Introduction and Vocabulary

Software Modelling
- model; views / viewpoints; 4+1 view

Modelling structure
- (simplified) Class & Object diagrams
- (simplified) Object Constraint Logic (OCL)

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

Design Patterns

Modelling behaviour
- Communicating Finite Automata (CFA)
- Uppaal query language

CFA vs. Software
- Unified Modelling Language (UML)
  - basic state-machines
  - an outlook on hierarchical state-machines

Model-driven/-based Software Engineering
Content

- Vocabulary
  - System, Architecture, Design

- Modelling

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

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

- Object Diagrams
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
Vocabulary
IEEE Standard Glossary of
Software Engineering Terminology

Sponsor
Standards Coordinating Committee
of the
Computer Society of the IEEE

Approved September 28, 1990
IEEE Standards Board


Keywords: Software engineering; glossary; terminology; definitions; dictionary

ISBN 0-7381-4073-8

Copyright © 1990 by
The Institute of Electrical and Electronics Engineers
345 East 47th Street, New York, NY 10017, USA

No part of this document may be reproduced in any form,
in an electronic retrieval system or otherwise,
without the prior written permission of the publisher.

Authorized licensed use limited to: UNIVERSITAET FREIBURG. Downloaded on April 03, 2015 at 13:47:32 UTC from IEEE Xplore. Restrictions apply.
**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)
**system**—A collection of components organized to accomplish a specific function or set of functions.

*IEEE 1471 (2000)*

**software system**—
A set of software units and their relations, if they together serve a common purpose. This purpose is in general complex, it usually includes, next to providing one (or more) executable program(s), also the organisation, usage, maintenance, and further development.

*(Ludewig and Lichter, 2013)*
**system**— A collection of components organized to accomplish a specific function or set of functions.

*IEEE 1471 (2000)*

---

**software system**—
A set of software units and their relations, if they together serve a common purpose. This purpose is in general complex, it usually includes, next to providing one (or more) executable program(s), also the organisation, usage, maintenance, and further development.

*(Ludewig and Lichter, 2013)*

---

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

*IEEE 610.12 (1990)*

---

**software component**— An architectural entity that
(1) encapsulates a subset of the system's functionality and/or data,
(2) restricts access to that subset via an explicitly defined interface, and
(3) has explicitly defined dependencies on its required execution context.

*(Taylor et al., 2010)*
module— (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine.

(2) A logically separable part of a program.

IEEE 610.12 (1990)

module— A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.

(Ludewig and Lichter, 2013)
**module**— (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine.

(2) A logically separable part of a program.

IEEE 610.12 (1990)

**module**— A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.

(Ludewig and Lichter, 2013)

**interface**— A boundary across which two independent entities meet and interact or communicate with each other.

(Bachmann et al., 2002)

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

(Ludewig and Lichter, 2013)
Once Again, Please

**Software System**

- consists of 1 or more
- is a

**Component**

- has
- is a

**Module**

- may be a

**System**

**Interface**

**Software Component**

**Component Interface**

---

**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)

---

**Architecture**

- is the result of
- is described by

**Design**

**Architectural Description**
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**.
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.
• **Introduction and Vocabulary**

• **Software Modelling**
  - model; views / viewpoints; 4+1 view

• **Modelling structure**
  - (simplified) Class & Object diagrams
  - (simplified) Object Constraint Logic (OCL)

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

• **Design Patterns**

• **Modelling behaviour**
  - Communicating Finite Automata (CFA)
  - Uppaal query language

• **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
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

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

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

2. Designmodel

3. System
1. Requirements

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

2. Designmodel

3. System

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

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

→ architects can efficiently work on appropriate level of abstraction
1. Requirements

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

2. Design model

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?
From Process Model to Concrete Process

**Compose**

**Building Blocks**

**Concretise**

**Plan Process**

---

**Examples for (Software) Models?**
Examples for (Software) Models?

From Process Model to Concrete Process

Plan Process

Building Blocks

compose

concretise

Did you upload...?

escalate?

handle issue, loc.

no tutor response in local forum

response time: 1 work day (after orig./int. post)

handle issue, int.

tutor response in internal forum

escalate issue

yes tutor internal forum post

handle issue, glob.

lecturer assistant tutor response in global forum or response time: 1 work day (after orig. post)

newlocalpost escalate?

escalate?

handle issue, loc.

no tutor response in local forum

response time: 1 work day (after orig./int. post)

handle issue, int.

tutor response in internal forum

escalate issue

yes tutor internal forum post

handle issue, glob.

lecturer assistant tutor response in global forum or response time: 1 work day (after orig. post)
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 $T$.

$$
T: \text{room ventilation} \quad | \quad p_1 \quad | \quad p_2 \quad | \quad p_3 \\
\hline
h \quad \text{button pressed?} & \times & \times & - \\
off \quad \text{ventilation off?} & \times & - & + \\
on \quad \text{ventilation on?} & - & \times & + \\
go \quad \text{start ventilation} & \times & - & - \\
stop \quad \text{stop ventilation} & - & \times & - \\
$$

29/63
Examples for (Software) Models?

From Process Model to Concrete Process

Decision Tables as Specification Language

- Decision Tables can be used to objectively describe desired software behavior.
- Example: User enters data, then:
  - in each situation: function performed, validation required,
  - unless the software data longer matches?

Plan

Building Blocks

compose

concretise

Process

newlocalpost escalate?

handle issue, loc.

no tutor response in local forum

response time: 1 work day (after orig./int. post)

handle issue, int.
tutor response in internal forum

handle issue, glob.
tutor response in global forum or

response time: 1 work day (after orig. post)
Example: Vending Machine

- **Requirement:** Buy Water
  We (only) accept the software if,
  (i) **Whenever** we insert 0.50 €,
  (ii) and press the 'water' button (and no other button),
  (iii) and there is water in stock,
  (iv) **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.
  (i) Insert one 1 euro coin.
  (ii) Press the 'softdrink' button.
  (iii) Do not insert any more money.
  (iv) Get **two** softdrinks.
**Views and Viewpoints**

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

**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.

**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.

---

- A **perspective** is determined by **concerns** and **information needs**:
  - **team leader**, e.g., needs to know which team is working on what component,
  - **operator**, e.g., needs to know which component is running on which host,
  - **developer**, e.g., needs to know interfaces of other components.
  - etc.
An Early Proposal: The 4+1 View (Kruchten, 1995)

- **Logical View**
  - end-user functionality

- **Development View**
  - programmers, software management

- **Process View**
  - integrators, performance, scalability

- **Physical View**
  - system engineers, topology, communication

Scenarios connect these views, illustrating the flow of information and collaboration among different stakeholders.
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?
Newer proposals (Ludewig and Lichter, 2013):

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

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

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

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

(“Purpose of architecture: support functionality; functionality is not part of the architecture.” ?!)
Example: modern cars

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

---

- **Form of the states** in $\Sigma$ (and actions in $A$): **structure** of $S$
- **Computation paths** $\pi$ of $S$: **behaviour** of $S$
Structure vs. Behaviour / Constructive vs. Reflective

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

- **Computation paths** $\pi$ of $S$: behaviour of $S$

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

**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 $\Sigma$ (and actions in $A$):
  - structure of $S$

- **Computation paths** $\pi$ 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."
  - $\rightarrow$ **what should (or should not) be computed.**

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

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

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

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

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

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

---

**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 \( S \).
Content

- **Vocabulary**
  - System, Architecture, Design

- **Modelling**

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

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

- **Object Diagrams**
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
Class Diagrams
Class Diagrams: Concrete Syntax

where

- $T_1, \ldots, T_m, 0 \in \mathcal{T} \cup \{C_{0,1}, C^* \mid C \text{ a class name}\}$

- $\mathcal{T}$ is a set of basic types, e.g. $\text{Int}$, $\text{Bool}$, \ldots.
Concrete Syntax: Example

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>n : C*</td>
<td>x : Int</td>
</tr>
<tr>
<td>p : C₀,₁</td>
<td>p : C₀,₁</td>
</tr>
<tr>
<td></td>
<td>f(Int) : Bool</td>
</tr>
<tr>
<td></td>
<td>get_x() : Int</td>
</tr>
</tbody>
</table>
Concrete Syntax: Example

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

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

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

Alternative notation for $C_{0,1}$ and $C_\ast$ 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.
Definition. An **(Object System) Signature** is a 6-tuple

\[ S = (T, C, V, \text{atr}, F, mth) \]

where

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

**Note:** Inspired by OCL 2.0 standard *OMG (2006)*, Annex A.
Definition. An **(Object System) Signature** is a 6-tuple

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

where

- \( \mathcal{T} \) is a set of (basic) **types**, 
- \( \mathcal{C} \) is a finite set of **classes**, 
- \( V \) is a finite set of **typed attributes** \( v : T \), i.e., each \( v \in V \) has type \( T \), 
- \( \text{atr} : \mathcal{C} \rightarrow 2^V \) maps each class to its set of attributes. 
- \( F \) is a finite set of **typed behavioural features** \( f : T_1, \ldots, T_n \rightarrow T \), 
- \( \text{mth} : \mathcal{C} \rightarrow 2^F \) maps each class to its set of behavioural features. 
- A type can be a basic type \( \tau \in \mathcal{T} \), or \( C_{0,1} \), or \( C^* \), where \( C \in \mathcal{C} \).
Definition. An **Object System Signature** is a 6-tuple

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

where

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

**Example**

\[ \mathcal{S}_0 = (\{\text{Int}, \text{Bool}\}, \)
\[ \{\mathcal{C}, D\}, \]
\[ \{x : \text{Int}, p : \mathcal{C}^{0,1}, n : \mathcal{C}^*\}, \]
\[ \text{atr} : \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \]
\[ \text{mth} : \{f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int}\}, \]
\[ \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\} ) \]
From Abstract to Concrete Syntax

\[ \mathcal{I} = (\mathcal{F}, \mathcal{C}, V, \text{atr}, F, mth) \]

- \( \mathcal{F} = \{ \text{Int}, \text{Bool} \} \)
- \( \mathcal{C} = \{ C, D \} \)
- \( V = \{ x : \text{Int}, p : C_{0..1}, n : C^{*} \} \)
- \( \text{atr} = \{ C \mapsto \{ n, p \}, D \mapsto \{ x, p \} \} \)
- \( F = \{ f : \text{Int} \to \text{Bool}, \ldots \} \)
- \( mth = \{ C \mapsto \emptyset, \ldots \} \)
Once Again: Concrete vs. Abstract Syntax

\[
\begin{align*}
C & : C_\ast \\
p & : C_{0,1}
\end{align*}
\]

\[
\begin{align*}
D & : x : \text{Int} \\
p & : C_{0,1} \\
f(\text{Int}) & : \text{Bool} \\
get_x() & : \text{Int}
\end{align*}
\]

\[
\begin{align*}
\text{get}_x() &: \text{Int}
\end{align*}
\]
\( \mathcal{A}_0 = \langle \{\text{Int, Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\} \rangle \)
Once Again: Concrete vs. Abstract Syntax

\[ \mathcal{A}_0 = (\{\text{Int, Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_{\ast}\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x() : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}) \]
Once Again: Concrete vs. Abstract Syntax

\[ \mathcal{A}_0 = (\{\text{Int, Bool}\}, \{n, \text{C}\}, \{x : \text{Int}, p : \text{C}_0, n : \text{C}_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}) \]
Once Again: Concrete vs. Abstract Syntax

$\mathcal{H}_0 = (\{\text{Int}, \text{Bool}\},$
$\{C, D\},$
$\{x : \text{Int}, p : C_{0,1}, n : C_{\ast}\},$
$\{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},$
$\{f : \text{Int} \to \text{Bool, get}_x : \text{Int}\},$
$\{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\})$
The class diagram syntax can be used to **visualise code**: Provide rules which map (parts of) the code to class diagram elements.

```java
package pac;

import pac.D;

public class C {
  public D n;

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

  public C() {};
}
```

```java
package pac;

import pac.C;

public class D {
  private int x;

  public int get_x() {
    return x;
  }

  public D() {};
}
```
• The class diagram syntax can be used to **visualise code**:
  Provide rules which map (parts of) the code to class diagram elements.

```
package pac;
import pac.D;

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

    public C() {};
}
```

```
package pac;
import pac.C;

public class D {
    private int x;
    public int get_x() {
        return x;
    }

    public D() {};
}
```

- Visualisation of Implementation
• 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
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

- 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 \( 2^{\mathcal{D}(C)} \).

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

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

**Wanted:** a structure for signature

\[ \mathcal{S}_0 = (\{\text{Int, Bool}\}, \{C, D\}, \{x : \text{Int, p : C}_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\} ) \]

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

\[
\begin{align*}
\mathcal{D}(\text{Int}) &= \mathbb{Z} \\
\mathcal{D}(C) &= \mathbb{N} \times \{C\} = \{c_1, c_2, c_3, \ldots\} \quad \text{for } C \in \mathcal{C} \\
\mathcal{D}(D) &= \mathbb{N} \times \{D\} = \{d_1, d_2, d_3, \ldots\} \quad \text{for } D \in \mathcal{D} \\
\mathcal{D}(C_{0,1}) = \mathcal{D}(C_*) &= 2^{\mathcal{D}(C)} \\
\mathcal{D}(D_{0,1}) = \mathcal{D}(D_*) &= 2^{\mathcal{D}(D)}
\end{align*}
\]

\( \mathcal{D}' \) is a mapping with:

- \( \mathcal{D}' : \{3, 17, 25\} \)
- \( \mathcal{D}' : \mathcal{D} \) with values:
  - \( c_1, a, b, \ldots \) for \( C \)
  - \( d_1, a, a, \ldots \) for \( D \).
**Definition.** Let $\mathcal{D}$ be a structure of $\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, atr, F, mth)$. A system state of $\mathcal{I}$ wrt. $\mathcal{D}$ is a type-consistent mapping $
abla : \mathcal{D}(\mathcal{C}) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{T}) \cup \mathcal{D}(\mathcal{C}^*))).

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

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

We call $u \in \mathcal{D}(\mathcal{C})$ alive in $\nabla$ if and only if $u \in \text{dom}(\nabla)$. We use $\Sigma_{\mathcal{D}}$ to denote the set of all system states of $\mathcal{I}$ wrt. $\mathcal{D}$. 
A system state is a partial function $\sigma : \mathcal{D}(C) \nrightarrow (V \nrightarrow (\mathcal{D}(T) \cup \mathcal{D}(C_*)))$ 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 $v : D_\ast$ or $v : D_{0,1}$ with $D \in \mathcal{C}$.

\[ \mathcal{S}_0 = (\{\text{Int}, \text{Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_\ast\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}) \]

\[ \mathcal{D}(\text{Int}) = \mathbb{Z}, \quad \mathcal{D}(C) = \{1_C, 2_C, 3_C, \ldots\}, \quad \mathcal{D}(D) = \{1_D, 2_D, 3_D, \ldots\} \]
Class Diagrams at Work
The class diagram syntax can be used to **visualise code**:

Provide rules which map (parts of) the code to class diagram elements.
• 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() {
    }
}

package pac;

import pac.C;

public class D {
    private int x;
    public int get_x() {
        return x;
    }
    public D() {
    }
}
```

```
C

| print_nx(); | n | D
|-------------|---|---
| 0..1        |   | x:int
| C();        |   | get_x():int;
|             |   | 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**...
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
A diagram is a good diagram if (and only if?) it serves its purpose!
A diagram is a good diagram if (and only if?) it serves its purpose!

Note: a class diagram for visualisation may be partial.
→ show only the most relevant classes and attributes (for the given purpose).

Note: a signature can be defined by a set of class diagrams.
→ use multiple class diagrams with a manageable number of classes for different purposes.
Literature Recommendation

(Ambler, 2005)
Content

- Vocabulary
  - System, Architecture, Design

- Modelling

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

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

- Object Diagrams
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
Object Diagrams
\[ \mathcal{S}_0 = (\{\text{Int}, \text{Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \to \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}), \quad \mathcal{D}(\text{Int}) = \mathbb{Z} \]

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

\[ \mathcal{S}_0 = (\{\text{Int, Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}), \mathcal{D}(\text{Int}) = \mathbb{Z} \]

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

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

<table>
<thead>
<tr>
<th>1_C : C</th>
<th>5_C : C</th>
<th>1_D : D</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = \emptyset )</td>
<td>( p = \emptyset )</td>
<td>( p = {5_C} )</td>
</tr>
<tr>
<td>( n = {5_C} )</td>
<td>( n = \emptyset )</td>
<td>( x = 23 )</td>
</tr>
</tbody>
</table>

This is an object diagram.
\[ \mathcal{S}_0 = (\{\text{Int, Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0, 1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}), \quad \mathcal{O} (\text{Int}) = \mathbb{Z} \]

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

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

```
<table>
<thead>
<tr>
<th>1_C : C</th>
<th>5_C : C</th>
<th>1_D : D</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = \emptyset</td>
<td>p = \emptyset</td>
<td>p = {5_C}</td>
</tr>
<tr>
<td>n = {5_C}</td>
<td>n = \emptyset</td>
<td>x = 23</td>
</tr>
</tbody>
</table>
```

This is an **object diagram**.

- Alternative notation:
Object Diagrams

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

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

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

\[ \begin{array}{c|c|c}
1_C : C & 5_C : C & 1_D : D \\
\hline
p = \emptyset & p = \emptyset & p = \{5_C\} \\
n = \{5_C\} & n = \emptyset & x = 23 \\
\end{array} \]

This is an object diagram.

- Alternative notation:

\[ \begin{array}{c|c|c}
1_C : C & 5_C : C & 1_D : D \\
\hline
p = \emptyset & p = \emptyset & p = \{5_C\} \\
n \mapsto 5_C & n \mapsto \emptyset & x = 23 \\
\end{array} \]

- Alternative non-standard notation:
\[ \mathcal{S}_0 = (\{\text{Int}, \text{Bool}\}, \{C, D\}, \{x : \text{Int}, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\}, \{f : \text{Int} \rightarrow \text{Bool}, \text{get}_x : \text{Int}\}, \{C \mapsto \emptyset, D \mapsto \{f, \text{get}_x\}\}), \mathcal{D}(\text{Int}) = \mathbb{Z} \]

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

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

This is an object diagram.

- Alternative notation:

- Alternative non-standard notation:
Special Case: Dangling Reference

Definition.
Let $\sigma \in \Sigma^D$ be a system state and $u \in \text{dom}(\sigma)$ an alive object of class $C$ in $\sigma$.

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

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

Example:
- $\sigma = \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, 1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}$
**Definition.**

Let \( \sigma \in \Sigma^\mathcal{D} \) be a system state and \( u \in \text{dom}(\sigma) \) an alive object of class \( C \) in \( \sigma \).

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

\[
\sigma(u)(r) \not\subset \text{dom}(\sigma).
\]

**Example:**

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

```
[1_C : C
  p = \emptyset]  n  [5_C : C
  X]  p  [1_D : D
  x = 23]
```
• By now we discussed “object diagram represents system state”:

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

What about the other way round…?

• Object diagrams can be partial, e.g.

→ we may omit information.
By now we discussed “object diagram represents system state”:

\[
\{ 1_C \mapsto \{ p \mapsto \emptyset, n \mapsto \{ 5_C \} \}, \\
5_C \mapsto \{ p \mapsto \emptyset, n \mapsto \emptyset \}, \\
1_D \mapsto \{ p \mapsto \{ 5_C \}, x \mapsto 23 \} \}
\]

What about the other way round…?

Object diagrams can be partial, e.g.

\[
1_C : C \\ p = \emptyset \\
5_C : C \\ p = \emptyset, n = \emptyset \\
1_D : D \\ x = 23
\]

→ we may omit information.

Is the following object diagram partial or complete?

\[
1_C : C \\ p = \emptyset \\
5_C : C \\ p = \emptyset, n = \emptyset \\
1_D : D \\ x = 23
\]
By now we discussed “object diagram represents system state”:

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

What about the other way round...?

Object diagrams can be partial, e.g.

\[
\begin{align*}
1_C &: C & p = \emptyset \\
5_C &: C & n = \emptyset
\end{align*}
\]

or

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

→ 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 \( \sigma \).
**Special Case: Anonymous Objects**

If the object diagram

\[
\begin{array}{c}
1_C : C \\
p = \emptyset
\end{array}
\xrightarrow{n}
\begin{array}{c}
\vdash C \\
p = \emptyset \\
n = \emptyset
\end{array}
\xrightarrow{p}
\begin{array}{c}
\vdash D \\
x = 23
\end{array}
\]

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

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

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

**Intuition:** different boxes represent different objects.
Object Diagrams at Work
BaseNode:
- parent : BaseNode
- prevSibling : BaseNode
- nextSibling : BaseNode
- firstChild : BaseNode
- lastChild : BaseNode

Node:
- data : T
- Node(data : T)

Iterator:
- operator++(): Iterator
- operator--(): Iterator
- operator*(): BaseNode

Forest:
- appendTopLevel(data : T)
- appendChild(parent : Iterator, data : T)
- remove(it : Iterator)
- depth(it : Iterator) : int
- end() : Iterator
- begin() : Iterator
- empty() : bool
- size() : int
Example: Illustrative Object Diagram (Schumann et al., 2008)

```
: Iterator

node

begin_it

: Forest

end_it

: Iterator

node

A : Node

parent

nextSibling

firstChild

parent

B : Node

prevSibling

nextSibling

firstChild

C : Node

parent

lastChild

firstChild

D : Node

prevSibling

lastChild

parent

E : Node

nextSibling

parent

F : Node

BaseNode

parent : BaseNode,
prevSibling : BaseNode,
nextSibling : BaseNode,
firstChild : BaseNode,
lastChild : BaseNode,
node

Iterator

operator ++() : Iterator
operator --() : Iterator
operator () : BaseNode

begin_it

end_it

Node

data : T

Node(data : T)

Forest

appendTopLevel(data : T)
appendChild(parent : Iterator, data : T)
remove(it : Iterator)
depth(it : Iterator) : int
end() : Iterator
begin() : Iterator
empty() : bool
size() : int
```
Example: Illustrative Object Diagram (Schumann et al., 2008)

- Iterator
  node

- Forest
  nextSib
  prevSib
  parent
  firstChild
  lastChild

- Node
  data
  parent
  prevSibling
  nextSibling
  firstChild
  prevSibling
  parent

- BaseNode
  parent
  prevSibling
  nextSibling
  firstChild
  lastChild

- Iterator
  operator ++(): Iterator
  operator --(): Iterator
  operator *(): BaseNode

- Forest
  appendTopLevel(data: T)
  appendChild(parent: Iterator, data: T)
  remove(it: Iterator)
  depth(it: Iterator): int
  end(): Iterator
  begin(): Iterator
  empty(): bool
  size(): int

Summary of rows E contains:
false true
Object Diagrams for Analysis

```plaintext
: M
ctime = 27
  ┌───────────┐
  │           │
  │           │
  └───────────┘
  : N
  data = d₁

: M
ctime = 5
  ┌───────────┐
  │           │
  │           │
  └───────────┘
  : N
data = d₂

: N
data = d₃

: N
data = d₄

: M
ctime = 9
  ┌───────────┐
  │           │
  │           │
  └───────────┘
  : N
data = d₅
```
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** $\mathcal{S}$.

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

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


