Softwaretechnik / Software-Engineering

Lecture 12: Proto-OCL, Modularisation & Design Patterns

2017-07-03

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

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

VL 10	• **Introduction and Vocabulary**
	• **Software Modelling I**
⋮	(i) views and viewpoints, the 4+1 view
	(ii) model-driven/-based software engineering
	(iii) **Modelling structure**
VL 11	a) (simplified) class diagrams
⋮	b) (simplified) object diagrams
VL 12	c) (simplified) object constraint logic (OCL)
	d) Unified Modelling Language (UML)
	• **Principles of Design**
⋮	(i) modularity, separation of concerns
	(ii) information hiding and data encapsulation
	(iii) abstract data types, object orientation
	(iv) **Design Patterns**
VL 13	• **Software Modelling II**
⋮	(i) **Modelling behaviour**
	a) communicating finite automata
	b) Uppaal query language
VL 14	c) basic state-machines
⋮	d) an outlook on hierarchical state-machines

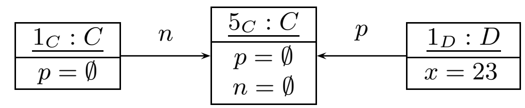
Content

- **Proto-OCL**
  - syntax, semantics,
  - Proto-OCL vs. OCL.
  - Proto-OCL vs. Software
- An outlook on **UML**
- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ~~…by example~~
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

Partial vs. Complete Object Diagrams

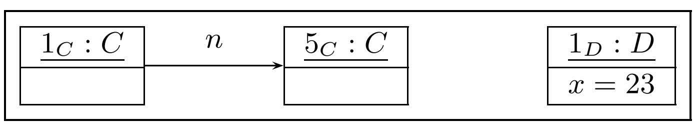
- By now we discussed "**object diagram represents system state**":

$$\begin{aligned}\{1_C &\mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\},\\ 5_C &\mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\},\\ 1_D &\mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\}\end{aligned} \quad \rightsquigarrow$$

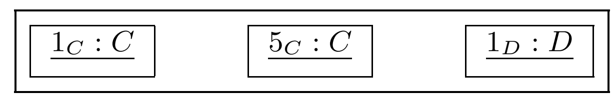


What about the other way round...?

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

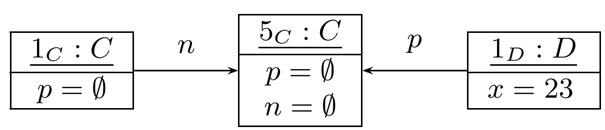


or



$\rightarrow$ we may omit information.

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

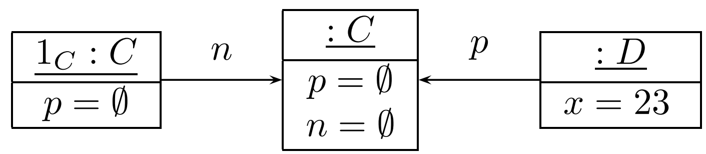


- If an object diagram
  - has values for **all** attributes of **all** objects in the diagram, and
  - if we **say that** it is meant to be complete

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

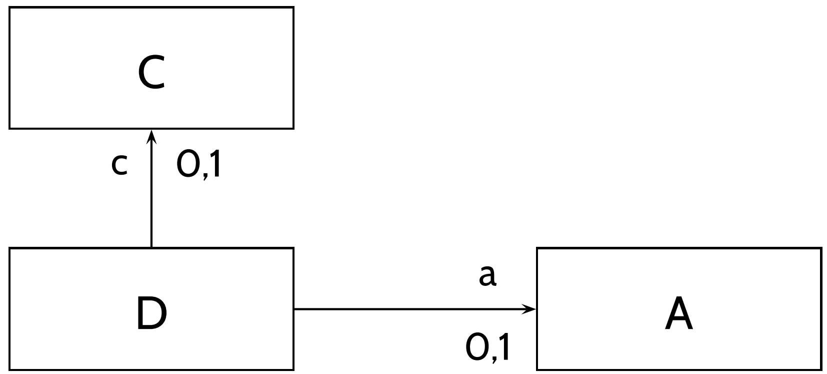
Intuition: different boxes represent different objects.

Content

- **Proto-OCL**
  - syntax, semantics,
  - Proto-OCL vs. OCL.
  - Proto-OCL vs. Software
- An outlook on **UML**
- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ...by example
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

Towards Object Constraint Logic (OCL)
— "Proto-OCL" —

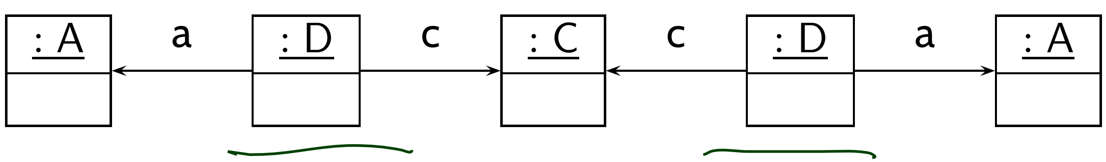
Motivation



- How do I **precisely, formally** tell **my developers** that

  All D-instances having a link to the same C object
  should have links to the same A.

- That is, the following system state is **forbidden** in the software:

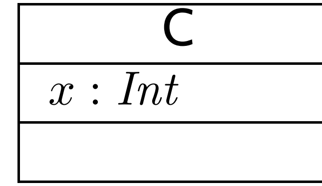


  Note: formally, it is a **proper system state**.

- Use **(Proto-)OCL**: "Dear developers, please only use system states which satisfy:"

$$\forall\, d_1 \in allInstances_C \bullet \forall\, d_2 \in allInstances_C \bullet c(d_1) = c(d_2) \implies a(d_1) = a(d_2)$$

Constraints on System States



- **Example**: for all C-instances, x should never have the value 27.

$$\forall\, c \in allInstances_C \bullet x(c) \neq 27$$

- **Proto-OCL Syntax** wrt. signature $(\mathscr{T}, \mathscr{C}, V, atr, F, mth)$, c is a **logical variable**, $C \in \mathscr{C}$:

$$
\begin{array}{rlll}
F ::= & c & : \tau_C & (*) \\
| & allInstances_C & : 2^{\tau_C} & \\
| & v(F) & : \tau_C \to \tau_\perp, & \text{if } v : \tau \in atr(C),\ \tau \in \mathscr{T} \quad (**) \\
| & v(F) & : \tau_C \to \tau_D, & \text{if } v : D_{0,1} \in atr(C) \\
| & v(F) & : \tau_C \to 2^{\tau_D}, & \text{if } v : D_* \in atr(C) \\
| & f(F_1, \dots, F_n) & : \tau_1 \times \dots \times \tau_n \to \tau, & \text{if } f : \tau_1 \times \dots \times \tau_n \to \tau \quad (***) \\
| & \forall\, c \in F_1 \bullet F_2 & : \tau_C \times 2^{\tau_C} \times \mathbb{B}_\perp \to \mathbb{B}_\perp &
\end{array}
$$

- The formula above in **prefix normal form**: $\forall\, c \in allInstances_C \bullet \underbrace{\underset{f}{\neq} (\underbrace{\overset{: Int}{x}(\underbrace{c}_{(*)})}_{(**)}, \underbrace{\overset{: Int}{27}}_{(***)})}_{(****)}$ $: \to Int$

Semantics

$\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow \mathscr{D})$

- **Proto-OCL Types:**
  - $\mathcal{I}[\![\tau_C]\!] = \mathscr{D}(C) \,\dot\cup\, \{\bot\}, \quad \mathcal{I}[\![\tau_\bot]\!] = \mathscr{D}(\tau) \,\dot\cup\, \{\bot\}, \quad \mathcal{I}[\![2^{\tau_C}]\!] = \mathscr{D}(C_*) \,\dot\cup\, \{\bot\}$
  - $\mathcal{I}[\![\mathbb{B}_\bot]\!] = \{\mathit{true}, \mathit{false}\} \,\dot\cup\, \{\bot\}, \quad \mathcal{I}[\![\mathbb{Z}_\bot]\!] = \mathbb{Z} \,\dot\cup\, \{\bot\}$

- **Functions:**
  - We assume $f_\mathcal{I}$ given for each function symbol f ($\to$ in a minute).

- **Proto-OCL Semantics** (interpretation function):
  - $\mathcal{I}[\![c]\!](\sigma, \beta) = \beta(c)$ (assuming β is a type-consistent valuation of the logical variables),
  - $\mathcal{I}[\![\mathit{allInstances}_C]\!](\sigma, \beta) = \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$,
  - $\underbrace{\mathcal{I}[\![v(F)]\!](\sigma, \beta)}_{:\tau_C} = \begin{cases} \sigma\,(\underbrace{\mathcal{I}[\![F]\!](\sigma, \beta)}_{:\tau_C})(v) & \text{, if } \mathcal{I}[\![F]\!](\sigma, \beta) \in \mathrm{dom}(\sigma) \\ \bot & \text{, otherwise} \end{cases}$ (if not $v : C_{0,1}$)

    $\mathcal{I}[\![F]\!](\sigma,\beta) \in \mathrm{dom}(\sigma), \quad \sigma(\mathcal{I}[\![F]\!](\sigma,\beta))(v) = \{u'\} \boxed{\cancel{\subseteq \mathrm{dom}(\sigma)}}$
  - $\mathcal{I}[\![v(F)]\!](\sigma, \beta) = \begin{cases} \overset{u'}{\cancel{\sigma(u')(v)}} & \text{, if } \cancel{\mathcal{I}[\![F]\!](\sigma, \beta) = \{u'\} \subseteq \mathrm{dom}(\sigma)} \\ \bot & \text{, otherwise} \end{cases}$ (if $v : C_{0,1}$)
  - $\mathcal{I}[\![f(F_1, \dots, F_n)]\!](\sigma, \beta) = f_\mathcal{I}(\mathcal{I}[\![F_1]\!](\sigma, \beta), \dots, \mathcal{I}[\![F_n]\!](\sigma, \beta))$,
  - $\mathcal{I}[\![\forall\, c \in F_1 \bullet F_2]\!](\sigma, \beta) = \begin{cases} \mathit{true} & \text{, if } \mathcal{I}[\![F_2]\!](\sigma, \beta[c := u]) = \mathit{true} \text{ for all } u \in \mathcal{I}[\![F_1]\!](\sigma, \beta) \\ \mathit{false} & \text{, if } \mathcal{I}[\![F_2]\!](\sigma, \beta[c := u]) = \mathit{false} \text{ for some } u \in \mathcal{I}[\![F_1]\!](\sigma, \beta) \\ \bot & \text{, otherwise} \end{cases}$

Semantics Cont'd

- Proto-OCL is a **three-valued** logic: a formula evaluates to *true*, *false*, or $\bot$.
- **Example**: $\wedge_\mathcal{I}(\cdot,\cdot) : \{\mathit{true}, \mathit{false}, \bot\} \times \{\mathit{true}, \mathit{false}, \bot\} \to \{\mathit{true}, \mathit{false}, \bot\}$ is defined as follows:

x_1	*true*	*true*	*true*	*false*	*false*	*false*	$\bot$	$\bot$	$\bot$
x_2	*true*	*false*	$\bot$	*true*	*false*	$\bot$	*true*	*false*	$\bot$
$\wedge_\mathcal{I}(x_1, x_2)$	*true*	*false*	$\bot$	*false*	*false*	*false*	$\bot$	*false*	$\bot$

  We assume common logical connectives $\neg, \wedge, \vee, \dots$ with canonical 3-valued interpretation.

- **Example**: $+_\mathcal{I}(\cdot,\cdot) : (\mathbb{Z} \,\dot\cup\, \{\bot\}) \times (\mathbb{Z} \,\dot\cup\, \{\bot\}) \to \mathbb{Z} \,\dot\cup\, \{\bot\}$

$$+_\mathcal{I}(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{, if } x_1 \neq \bot \text{ and } x_2 \neq \bot \\ \bot & \text{, otherwise} \end{cases}$$

  We assume common arithmetic operations $-, /, *, \dots$
  and relation symbols $>, <, \leq, \dots$ with **monotone** 3-valued interpretation.

- And we assume the special unary function symbol $\mathit{isUndefined}$:

$$\mathit{isUndefined}_\mathcal{I}(x) = \begin{cases} \mathit{true} & \text{, if } x = \bot, \\ \mathit{false} & \text{, otherwise} \end{cases}$$

  $\mathit{isUndefined}_\mathcal{I}$ is **definite**: it never yields $\bot$.

Example: Evaluate Formula for System State

$\underbrace{\{1_C \mapsto \underbrace{\{x \mapsto 13\}}_{\sigma(1_C)}\}}_{(\sigma(1_C))(x)}$

σ :

1_C : C
$x = 13$

$\in \Sigma_\mathscr{S}^\mathscr{D}$

$\mathscr{S}$:

C
$x : Int$

$$\forall\, c \in allInstances_C \bullet x(c) \neq 27$$

- Recall **prefix notation**: $\forall\, c \in allInstances_C \bullet \neq(x(c), 27)$

  Note: $\neq$ is a binary function symbol, 27 is a 0-ary function symbol.

- **Example**:

  $\mathcal{I}[\![\forall\, c \in allInstances_C \bullet \neq(x(c), 27)]\!](\sigma, \emptyset) =$ *true*, because…

  $\mathcal{I}[\![\neq(x(c), 27)]\!](\sigma, \beta), \quad \beta := \emptyset[c := 1_C] = \{c \mapsto 1_C\}$

  $= \neq_\mathcal{I}(\, \mathcal{I}[\![x(c)]\!](\sigma, \beta), \; \mathcal{I}[\![27]\!](\sigma, \beta)\,)$

  $= \neq_\mathcal{I}(\, \sigma(\, \mathcal{I}[\![c]\!](\sigma, \beta)\,)(x), \; 27_\mathcal{I}\,)$

  $= \neq_\mathcal{I}(\, \sigma(\, \beta(c)\,)(x), \; 27_\mathcal{I}\,)$

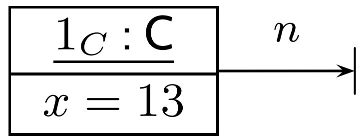
  $= \neq_\mathcal{I}(\, \sigma(\, 1_C\,)(x), \; 27_\mathcal{I}\,)$

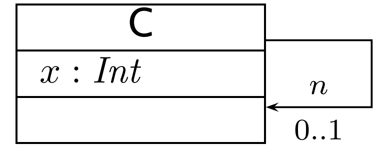
  $= \neq_\mathcal{I}(\, 13, \; 27\,) =$ *true* …and 1_C is the only C-object in σ: $\mathcal{I}[\![allInstances_C]\!](\sigma, \emptyset) = \{1_C\}$.

More Interesting Example

$\sigma = \{1_C \mapsto \{x \mapsto 13, n \mapsto \emptyset\}\}$

σ :





$$\forall\, c : allInstances_C \bullet x(n(c)) \neq 27$$

- Similar to the previous slide, we need the value of

$$\mathcal{I}[\![x(n(c))]\!](\sigma, \beta),\, \beta = \{c \mapsto 1_C\}$$

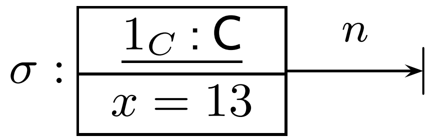
  - $\mathcal{I}[\![c]\!](\sigma, \beta) = \beta(c) = 1_C$
  - $\mathcal{I}[\![n(c)]\!](\sigma, \beta) = \bot$ since $\sigma(\, \mathcal{I}[\![c]\!](\sigma, \beta)\,)(n) = \emptyset \neq \{u'\}$ by rule

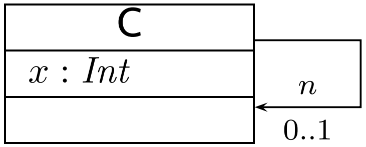
$$\mathcal{I}[\![v(F)]\!](\sigma, \beta) = \begin{cases} u' & \text{, if } \mathcal{I}[\![F]\!](\sigma, \beta) \in \mathrm{dom}(\sigma) \text{ and } \sigma(\mathcal{I}[\![F]\!](\sigma, \beta))(v) = \{u'\} \\ \bot & \text{, otherwise} \end{cases} \qquad (\text{if } v : C_{0,1})$$

  - $\mathcal{I}[\![x(n(c))]\!](\sigma, \beta) = \bot$ since $\mathcal{I}[\![n(c)]\!](\sigma, \beta) = \bot$ by rule

$$\mathcal{I}[\![v(F)]\!](\sigma, \beta) = \begin{cases} \sigma\left(\mathcal{I}[\![F]\!](\sigma, \beta)\right)(v) & \text{, if } \mathcal{I}[\![F]\!](\sigma, \beta) \in \mathrm{dom}(\sigma) \\ \bot & \text{, otherwise} \end{cases} \qquad (\text{if not } v : C_{0,1})$$

More Interesting Example





allInstances$_C$

$$\forall\, c : \cancel{C} \bullet x(n(c)) \neq 27$$

- Similar to the previous slide, we need the value of

$$\sigma\ (\ \sigma(\ \mathcal{I}[\![c]\!](\sigma,\beta)\ )(n)\ )\ (x)$$

  - $\mathcal{I}[\![c]\!](\sigma,\beta) = \beta(c) = 1_C$
  - $\sigma(\ \mathcal{I}[\![c]\!](\sigma,\beta)\ )(n) = \sigma(\ 1_C\ )(n) = \emptyset$
  - $\sigma\ (\ \sigma(\ \mathcal{I}[\![c]\!](\sigma,\beta)\ )(n)\ )\ (x) = \bot$

by the following rule:

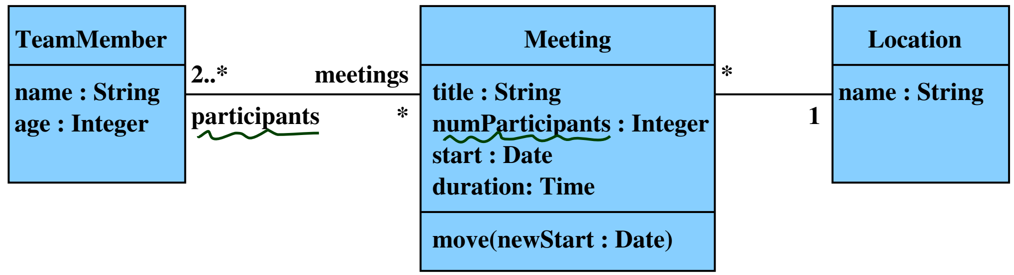
$$\mathcal{I}[\![v(F)]\!](\sigma,\beta) = \begin{cases} \sigma(u')(v) & \text{, if } \underline{\mathcal{I}[\![F]\!](\sigma,\beta) = \{u'\} \subseteq \mathrm{dom}(\sigma)} \\ \bot & \text{, otherwise} \end{cases} \qquad \text{(if } v : C_{0,1}\text{)}$$

Object Constraint Language (OCL)

OCL is the same – just with less readable (?) syntax.

Literature: (OMG, 2006; Warmer and Kleppe, 1999).

Examples (from lecture “Softwaretechnik 2008”)



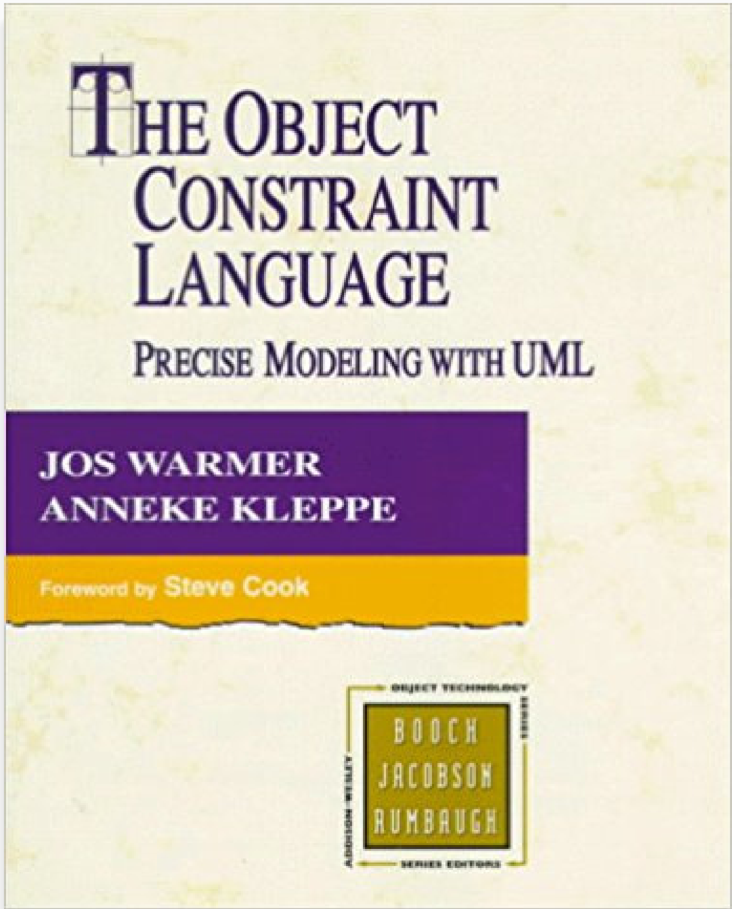
- **context** Meeting
  - **inv:** self.participants->size() =
    self. numParticipants
- **context** Location
  - **inv:** name="Lobby" **implies**
    meeting->isEmpty()

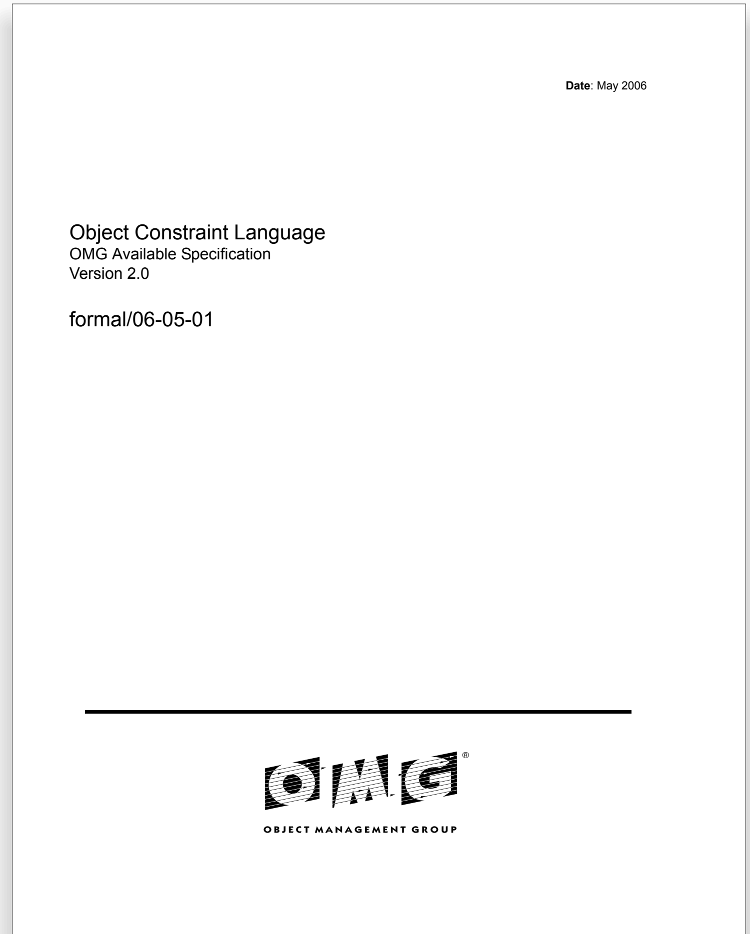
Prof. Dr. P. Thiemann, http://proglang.informatik.uni-freiburg.de/teaching/swt/2008/

$\forall\, self \in AllInstances_{Meeting} \bullet$

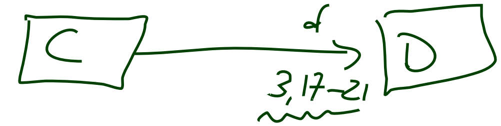
$size(participants(self)) = numParticipants(self)$

Literature


THE OBJECT CONSTRAINT LANGUAGE
PRECISE MODELING WITH UML
JOS WARMER
ANNEKE KLEPPE
Foreword by Steve Cook
BOOCH
JACOBSON
RUMBAUGH

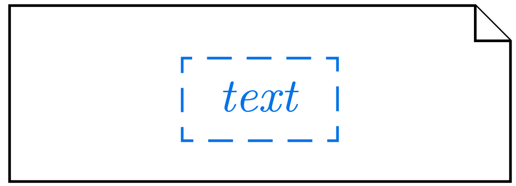

Date: May 2006
Object Constraint Language
OMG Available Specification
Version 2.0
formal/06-05-01
OBJECT MANAGEMENT GROUP

Where To Put OCL Constraints?



Type: d : D_*
Constraint:
forall c in AllInstances_C .
size(d(c)) = 3 or
size(d(c)) >= 17
and size(d(c)) <= 21

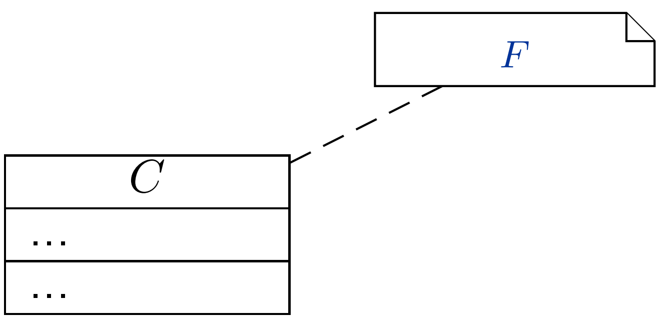
- **Notes**: A UML **note** is a diagram element of the form



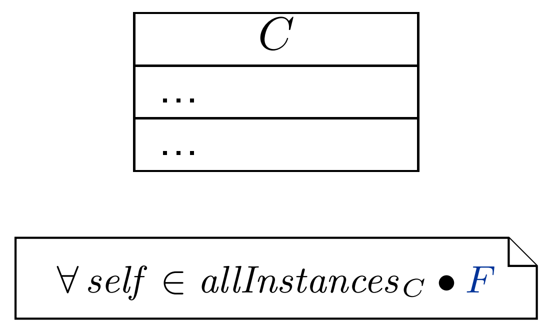
$text$ can principally be **everything**, in particular **comments** and **constraints**.

Sometimes, content is **explicitly classified** for clarity:

- Conventions:



stands for



Content

- **Proto-OCL**
  - syntax, semantics,
  - Proto-OCL vs. OCL.
  - Proto-OCL vs. Software
- An outlook on **UML**
- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ...by example
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

Putting It All Together

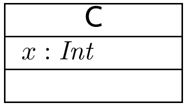
Modelling Structure with Class Diagrams

> **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.

- The set of **states** Σ could be the set of **system states** as defined by a class diagram, e.g.

$$\Sigma := \Sigma_{\mathscr{S}}^{\mathscr{D}}$$

$\mathscr{S}$:



- A corresponding **computation path** of a software S could be

$$\boxed{\begin{array}{c} \underline{27_C : \mathsf{C}} \\ x = 0 \end{array}} \xrightarrow{\tau} \boxed{\begin{array}{c} \underline{27_C : \mathsf{C}} \\ x = 1 \end{array}} \xrightarrow{\tau} \boxed{\begin{array}{c} \underline{27_C : \mathsf{C}} \\ x = 3 \end{array}} \xrightarrow{\tau} \boxed{\begin{array}{c} \underline{27_C : \mathsf{C}} \\ x = 4 \end{array}} \xrightarrow{\tau} \ldots$$

- If a requirement is formalised by the Proto-OCL constraint

$$F = \forall\, c \in allInstances_C \bullet x(c) < 4$$

then S **does not** satisfy the requirement.

More General: Software vs. Proto-OCL

- Let $\mathscr{S}$ be an **object system signature** and $\mathscr{D}$ a **structure**.
- Let S be a **software** with
  - states $\Sigma \subseteq \Sigma_{\mathscr{S}}^{\mathscr{D}}$, and
  - **computation paths** $[\![S]\!]$.
- Let F be a Proto-OCL constraint over $\mathscr{S}$.
- We say $[\![S]\!]$ **satisfies** F, denoted by $[\![S]\!] \models F$, if and only if for all

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]$$

  and all $i \in \mathbb{N}_0$,

$$\mathcal{I}[\![F]\!](\sigma_i, \emptyset) = \textit{true}.$$

- We say $[\![S]\!]$ **does not satisfy** F, denoted by $[\![S]\!] \not\models F$, if and only if there exists $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]$ and $i \in \mathbb{N}_0$, such that $\mathcal{I}[\![F]\!](\sigma_i, \emptyset) = \textit{false}$.
- **Note**: $\neg([\![S]\!] \not\models F)$ does not imply $[\![S]\!] \models F$.

Tell Them What You've Told Them...

- **Class Diagrams** can be used to **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** $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$
  - can be **visualised** by an **object diagram**.

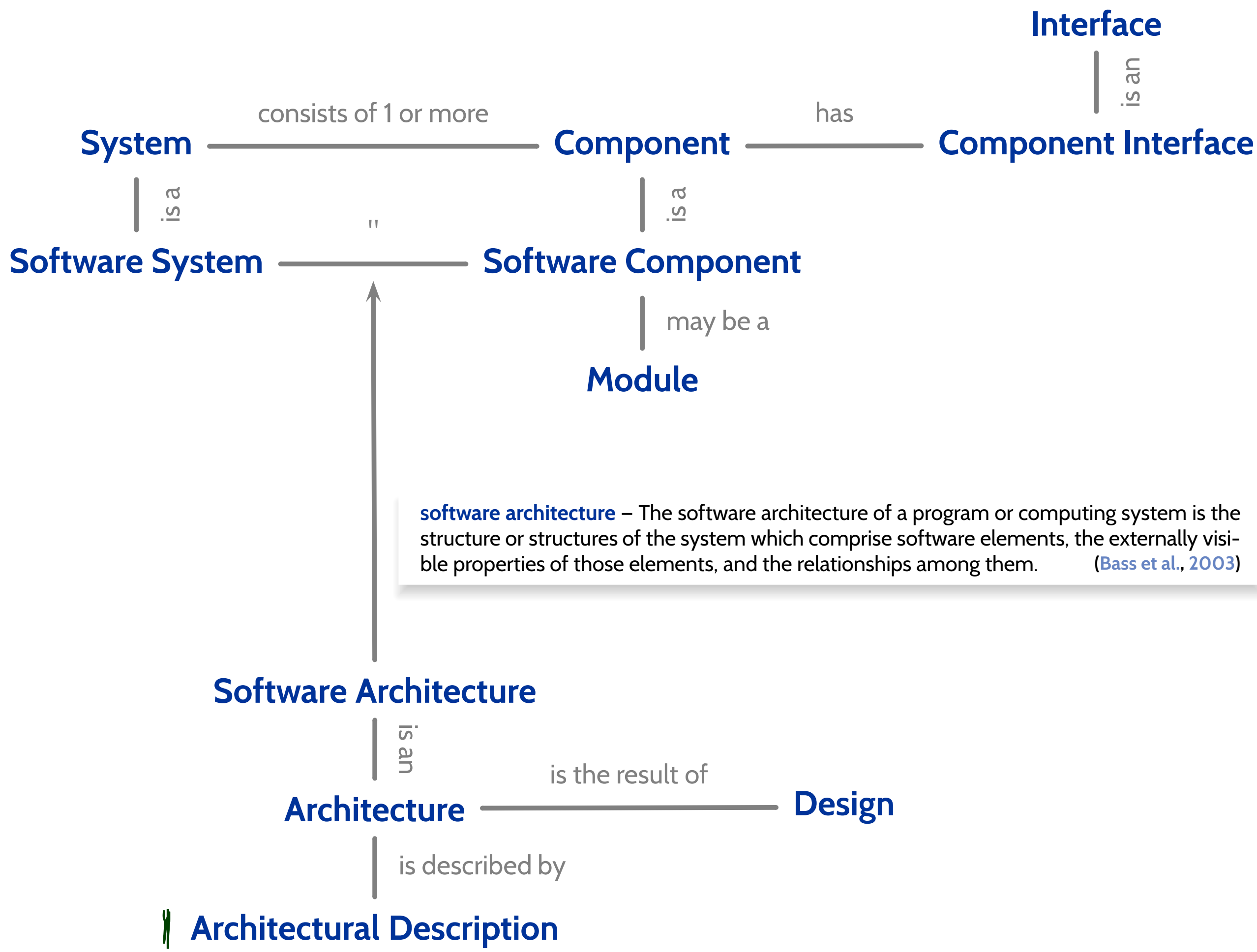
- **Proto-OCL** constraints can be evaluated on **system states**.

- A **software** over $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ satisfies a Proto-OCL constraint F if and only if F evaluates to *true* in all system states of all the software's computation paths.

Content

- **Proto-OCL**
  - syntax, semantics,
  - Proto-OCL vs. OCL.
  - Proto-OCL vs. Software
- ~~An outlook on **UML**~~
- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ...by example
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

Once Again, Please



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

Goals and Relevance of Design

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

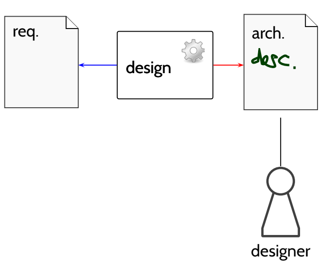
Design…

(i) **structures** a system into manageable units (yields software architecture),

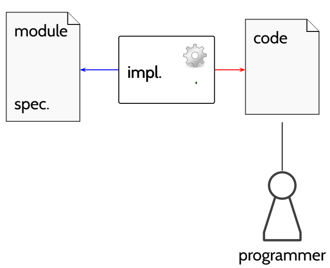
(ii) **determines** the approach for realising the required software,

(iii) provides **hierarchical structuring** into a manageable number of units at each hierarchy level.

Oversimplified process model "Design":

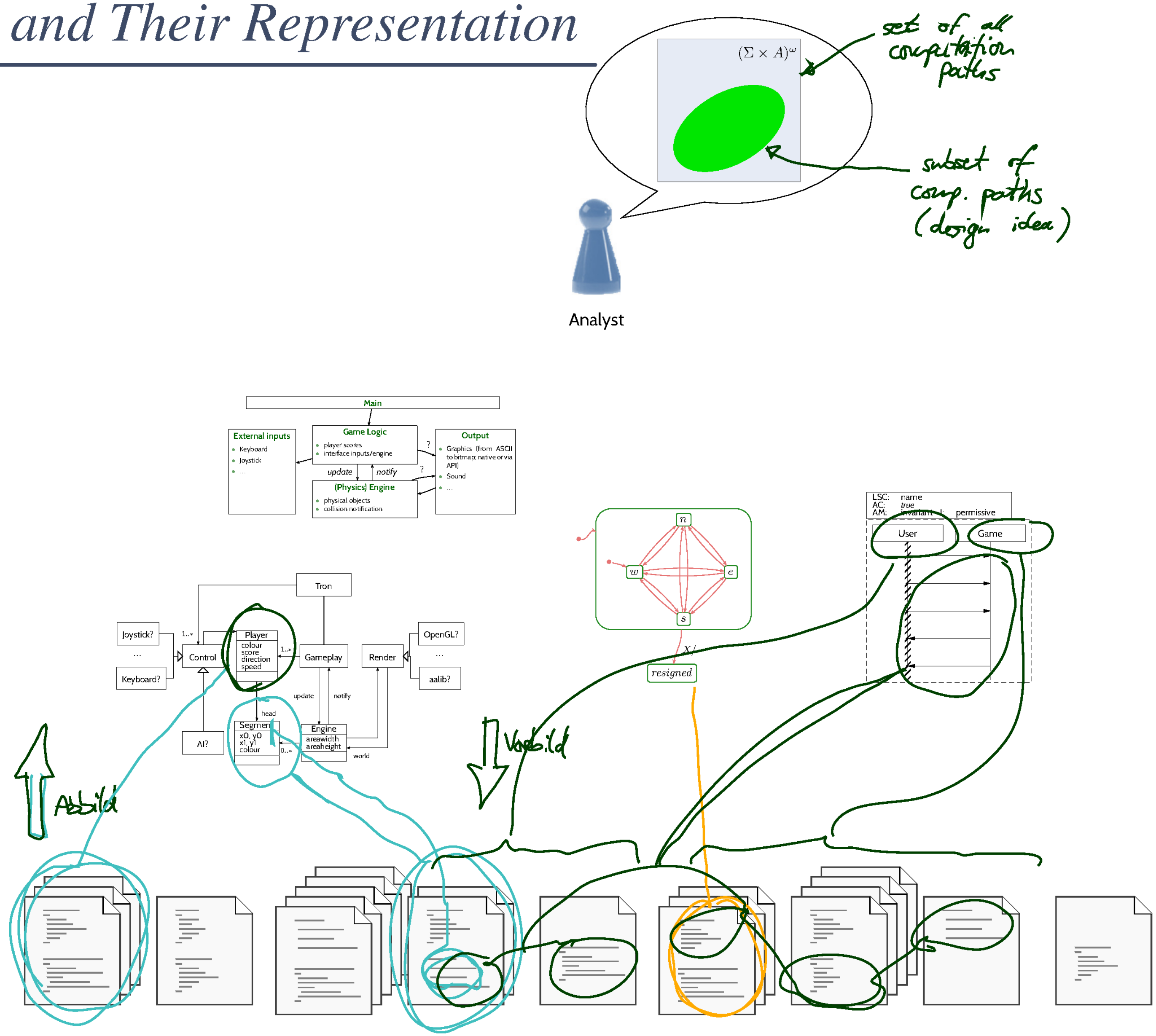


design



implementation

Views and Their Representation



Principles of (Architectural) Design

Overview

1.) **Modularisation**

- split software into units / components of **manageable size**
- provide well-defined interface

2.) **Separation of Concerns**

- each component should be **responsible for a particular area of tasks**
- group data and operation on that data; functional aspects; functional vs. technical; functionality and interaction

3.) **Information Hiding**

- the "need to know principle" / information hiding
- users (e.g. other developers) need not necessarily know the algorithm and helper data which realise the component's interface

4.) **Data Encapsulation**

- offer operations to access component data,
  instead of accessing data (variables, files, etc.) directly

→ many programming languages and systems offer means to **enforce** (some of) these principles **technically**; use these means.

1.) Modularisation

modular decomposition – The process of breaking a system into components to facilitate design and development; an element of modular programming. **IEEE 610.12 (1990)**

modularity – The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. **IEEE 610.12 (1990)**

- So, **modularity** is a **property** of an architecture.
- Goals of modular decomposition:
  - The **structure** of each module should be **simple** and **easily comprehensible**.
  - The **implementation** of a module should be **exchangeable**;
    information on the implementation of other modules should not be necessary.
    The other modules should not be affected by implementation exchanges.
  - Modules should be designed such that **expected changes**
    do not require modifications of the **module interface**.
  - **Bigger changes** should be the result of a set of **minor changes**.
    As long as the interface does not change,
    it should be possible to test old and new versions of a module together.

2.) Separation of Concerns

- **Separation of concerns** is a fundamental principle in software engineering:
  - each component should be **responsible for a particular area of tasks**,
  - components which try to cover different task areas tend to be unnecessarily complex, thus hard to understand and maintain.

- **Criteria** for separation/grouping:
  - in **object oriented design**, data and operations on that data are grouped into classes,
  - sometimes, functional aspects (features) like printing are realised as separate components,
  - separate **functional** and **technical** components,

    Example: logical flow of (logical) messages in a communication protocol (**functional**) vs. exchange of (physical) messages using a certain technology (**technical**).
  - assign flexible or variable functionality to own components.

    Example: different networking technology (wireless, etc.)
  - assign functionality which is expected to need extensions or changes later to own components.
  - separate system **functionality** and **interaction**

    Example: most prominently graphical user interfaces (GUI), also file input/output

3.) Information Hiding

- By now, we only discussed the **grouping** of data and operations.
  One should also consider **accessibility**.
- The "**need to know principle**" is called **information hiding** in SW engineering. (Parnas, 1972)

> **information hiding**– A software development technique in which each module's interfaces reveal as little as possible about the module's inner workings, and other modules are prevented from using information about the module that is not in the module's interface specification. **IEEE 610.12 (1990)**

- **Note**: what is hidden is information which other components **need not know**
  (e.g., how data is stored and accessed, how operations are implemented).

  In other words: information hiding is about making explicit for one component which data or operations other components may use of this component.

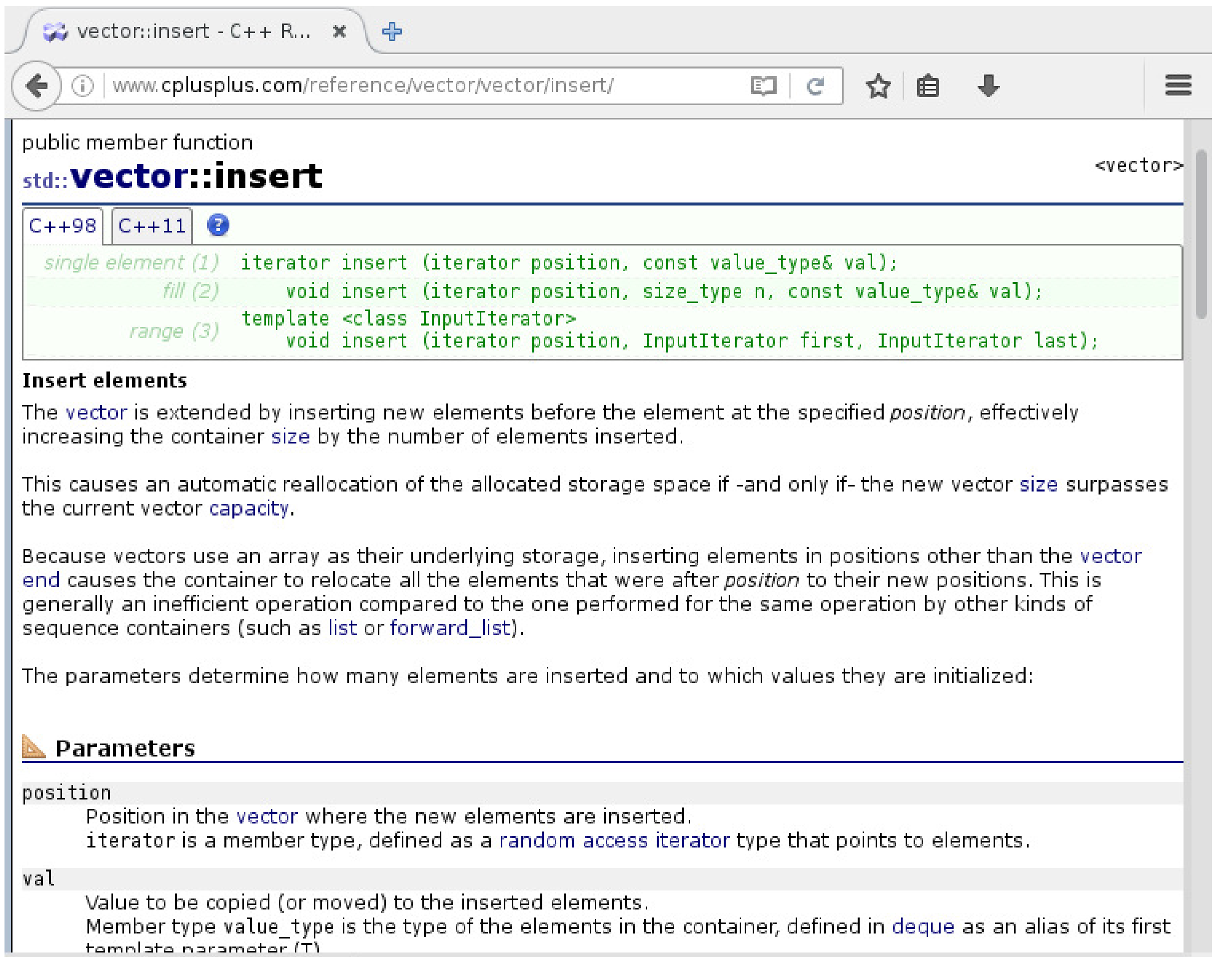
- **Advantages / goals**:
  - Hidden solutions may be **changed** without other components noticing, as long as the visible behaviour stays the same (e.g. the employed sorting algorithm).

    IOW: other components cannot (unintentionally) depend on details they are not supposed to.
  - Components can be verified / validated in isolation.

4.) Data Encapsulation

- Similar direction: **data encapsulation** (examples later).
  - Do not access data (variables, files, etc.) directly where needed, but encapsulate the data in a component which offers operations to access (read, write, etc.) the data.

    Real-World Example: Users do not write to bank accounts directly, only bank clerks do.



vector::insert - C++ R...

www.cplusplus.com/reference/vector/vector/insert/

public member function

std::**vector::insert** <vector>

C++98 C++11

single element (1)	`iterator insert (iterator position, const value_type& val);`
fill (2)	`void insert (iterator position, size_type n, const value_type& val);`
range (3)	`template <class InputIterator> void insert (iterator position, InputIterator first, InputIterator last);`

Insert elements

The vector is extended by inserting new elements before the element at the specified *position*, effectively increasing the container size by the number of elements inserted.

This causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

Because vectors use an array as their underlying storage, inserting elements in positions other than the vector end causes the container to relocate all the elements that were after *position* to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as list or forward_list).

The parameters determine how many elements are inserted and to which values they are initialized:

Parameters

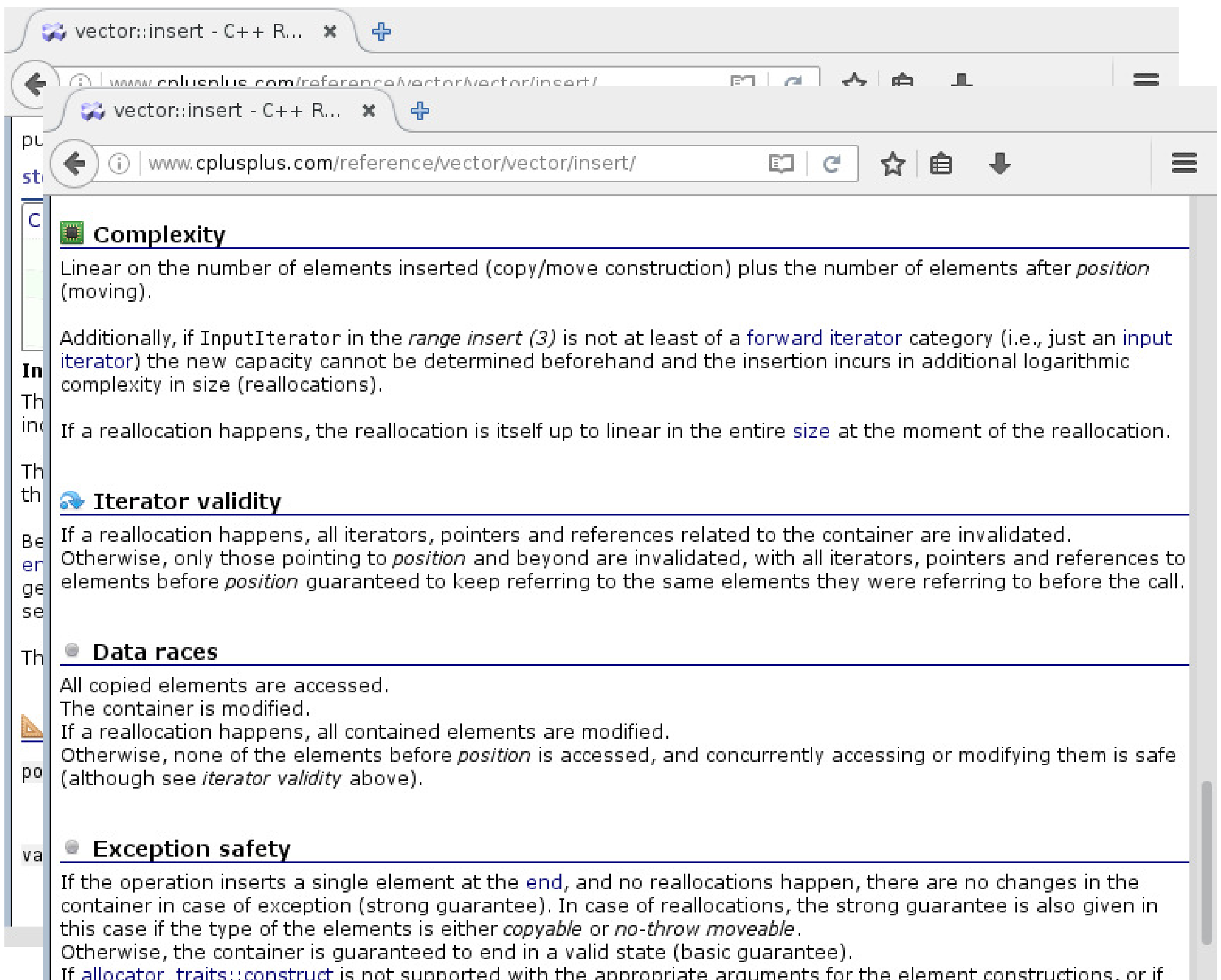
`position`
Position in the vector where the new elements are inserted.
iterator is a member type, defined as a random access iterator type that points to elements.

`val`
Value to be copied (or moved) to the inserted elements.
Member type value_type is the type of the elements in the container, defined in deque as an alias of its first template parameter (T).

4.) Data Encapsulation

- Similar direction: **data encapsulation** (examples later).
  - Do not access data (variables, files, etc.) directly where needed, but encapsulate the data in a component which offers operations to access (read, write, etc.) the data.

    Real-World Example: Users do not write to bank accounts directly, only bank clerks do.



Complexity

Linear on the number of elements inserted (copy/move construction) plus the number of elements after *position* (moving).

Additionally, if InputIterator in the *range insert (3)* is not at least of a forward iterator category (i.e., just an input iterator) the new capacity cannot be determined beforehand and the insertion incurs in additional logarithmic complexity in size (reallocations).

If a reallocation happens, the reallocation is itself up to linear in the entire size at the moment of the reallocation.

Iterator validity

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.
Otherwise, only those pointing to *position* and beyond are invalidated, with all iterators, pointers and references to elements before *position* guaranteed to keep referring to the same elements they were referring to before the call.

Data races

All copied elements are accessed.
The container is modified.
If a reallocation happens, all contained elements are modified.
Otherwise, none of the elements before *position* is accessed, and concurrently accessing or modifying them is safe (although see *iterator validity* above).

Exception safety

If the operation inserts a single element at the end, and no reallocations happen, there are no changes in the container in case of exception (strong guarantee). In case of reallocations, the strong guarantee is also given in this case if the type of the elements is either *copyable* or *no-throw moveable*.
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).
If allocator_traits::construct is not supported with the appropriate arguments for the element constructions, or if

4.) Data Encapsulation

- Similar direction: **data encapsulation** (examples later).
  - Do not access data (variables, files, etc.) directly where needed, but encapsulate the data in a component which offers operations to access (read, write, etc.) the data.

    Real-World Example: Users do not write to bank accounts directly, only bank clerks do.

- **Information hiding** and **data encapsulation** – when enforced technically (examples later) – usually **come at the price** of worse efficiency.
  - It is more efficient to read a component's data directly than calling an operation to provide the value: there is an overhead of one operation call.
  - Knowing how a component works internally may enable more efficient operation.

    Example: if a sequence of data items is stored as a singly-linked list, accessing the data items in list-order may be more efficient than accessing them in reverse order by position.

    Good modules give usage hints in their documentation (e.g. C++ standard library).

    Example: if an implementation stores intermediate results at a certain place, it may be tempting to "quickly" read that place when the intermediate results is needed in a different context.

    → **maintenance nightmare** – If the result is needed in another context, add a corresponding operation explicitly to the interface.

Yet with today's hardware and programming languages, this is hardly an issue any more; at the time of (Parnas, 1972), it clearly was.

A Classification of Modules (Nagl, 1990)

- **functional modules**
  - group computations which belong together logically,
  - do not have "memory" or state, that is, behaviour of offered functionality does not depend on prior program evolution,
  - **Examples**: mathematical functions, transformations
- **data object modules**
  - realise encapsulation of data,
  - a data module hides kind and structure of data, interface offers operations to manipulate encapsulated data
  - **Examples**: modules encapsulating global configuration data, databases
- **data type modules**
  - implement a user-defined data type in form of an abstract data type (ADT)
  - allows to create and use as many exemplars of the data type
  - **Example**: game object
- In an object-oriented design,
  - classes are **data type modules**,
  - **data object modules** correspond to classes offering only class methods or singletons ($\rightarrow$ later),
  - **functional modules** occur seldom, one example is Java's class `Math`.

Content

- **Proto-OCL**
  - syntax, semantics,
  - Proto-OCL vs. OCL.
  - Proto-OCL vs. Software
- An outlook on **UML**
- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - …by example
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

Architecture Patterns

Introduction

- Over decades of software engineering,
  many **clever**, **proved** and **tested** designs
  of solutions for particular problems emerged.

- **Question**: can we **generalise**, **document** and **re-use** these designs?

- **Goals**:
  - "**don't re-invent the wheel**",
  - benefit from "**clever**", from "**proven and tested**", and from "**solution**".

> **architectural pattern** – An architectural pattern expresses a fundamental structural organization schema for software systems.
>
> It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
>
> Buschmann et al. (1996)

Introduction Cont'd

> **architectural pattern** – An architectural pattern expresses a fundamental structural organization schema for software systems.
>
> It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
>
> Buschmann et al. (1996)

- **Using** an architectural pattern
  - **implies** certain characteristics or properties of the software (construction, extendibility, communication, dependencies, etc.),
  - **determines** structures on a high level of the architecture, thus is typically a central and fundamental design decision.
- The information that (where, how, ...) a well-known architecture / design pattern **is used** in a given software can
  - make **comprehension** and **maintenance** significantly easier,
  - avoid errors.

Layered Architectures

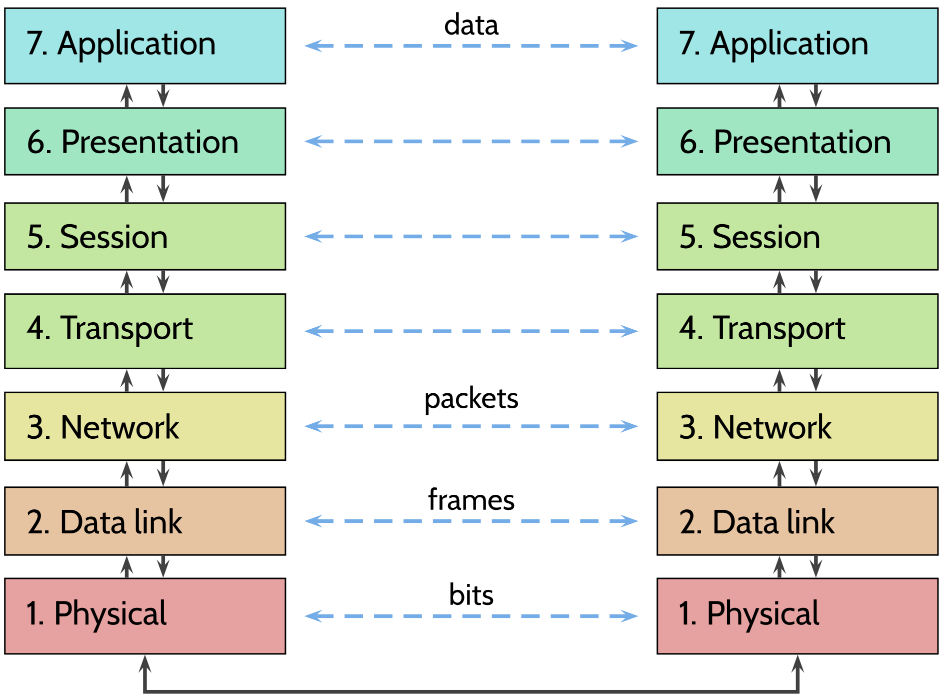
Example: Layered Architectures

- (Züllighoven, 2005):

  A **layer** whose components only interact with components of their **direct neighbour** layers is called **protocol-based** layer.

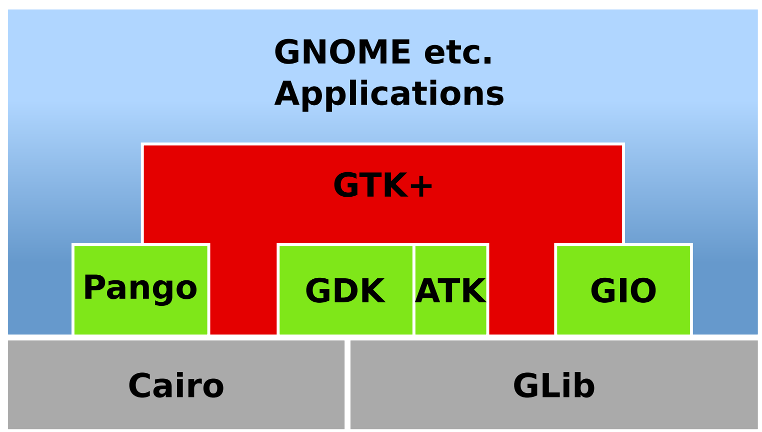
  A **protocol-based layer** hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.

- **Example: The ISO/OSI reference model.**



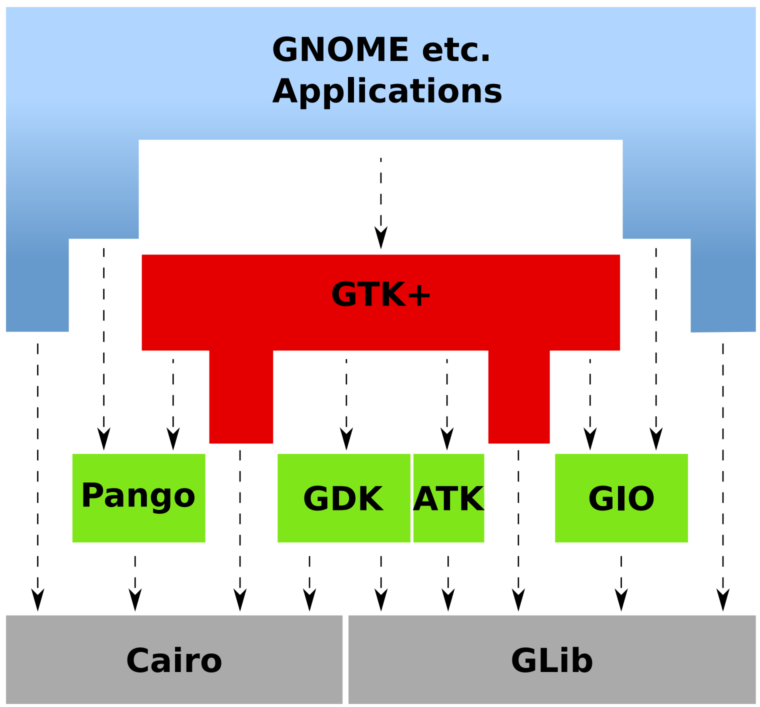
Example: Layered Architectures Cont'd

- **Object-oriented layer**: interacts with layers directly (and possibly further) above and below.
- **Rules**: the components of a layer may use
  - **only** components of the protocol-based layer directly beneath, or
  - **all** components of layers further beneath.



Example: Layered Architectures Cont'd

- **Object-oriented layer**: interacts with layers directly (and possibly further) above and below.
- **Rules**: the components of a layer may use
  - **only** components of the protocol-based layer directly beneath, or
  - **all** components of layers further beneath.



Example: Three-Tier Architecture

- **presentation layer** (or **tier**):

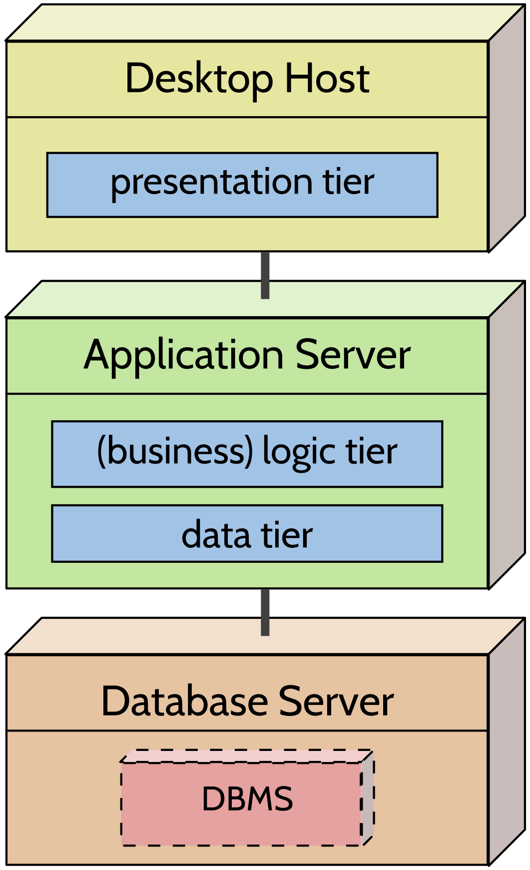
  user interface; presents information obtained from the logic layer to the user, controls interaction with the user, i.e. requests actions at the logic layer according to user inputs.

- **logic layer**:

  core system functionality; layer is designed without information about the presentation layer, may only read/write data according to data layer interface.

- **data layer**:

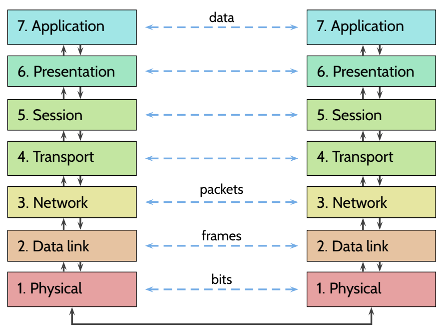
  persistent data storage; hides information about how data is organised, read, and written, offers particular chunks of information in a form useful for the logic layer.

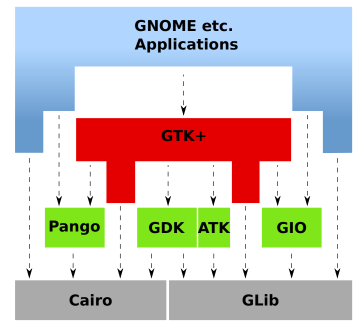


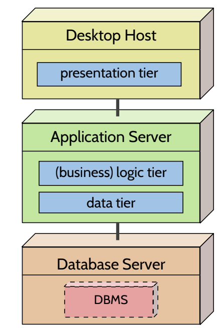
(Ludewig and Lichter, 2013)

- **Examples**: Web-shop, business software (enterprise resource planning), etc.

Layered Architectures: Discussion







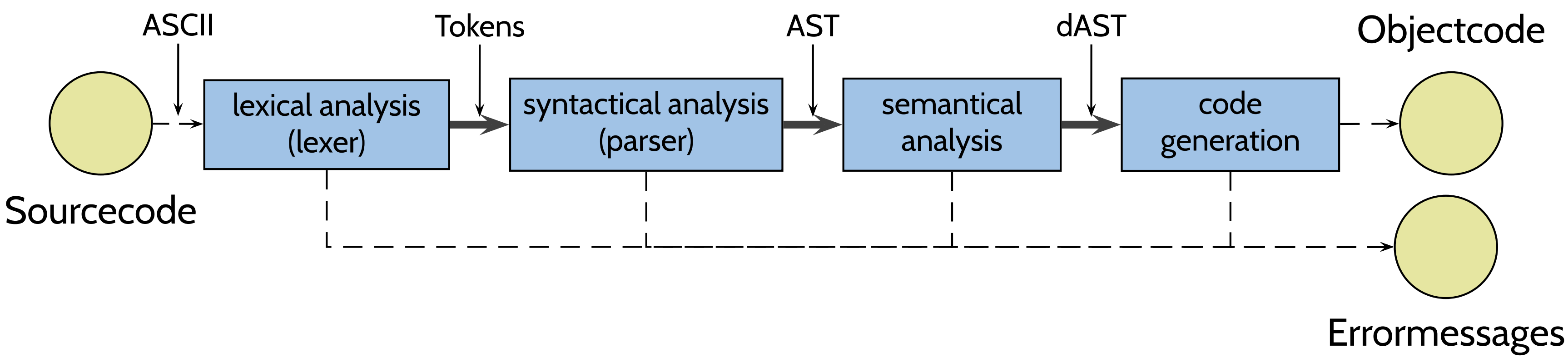
(Ludewig and Lichter, 2013)

- **Advantages**:
  - **protocol-based**:
    only neighouring layers are coupled, i.e. components of these layers interact,
  - coupling is low, data usually encapsulated,
  - changes have local effect (only neighbouring layers affected),
  - **protocol-based**: **distributed** implementation often easy.
- **Disadvantages**:
  - performance (as usual) – nowadays often not a problem.

Pipe-Filter

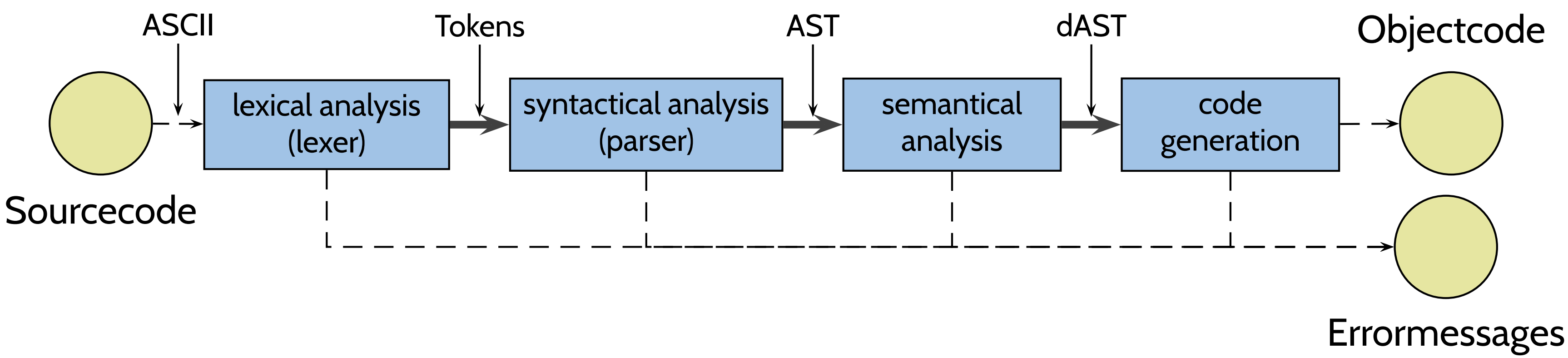
Example: Pipe-Filter

Example: **Compiler**



Example: Pipe-Filter

Example: **Compiler**



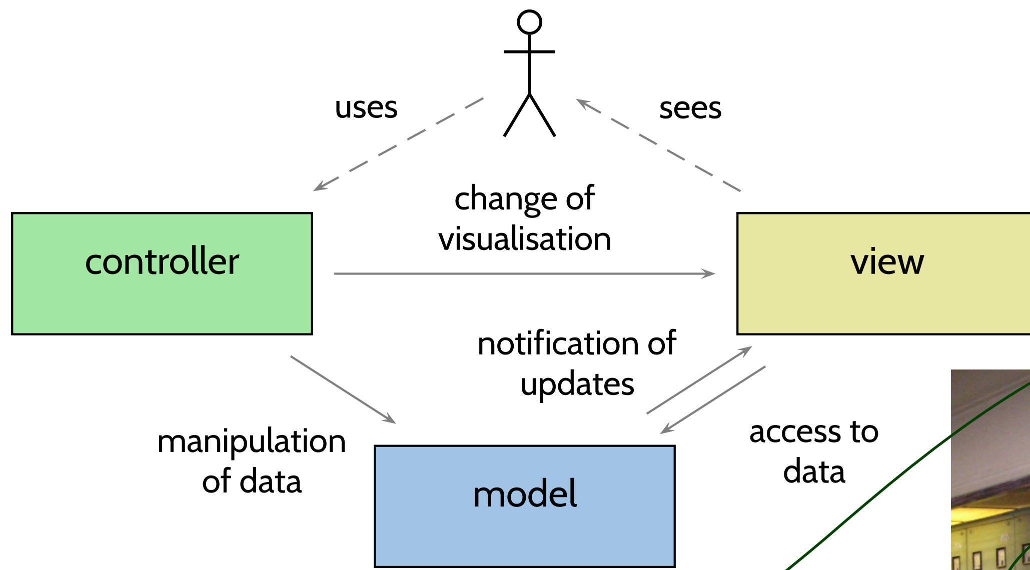
Example: **UNIX Pipes**

```
ls -l | grep Sarch.tex | awk '{ print $5 }'
```

- **Disadvantages**:
  - if the filters use a common data exchange format, all filters may need changes if the format is changed, or need to employ (costly) conversions.
  - filters do not use global data, in particular not to handle error conditions.

Model-View-Controller

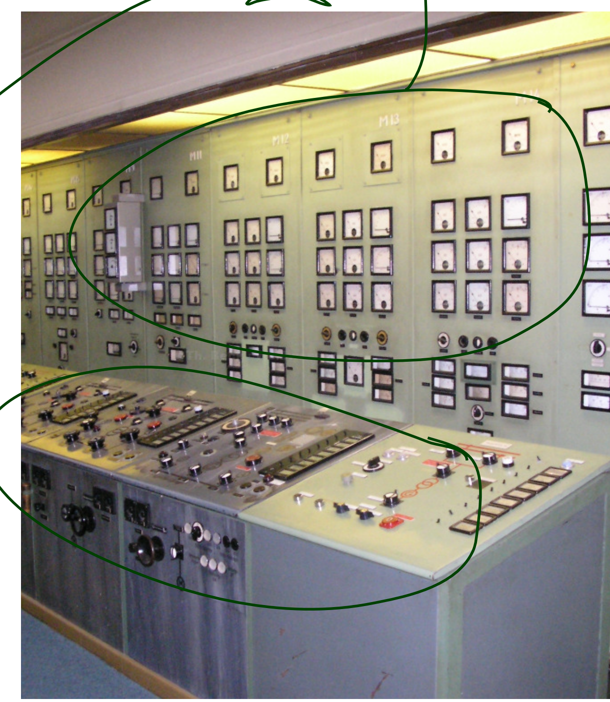
Example: Model-View-Controller



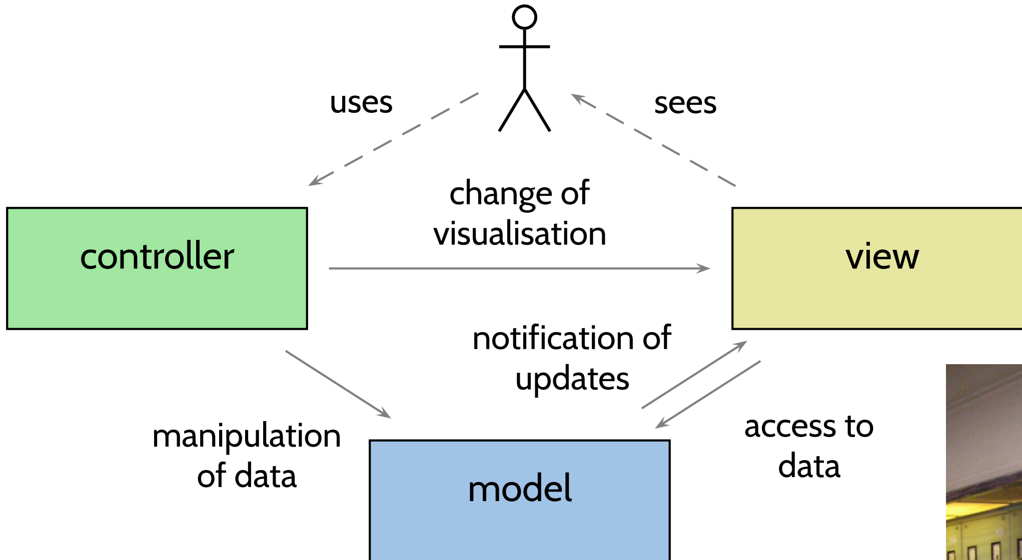
view

model

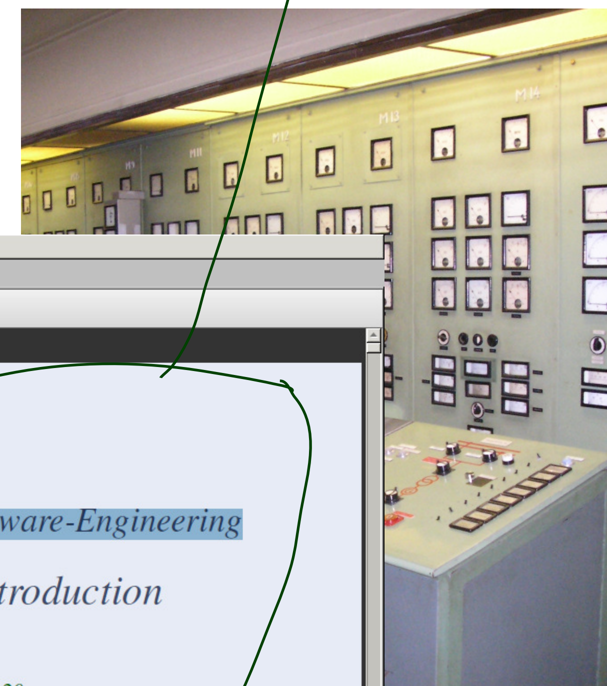
control



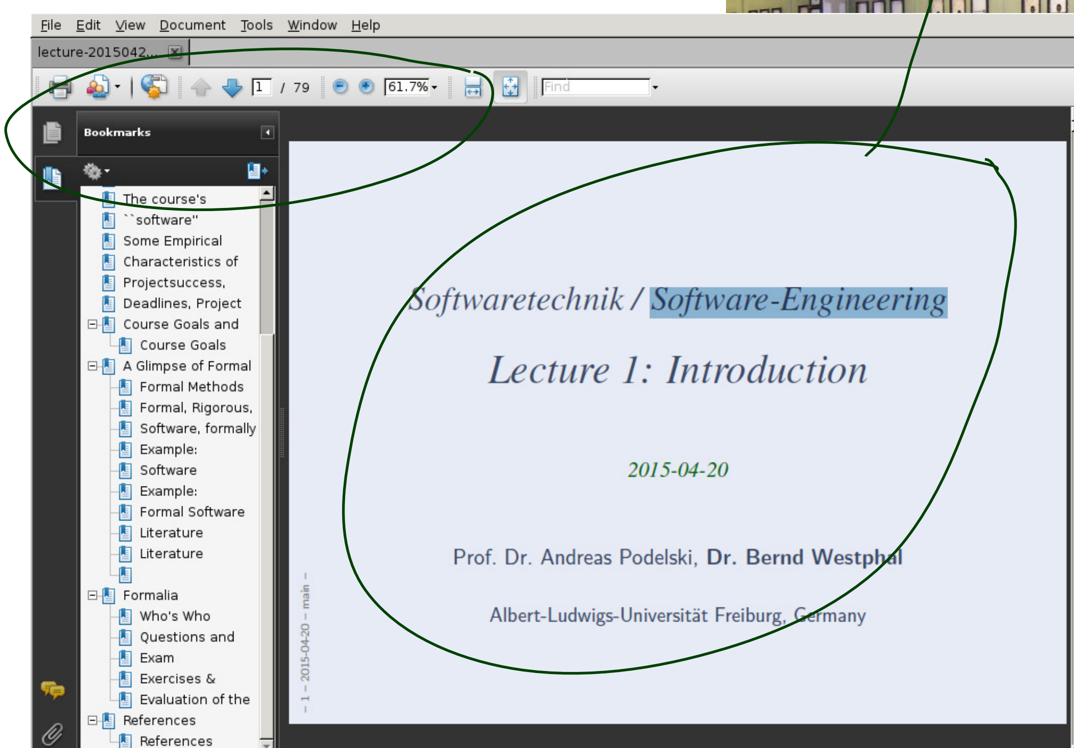
Example: Model-View-Controller



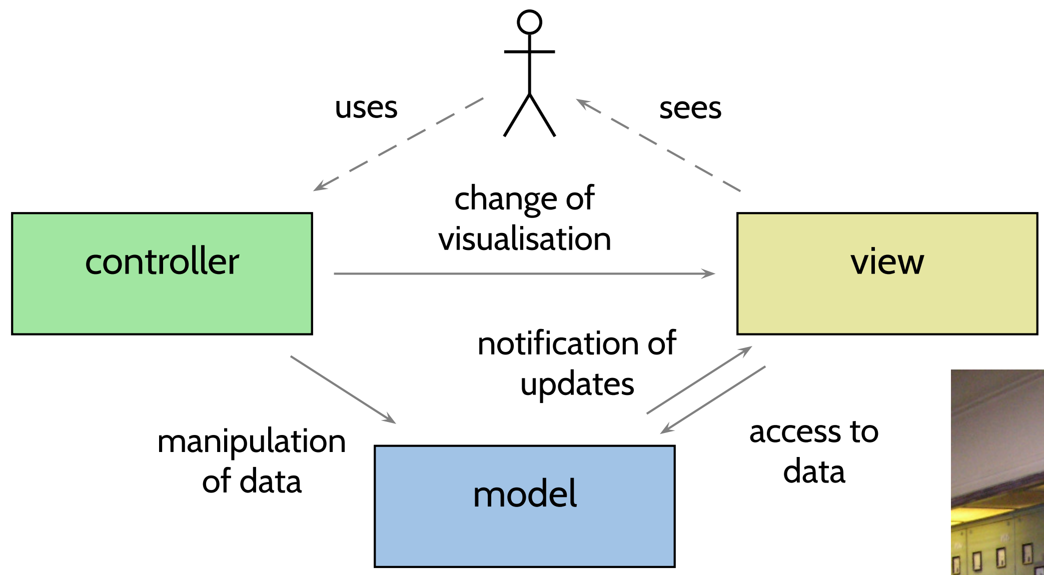
view



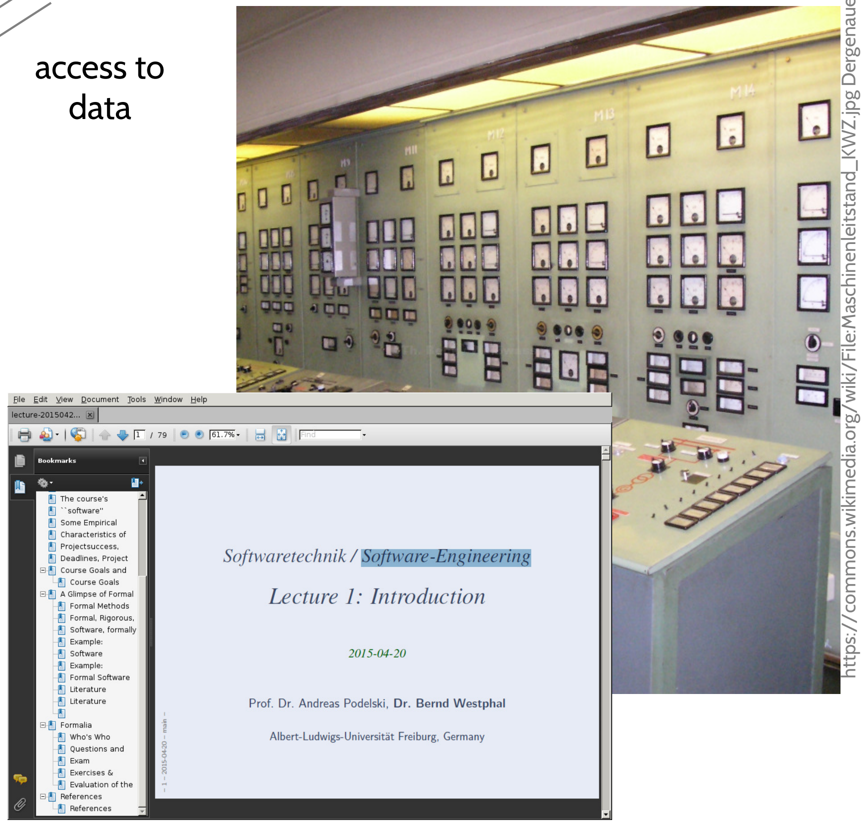
control



Example: Model-View-Controller



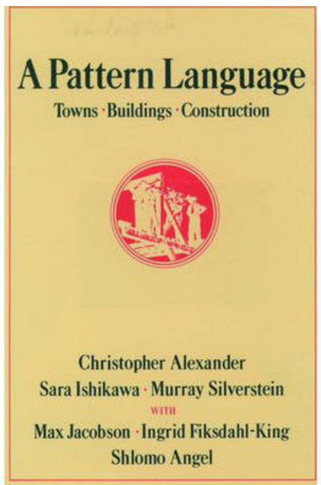
- **Advantages**:
  - one model can serve multiple view/controller pairs;
  - view/controller pairs can be added and removed at runtime;
  - model visualisation always up-to-date in all views;
  - distributed implementation (more or less) easily.
- **Disadvantages**:
  - if the view needs **a lot of data**, updating the view can be inefficient.

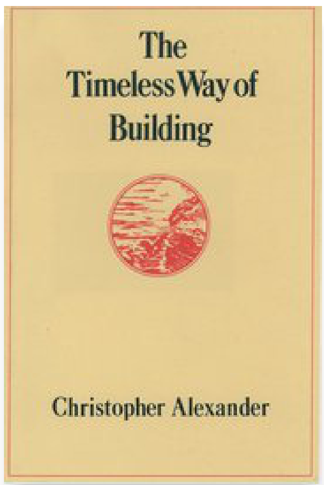


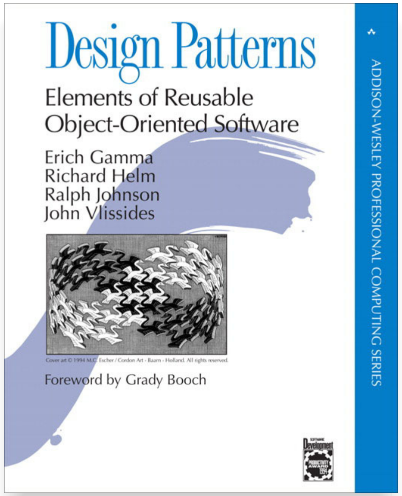
Design Patterns

Design Patterns

- In a sense the same as **architectural patterns**, but on a lower scale.
- Often traced back to (Alexander et al., 1977; Alexander, 1979).







> **Design patterns** ... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
>
> A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.
>
> (Gamma et al., 1995)

Tell Them What You've Told Them...

- **Architecture & Design Patterns**
  - allow **re-use** of practice-proven designs,
  - promise easier **comprehension** and **maintenance**.
- Notable **Architecture Patterns**
  - Layered Architecture,
  - Pipe-Filter,
  - Model-View-Controller.
- **Design Patterns**: read (Gamma et al., 1995)
- Rule-of-thumb:
  - **library modules** are called from user-code,
  - **framework modules** call user-code.

References

References

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language – Towns, Buildings, Construction*. Oxford University Press.

Booch, G. (1993). *Object-oriented Analysis and Design with Applications*. Prentice-Hall.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, E., and Stal, M. (1996). *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons.

Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–114.

Gamma, E., Helm, R., Johnsson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D., Lachover, H., et al. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

Jacobson, I., Christerson, M., and Jonsson, P. (1992). *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley.

JHotDraw (2007). `http://www.jhotdraw.org`.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

Nagl, M. (1990). *Softwaretechnik: Methodisches Programmieren im Großen*. Springer-Verlag.

OMG (2006). Object Constraint Language, version 2.0. Technical Report formal/06-05-01.

OMG (2007). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1990). *Object-Oriented Modeling and Design*. Prentice Hall.

Warmer, J. and Kleppe, A. (1999). *The Object Constraint Language*. Addison-Wesley.

Züllighoven, H. (2005). *Object-Oriented Construction Handbook - Developing Application-Oriented Software with the Tools and Materials Approach*. dpunkt.verlag/Morgan Kaufmann.

Content

- **Proto-OCL**
  - syntax, semantics,
  - Proto-OCL vs. OCL.
  - Proto-OCL vs. Software
- An outlook on **UML**
- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ...by example
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

A Brief History of the Unified Modelling Language (UML)

- Boxes/lines and finite automata are used to visualise software **for ages**.
- **1970's**, **Software Crisis**™
  – Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**
- Mid **1980**'s: **Statecharts** (Harel, 1987), **StateMate**™ (Harel et al., 1990)
- Early **1990's**, advent of **Object-Oriented**-Analysis/Design/Programming
  – Inflation of notations and methods, most prominent:

A Brief History of the Unified Modelling Language (UML)

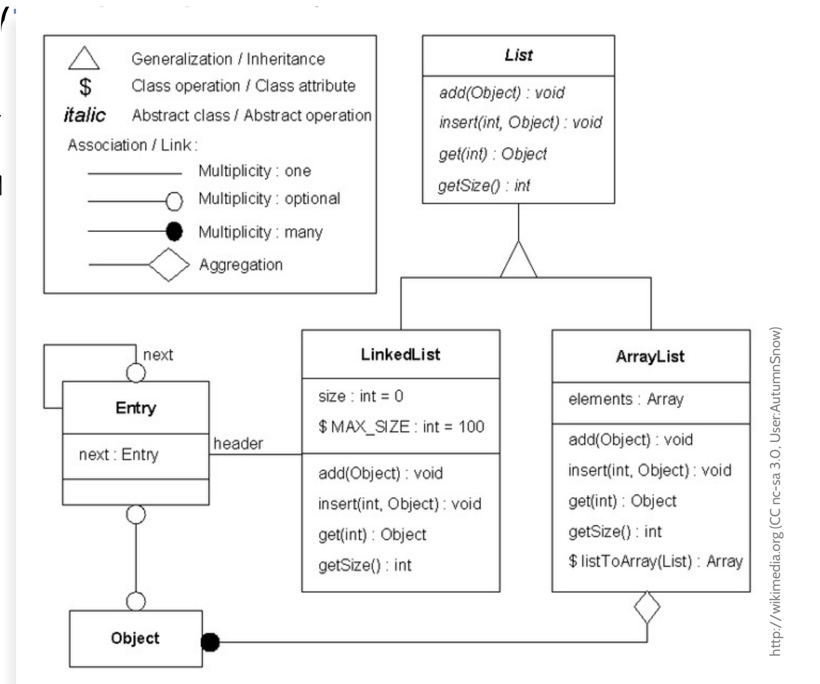
- Boxes/lines and finite automata are used to visualise software **for ages**.

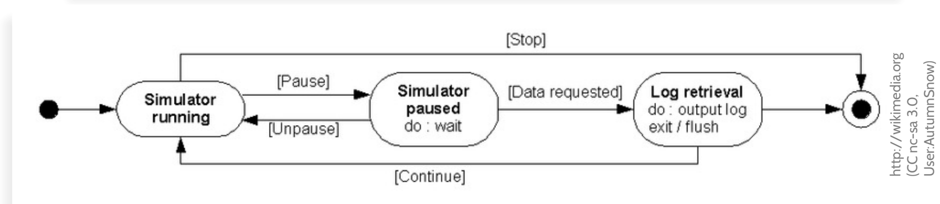
- **1970's**, **Software Crisis**™
  – Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**

- Mid **1980**'s: **Statecharts** (Harel, 1987), **StateMate**™ (

- Early **1990's**, advent of **Object-Oriented**-Analysis/
  – Inflation of notations and methods, most promin

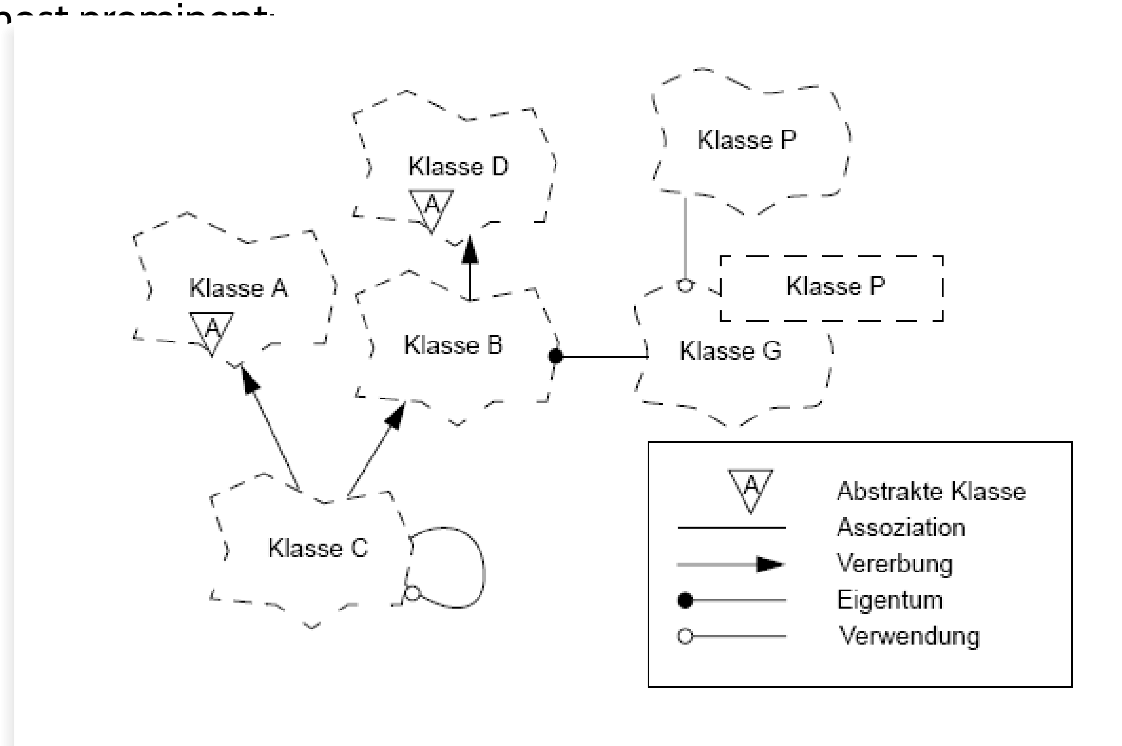
  - **Object-Modeling Technique** (OMT)
    (Rumbaugh et al., 1990)





A Brief History of the Unified Modelling Language (UML)

- Boxes/lines and finite automata are used to visualise software **for ages**.
- **1970's**, **Software Crisis**™
  – Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**
- Mid **1980**'s: **Statecharts** (Harel, 1987), **StateMate**™ (Harel et al., 1990)
- Early **1990's**, advent of **Object-Oriented**-Analysis/Design/Programming
  – Inflation of notations and methods, most prominent:
  - **Object-Modeling Technique** (OMT)
    (Rumbaugh et al., 1990)
  - **Booch Method and Notation**
    (Booch, 1993)



http://wikimedia.org (Public domain, Johannes Fasolt)

- Boxes/lines and finite automata are used to visualise software **for ages**.

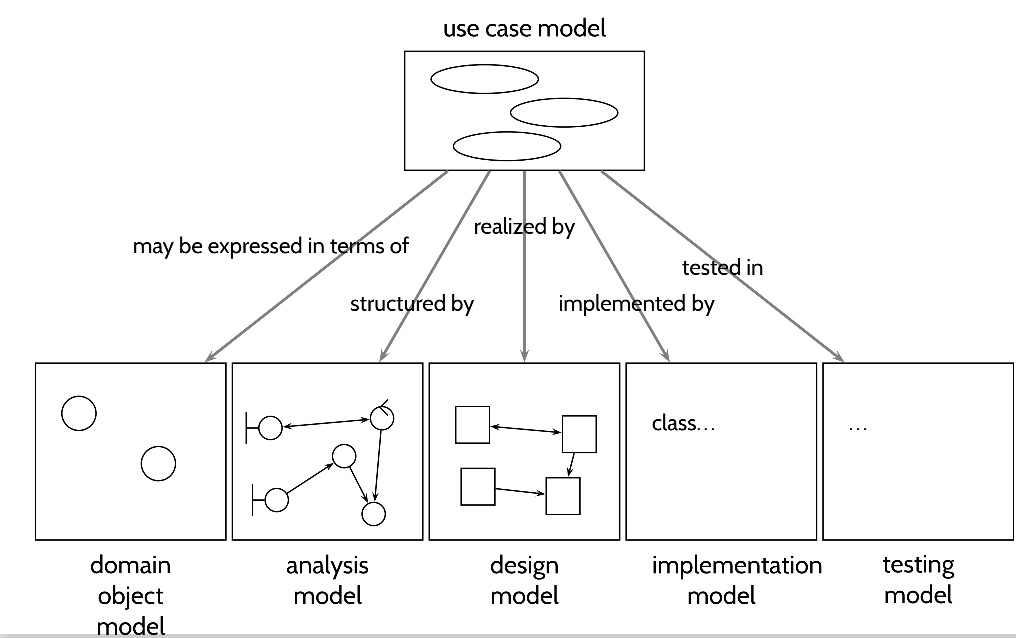
- **1970's**, **Software Crisis**™
  – Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**

- Mid **1980**'s: **Statecharts** (Harel, 1987), **StateMate**™ (Harel et al., 1990)

- Early **1990's**, advent of **Object-Oriented**-Analysis/Design/Programming
  – Inflation of notations and methods, most prominent:

  - **Object-Modeling Technique** (OMT) (Rumbaugh et al., 1990)
  - **Booch Method and Notation** (Booch, 1993)
  - **Object-Oriented Software Engineering** (OOSE) (Jacobson et al., 1992)

  Each "persuasion" selling books, tools, seminars...



A Brief History of the Unified Modelling Language (UML)

- Boxes/lines and finite automata are used to visualise software **for ages**.
- **1970's**, **Software Crisis**™
  – Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**
- Mid **1980**'s: **Statecharts** (Harel, 1987), **StateMate**™ (Harel et al., 1990)
- Early **1990's**, advent of **Object-Oriented**-Analysis/Design/Programming
  – Inflation of notations and methods, most prominent:
  - **Object-Modeling Technique** (OMT)
    (Rumbaugh et al., 1990)
  - **Booch Method and Notation**
    (Booch, 1993)
  - **Object-Oriented Software Engineering** (OOSE)
    (Jacobson et al., 1992)

  Each "persuasion" selling books, tools, seminars...
- Late **1990's**: joint effort of "the three amigos" **UML 0.x** and **1.x**
  Standards published by **Object Management Group** (OMG), "*international, open membership, not-for-profit* ***computer industry*** *consortium*". Much criticised for lack of formality.
- Since **2005**: **UML 2.x**, split into infra- and superstructure documents.

UML Overview (OMG, 2007, 684)

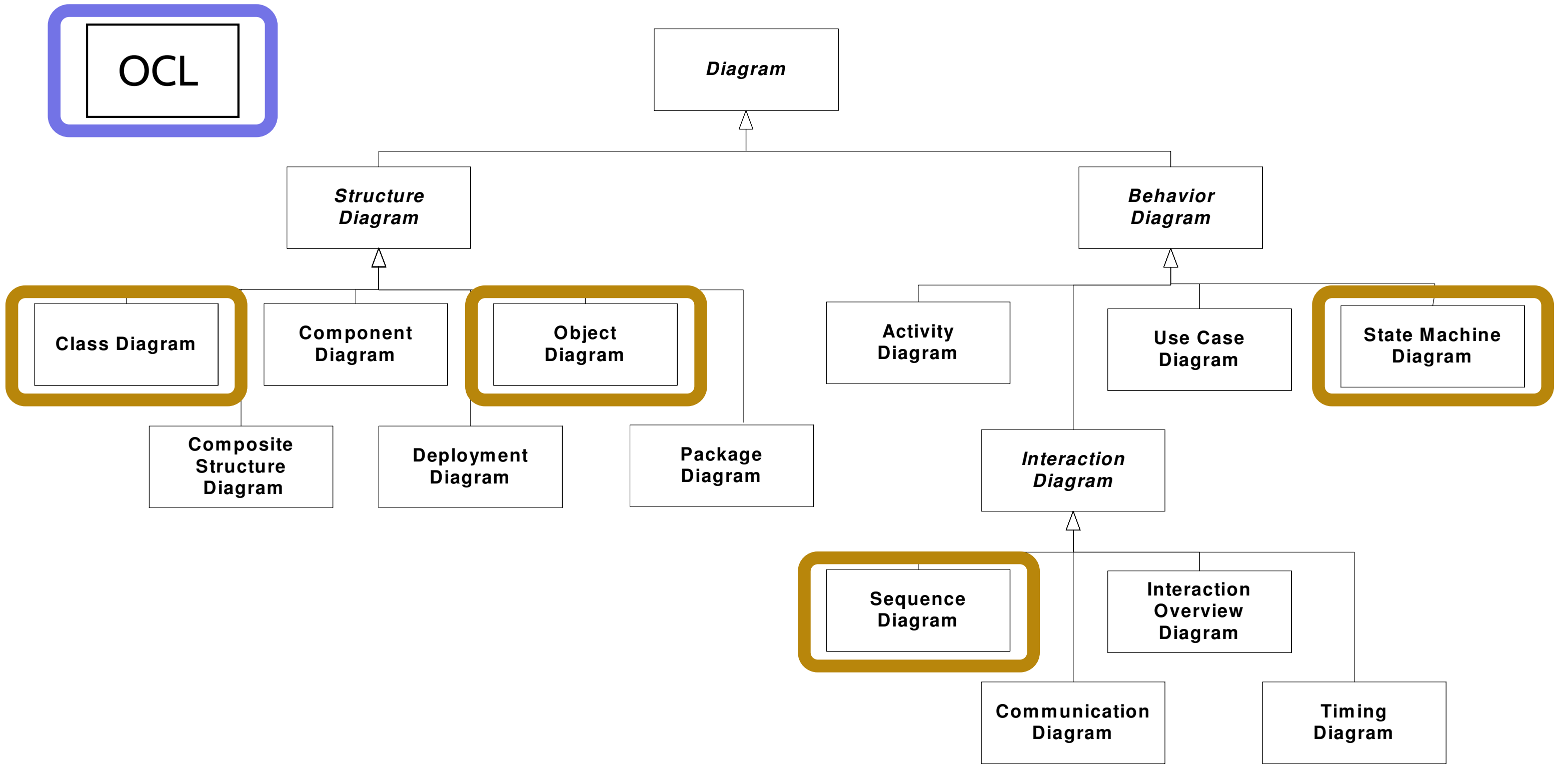


Figure A.5 - The taxonomy of structure and behavior diagram

Dobing and Parsons (2006)