

Libraries and Frameworks

Layered Architectures, Pipe-Filter, Model-View-Controller.

Architecture Patterns

Principles of (Good) Design

Proto-OCL vs. Software

Proto-OCL vs. OCL.

Special Case: Anonymous Objects

Partial vs. Complete Object Diagrams

Modelling behaviour

Modelling structure

VL 13

VL 12

VL 11

(ii) model-driven/-based software engineering

Software Modelling I

(ii) information hiding and data encapsulation

Unified Modelling Language (UML)

Modularisation & Design Patterns

...
Example: Evaluate Formula for System State

\[ F \]
F • allInstances ∈ ∀ . . . stands for

Conventions:
• implies

name="Lobby"
Location, content is
inv: Meeting
context

and comments can principally be
text

participants
name : String
TeamMember
note

• Where To Put OCL Constraints?
(from lecture "Softwaretechnik 2008")

OMG Available SpecificationVersion 2.0
Object Constraint Language

text

More Interesting Example

More Interesting Example

DIFFERENCE CONSTRAINT LANGUAGE (OCL)

MCs in Constraint (left) and Model (right)

 modelo modelo

modelo modelo

modelo modelo

MC: methods – model, entities modeled (E) – methods

modelo modelo

modelo modelo

modelo modelo

MC: methods – model, entities modeled (E) – methods

DIFFERENCE CONSTRAINT LANGUAGE (OCL)
Layered Architectures, Pipe-Filter, Model-View-Controller. If a requirement is formalised by the Proto-OCL constraint \( \alpha = S \), denoted by \( \Sigma \in \alpha \), defines a set of system states \( N \in i \), \( \alpha \rightarrow N \), and all constraints can be evaluated on abstract data types, object orientation can be visualised as a set of Class Diagrams, and a system state space of a software system can be defined. Let \( S \) and \( F \) be the set of system states and any set of states that \( F \) can be evaluated on. If \( F \) could be the set of system states, the interpretation could be the set of states that \( F \) can be evaluated on, and the function \( F \) could be any set of functions that \( F \) could be evaluated on. The (possibly partial) function \( T \) from requirement to software is called Proto-OCL vs. Software. Proto-OCL vs. OCL can be more general, where Proto-OCL syntax, semantics, and principles of (good) design could be the set of a software system states and any set of states that \( F \) could be evaluated on.
Once Again, Please

System Software System
Component Software Component Module Interface Component Interface consists of 1 or more
is a is a may be a has is an Software Architecture Architecture Architectural Description Design — 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”:

req. design design arch. designer modulespec. impl. impl. code programmer implementation

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 — 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.
• Functional modules

• Data object modules

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

In an object-oriented design, • classes are

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

Example

– Good modules implement a user-defined data type in form of an abstract data type (ADT)

Knowing how a component works internally may enable more efficient operation.

– Examples

: modules encapsulating global configuration data, databases

: most prominently graphical interaction

: different networking technology

: sometimes, functional aspects (features) like printing are realised as separate components,

• Information hiding → what is hidden is information which other components need not know.

: what is hidden is information which other components need to know principle

: needs to know principle

The " is called

As a software development technique in which each module’s inter-

faces 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 in-

face specification.

As long as the visible behaviour stays the same (e.g. the employed sorting algorithm).

Hidden solutions may be

in a communication protocol (examples later).

Components can be verified / validated in isolation.

IOW: other components cannot (examples later).

as long as the visible behaviour stays the same (e.g. the employed sorting algorithm).

by now, we only discussed the

Advantages

• Information hiding

• Advantages

SIMILAR DIRECTIONS:

• Separation of Concerns

• Advantages

3.) Information Hiding

• Advantages

4.) Data Encapsulation

• Advantages

C. J. Nagl (1990)

IEEE 610.12 (1972), D. Parnas

A Classification of Modules
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)

• 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.

7. Application
6. Presentation
5. Session
4. Transport
3. Network
2. Data link
1. Physical

7. Application
6. Presentation
5. Session
4. Transport
3. Network
2. Data link
1. Physical

Data packets
Frames
Bits
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,
  - all components of layers further beneath.

GNOME etc. Applications

- GTK+
- GDK ATK
- Cairo GLib
- GIO Pango

Example: Layered Architectures Cont'd

7. Application
   6. Presentation
   5. Session
   4. Transport
   3. Network
   2. Data link
   1. Physical

desktop host

Application Server
(business) logic tier

- 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.

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.

Pipe-Filter

Example: Pipe-Filter

- Compiler
  - lexical analysis (lexer)
  - syntactical analysis (parser)
  - semantical analysis
  - code generation
  - ASCII Tokens
  - AST
dAST
Sourcecode
Objectcode
Errormessages
Example: Pipe-Filter

Example:
- Compiler
  - lexical analysis (lexer)
  - syntactical analysis (parser)
  - semantical analysis
  - code generation

ASCII

Tokens

AST
dAST

Sourcecode

Objectcode

Errormessages

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

• 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

https://commons.wikimedia.org/wiki/File:Maschinenleitstand_KWZ.jpg Der Genaue, CC-BY-SA-2.5
Libraries and Frameworks

- Strategy, Examples
- Design Patterns

Object-Oriented Construction Handbook - Developing Application-Oriented Software with the Tools and Materials Approach

The Object Constraint Language

Object-Oriented Modeling and Design

Layered Architectures, Pipe-Filter, Model-View-Controller.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules.


Architecture Patterns

Softwaretechnik: Methodisches Programmieren im Großen

Software Engineering

JHotDraw (2007).

Object-Oriented Software Engineering - A Use Case Driven Approach


IEEE Standard Glossary of Software Engineering Terminology

UML — Analysis/Design/Programming

Proto-OCL vs. Software
— Idea: learn from engineering disciplines to handle growing complexity.

StateMate: A working environment for the development of complex reactive systems.

Statecharts ’s:
1980, 1990

Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams

software Crisis
Mid 1970’s

UML — Inflation of notations and methods, most prominent:
• modularity, separation of concerns

— Idea: learn from engineering disciplines to handle growing complexity.

References
A Brief History of the Unified Modelling Language (UML)

- In the 1970s, the Software Crisis led to a need for better tools to manage software projects.
- For ages, boxes/lines and finite automata were used to visualize software.
- The idea was to learn from engineering disciplines to handle growing complexity.

• Modelling languages: Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams.

- In the 1980s, the Object-Oriented Methodology (OOSE) was introduced by Jacobson et al. (1992) and Booch (1993).
- Rumbaugh et al. (1990) introduced the Object-Modeling Technique (OMT).
- Harel et al. (1987) developed Statecharts, and Harel (1987) introduced StateMate.

- In the early 1990s, the object-oriented paradigm gained traction with the advent of UML 1.x and the joint effort of “the three amigos”.

- UML 2.x was much criticized for lack of formality.
- The UML Notation and Methodology is not-for-profit, structured by the Object Management Group (OMG).

- The taxonomy of structure and behavior diagrams includes:
  - Use Case
  - State Machine
  - Class Diagram
  - Sequence Diagram
  - Deployment Diagram
  - Activity Diagram
  - Interaction Diagram
  - Structure Diagram
  - Behavior Diagram

- Early versions of UML were implemented by vendors, leading to a proliferation of methods and notations.
- In 2006, the OMG published UML 2.0, consolidating the various methods under a single standard.

- Each “persuasion” is selling books, tools, seminars, and certifications for ages.

- The success of UML led to its adoption by the computer industry and became a de facto standard for object-oriented modeling.

- Figure A.5 displays the taxonomy of structure and behavior diagram types.

- OCL (Object Constraint Language) was introduced in 1997 to provide a formal language for expressing constraints on models.

- For further details, visit [http://wikimedia.org](http://wikimedia.org) (Public domain, User:AutumnSnow).