system structures **graphically**
  - visualise code,
  - define an **object system structure** $\mathscr{S}$.
- An **Object System Structure** $\mathscr{S}$ (together with a structure $\mathscr{D}$)
  - defines a set of **system states** $\Sigma_{\mathscr{S}}^{\mathscr{D}}$;
  - a **system state** is **structured** according to $\mathscr{S}$.
- A **System State** $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$
  - can be **visualised** by an **object diagram**.

References

References

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

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.