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 |
| VL 11 | Software Modelling I |
|   | (i) views and viewpoints, the 4+1 view |
|   | (ii) model-driven/-based software engineering |
|   | (iii) Modelling structure |
|     | a) (simplified) class diagrams |
|     | b) (simplified) object diagrams |
| VL 12 | c) (simplified) object constraint logic (OCL) |
|       | d) Unified Modelling Language (UML) |

| VL 13 | Principles of Design |
|       | (i) modularity, separation of concerns |
|       | (ii) information hiding and data encapsulation |
|       | (iii) abstract data types, object orientation |
|       | (iv) Design Patterns |

| VL 14 | Software Modelling II |
|       | (i) Modelling behaviour |
|       | a) communicating finite automata |
|       | b) Uppaal query language |
|       | c) basic state-machines |
|       | d) an outlook on hierarchical state-machines |
Partial vs. Complete Object Diagrams

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

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

\[\mapsto \]

What about the other way round...?

- Object diagrams can be partial, e.g.

\[
\begin{array}{c}
\text{Object Diagram 1} \\
\text{Object Diagram 2}
\end{array}
\]

\[\mapsto \]

→ we may omit information.

- Is the following object diagram partial or complete?

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

- 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

\[
\begin{array}{c}
\text{obj: C} \\
p = 0 \\
\end{array}
\quad \begin{array}{c}
\text{p} \\
n = 0 \\
\end{array}
\quad \begin{array}{c}
D \\
x = 23 \\
\end{array}
\]

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

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:

  ![Diagram](image)

  Note: formally, it is a proper system state.

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

\[ \forall d_1 \in \text{allInstances}_A \bullet \forall d_2 \in \text{allInstances}_A \bullet c(d_1) = c(d_2) \implies a(d_1) = a(d_2) \]
• **Example:** for all $C$-instances, $x$ should never have the value $27$.

$$\forall c \in allInstances_C \cdot x(c) \neq 27$$

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

\[
F ::= c : \tau_C \quad (\#)
\|
allInstances_C : 2^{\tau_C}
\|
v(F) : \tau_C \rightarrow \tau_D, \quad \text{if } v : \tau \in atr(C), \ r \in \mathcal{F} \ (\#)
\|
v(F) : \tau_C \rightarrow \tau_D, \quad \text{if } v : D_{0,1} \in atr(C)
\|
f(F_1, \ldots, F_n) : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad \text{if } f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau
\|
\forall c \in F_1 \bullet F_2 : \tau_C \times 2^{\tau_C} \times B \rightarrow B
\]

• The formula above in **prefix normal form**: $$\forall c \in allInstances_C \cdot \neg (x(c), 27) \quad \begin{array}{c}
\neg x(c) \\
27
\end{array}$$

### Semantics

#### Proto-OCL Types:

- $I[T_C] = \mathcal{P}(C) \cup \{\bot\}$, $I[T_{\bot}] = \mathcal{P}(\tau) \cup \{\bot\}$
- $I[B_{\bot}] = \{\text{true, false}\} \cup \{\bot\}$

#### Functions:

- We assume $f_2$ given for each function symbol $f$ (→ in a minute).

#### Proto-OCL Semantics (interpretation function):

- $I[c]((\sigma, \beta)) = \beta(c)$ (assuming $\beta$ is a type-consistent valuation of the logical variables).

- $I[allInstances_C]((\sigma, \beta)) = \text{dom}(\sigma) \cap \mathcal{P}(C)$.

- $I[v(F)]((\sigma, \beta)) = \begin{cases} I[v(F)]((\sigma, \beta)) & \text{if } I[F]((\sigma, \beta)) \in \text{dom}(\sigma) \\ \bot & \text{otherwise} \end{cases}$

- $I[f(F_1, \ldots, F_n)]((\sigma, \beta)) = \begin{cases} I[f(F_1)]((\sigma, \beta)), \ldots, I[f(F_n)]((\sigma, \beta)) & \text{if } f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \\ \bot & \text{otherwise} \end{cases}$
Proto-OCL is a three-valued logic: a formula evaluates to true, false, or ⊥.

Example: $\land I(\cdot, \cdot): \{true, false, \bot\} \times \{true, false, \bot\} \rightarrow \{true, false, \bot\}$ is defined as follows:

<table>
<thead>
<tr>
<th>$\land I(x_1, x_2)$</th>
<th>true</th>
<th>false</th>
<th>⊥</th>
<th>true</th>
<th>false</th>
<th>⊥</th>
<th>false</th>
<th>⊥</th>
<th>false</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>true</td>
<td>false</td>
<td>⊥</td>
<td>true</td>
<td>false</td>
<td>⊥</td>
<td>false</td>
<td>⊥</td>
<td>false</td>
<td>⊥</td>
</tr>
<tr>
<td>$x_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We assume common logical connectives $\neg, \land, \lor, \ldots$ with canonical 3-valued interpretation.

Example: $+ I(\cdot, \cdot): (\mathbb{Z} \cup \{\bot\}) \times (\mathbb{Z} \cup \{\bot\}) \rightarrow \mathbb{Z} \cup \{\bot\}$

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

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

And we assume the special unary function symbol isUndefined:

$$isUndefined_I(x) = \begin{cases} true, & \text{if } x = \bot \\ false, & \text{otherwise} \end{cases}$$

isUndefined$_I$ is definite: it never yields ⊥.

Example: Evaluate Formula for System State

$$\sigma: 1_C : C \in \Sigma^\sigma \quad \exists!_\sigma$$  

$\forall c \in \text{allInstances}_C: x(c) \neq 27$

Recall prefix notation: $\forall c \in \text{allInstances}_C: \lnot (x(c), 27)$

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

Example:

$I[\forall c \in \text{allInstances}_C: \lnot (x(c), 27)](\sigma, \emptyset) = true$, because...

$I[\lnot (x(c), 27)](\sigma, \beta) = \lnot (I x(c)[(\sigma, \beta), I[27]](\sigma, \beta))$

$= \lnot (I x(c)[(\sigma, \beta), I[27]](\sigma, \beta))$

$= \lnot (\sigma( I x(c)[(\sigma, \beta), I[27]](\sigma, \beta)))$

$= \lnot (\sigma( I x(c)[(\sigma, \beta), I[27]](\sigma, \beta)))$

$= \lnot (\sigma( I x(c)[(\sigma, \beta), I[27]](\sigma, \beta)))$

$= \lnot (13, 27) = true$  

...and 1$C$ is the only $C$-object in $\sigma$, $I[\text{allInstances}_C](\sigma, \emptyset) = \{1_C\}$.  


More Interesting Example

\[ \forall c : \text{allInstances}_C \cdot x(n(c)) \neq 27 \]

- Similar to the previous slide, we need the value of
  \[ I[x(n(c))](\sigma, \beta) = \{ c \mapsto 1_C \} \]

- \( I[c](\sigma, \beta) = \beta(c) = 1_C \)

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

- \( I[x(n(c))](\sigma, \beta) = \bot \) since \( I[n(c)](\sigma, \beta) = \bot \) by rule

by the following rule:

\[ I[v(F)](\sigma, \beta) = \begin{cases} \sigma( I[c](\sigma, \beta))(v) & \text{if } I[F](\sigma, \beta) \in \text{dom}(\sigma) \text{ and } \sigma(I[F](\sigma, \beta))(v) = \{ u' \} \\ \bot & \text{otherwise} \end{cases} \quad (\text{if } v : C_0,1) \]
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”)
Where To Put OCL Constraints?

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

\[
\begin{array}{c}
\text{text}
\end{array}
\]

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

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

\[
\text{OCL: } F
\]

- **Conventions:**

\[
\forall \text{self } \in \text{allInstances}_C \cdot F
\]
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
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 $\llbracket \cdot \rrbracket : \Sigma \mapsto \{S\}$ is called interpretation of $S$.

- The set of states $\Sigma$ could be the set of system states as defined by a class diagram, e.g.

$\Sigma := \Sigma_D$

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

$\llbracket S \rrbracket$

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

$F = \forall c \in \text{allInstances}_C : x(c) < 4$

then $S$ does not satisfy the requirement.

More General: Software vs. Proto-OCL

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

$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in \llbracket S \rrbracket$

and all $i \in \mathbb{N}_0$, $\mathcal{I}[F](\sigma_i, 0) = \text{true}$.

- We say $\llbracket S \rrbracket$ does not satisfy $F$, denoted by $\llbracket S \rrbracket \not\models F$, if and only if there exists

$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in \llbracket S \rrbracket$ and $i \in \mathbb{N}_0$, such that $\mathcal{I}[F](\sigma_i, 0) = \text{false}$.

- Note: $\neg(\llbracket S \rrbracket \not\models F)$ does not imply $\llbracket S \rrbracket \models F$. 

21/66
Class Diagrams can be used to 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 $\sigma \in \Sigma_{\mathcal{D}}$
can be visualised by an object diagram.

Proto-OCL constraints can be evaluated on system states.

A software over $\Sigma_{\mathcal{D}}$ 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

System consists of 1 or more Component
has Component Interface
Software System Software Component
may be a Module

Software Architecture
is the result of Design
is described by 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.

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”:
Principles of (Architectural) Design
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. Modularity**

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

*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.

- 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.
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 (→ 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
• 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**

• 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.
**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.**
• **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.

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

---

**Layered Architectures: Discussion**

- **Advantages**:
  - protocol-based: only neighbouring 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.
**Example: Pipe-Filter**

**Example: Compiler**

- **Sourcecode**
- **ASCII**
- **Tokens**
- **AST**
- **dAST**
- **Objectcode**
- **Errormessages**
**Example: Compiler**

![Diagram of the compiler process]

**Example: UNIX Pipes**

```plaintext
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
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**
• 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

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:
• 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/Design/Programming
  – Inflation of notations and methods, most prominent:
    • Object-Modeling Technique (OMT) (Rumbaugh et al., 1990)
    • Booch Method and Notation (Booch, 1993)
• 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.
Figure A.5 - The taxonomy of structure and behavior diagram

Dobing and Parsons (2006)