Softwaretechnik / Software-Engineering

Lecture 11: Architecture & Design

2015-06-22

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents of the Block "Design"

(i) **Introduction and Vocabulary**

(ii) **Principles of Design**

a) modularity
b) separation of concerns
c) information hiding and data encapsulation
d) abstract data types, object orientation

(iii) **Software Modelling**

a) views and viewpoints, the 4+1 view
b) model-driven/based software engineering
c) Unified Modelling Language (UML)
d) **modelling structure**

1. (simplified) class diagrams
2. (simplified) object diagrams
3. (simplified) object constraint logic (OCL)

e) **modelling behaviour**

1. communicating finite automata
2. Uppaal query language
3. basic state-machines
4. an outlook on hierarchical state-machines

(iv) **Design Patterns**

Introduction	L 1:	20.4., Mo
	T 1:	23.4., Do
Development Process, Metrics	L 2:	27.4., Mo
	L 3:	30.4., Do
	L 4:	4.5., Mo
	T 2:	7.5., Do
	L 5:	11.5., Mo
Requirements Engineering	-	14.5., Do
	L 6:	18.5., Mo
	L 7:	21.5., Do
	-	25.5., Mo
	-	28.5., Do
	T 3:	1.6., Mo
	-	4.6., Do
	L 8:	8.6., Mo
	L 9:	11.6., Do
	L 10:	15.6., Mo
	T 4:	18.6., Do
Architecture & Design, Software Modelling	L 11:	22.6., Mo
	L 12:	25.6., Do
	L 13:	29.6., Mo
	T 5:	2.7., Do
	L 14:	6.7., Mo
Quality Assurance	L 15:	9.7., Do
	L 16:	13.7., Mo
Invited Talks	L 17:	16.7., Do
	T 6:	20.7., Mo
Wrap-Up	L 18:	23.7., Do

Contents & Goals

Last Lecture:

- Requirements Engineering completed

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What's the definition of 'design'?
  - What are the basic principles of software design?
  - What is a view and viewpoint?
  - What is the signature of this class diagram?
  - Which system states does this class diagram denote?

- **Content:**
  - Introduction and vocabulary
  - Principles of design
  - Software modelling
  - Modelling structure

Design Modelling & Analysis: Introduction

Goals and Relevance of Design

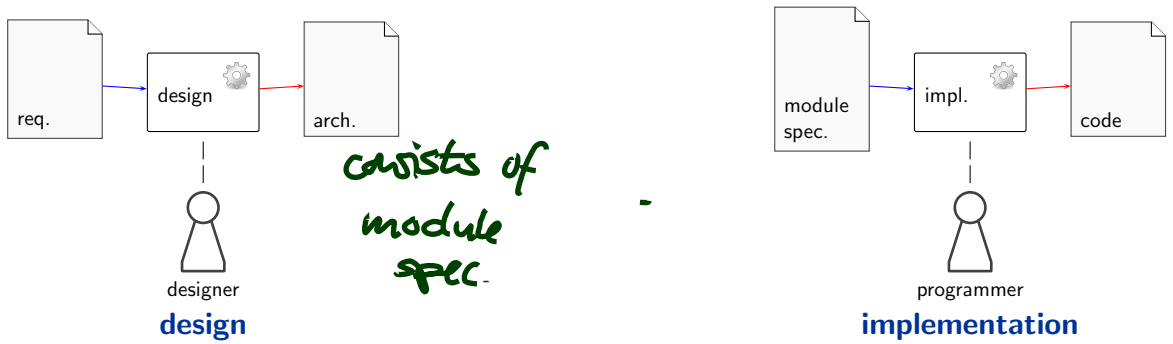
(i) **structuring** the system into **manageable** units (yields software architecture),

(ii) **concretising** the approach to realise the required software,

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

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

Oversimplified process model:



IEEE
Std 610.12-1990
(Revision and redesignation of
IEEE Std 792-1983)

IEEE Standard Glossary of Software Engineering Terminology

Sponsor

**Standards Coordinating Committee
of the
Computer Society of the IEEE**

Approved September 28, 1990

IEEE Standards Board

Abstract: IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, identifies terms currently in use in the field of Software Engineering. Standard definitions for those terms are established.
Keywords: Software engineering; glossary; terminology; definitions; dictionary

ISBN 1-55937-067-X

Vocabulary

system — A collection of components organized to accomplish a specific function or set of functions.

IEEE 1471 (2000)

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

(Ludewig and Lichter, 2013)

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

IEEE 610.12 (1990)

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

(Taylor et al., 2010)

Vocabulary Cont'd

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

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

(Ludewig and Lichter, 2013)

interface — A boundary across which two independent entities meet and interact or communicate with each other. **(Bachmann et al., 2002)**

interface (of component) — The boundary between two communicating components. The interface of a component provides the services of the component to the component's environment and/or requires services needed by the component from the requirement. **(Ludewig and Lichter, 2013)**

Even More Vocabulary

design — (1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.
(2) The result of the process in (1).

IEEE 610.12 (1990)

architecture — The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

IEEE 1471 (2000)

software architecture — The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them.

(Bass et al., 2003)

architectural description — A model - document, product or other artifact - to communicate and record a system's architecture. An architectural description conveys a set of views each of which depicts the system by describing domain concerns.

(Ellis et al., 1996)

Principles of (Architectural) Design

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.

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: the logical flow of (logical) messages in a communication protocol (**functional**) vs. the 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

Information Hiding and Data Encapsulation

- By now, we strictly speaking only discussed the **grouping** of data and operations.
- One should also consider **accessibility**, i.e. which component "sees" or has access to which data and operations of which other component.
- The "**need to know principle**" is known as **information hiding** in software engineering. (Parnas, 1972)

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

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

  In other words: **information hiding** is about **making explicit** for one component what other components may use of this component.
- **Advantages**:
  - Solutions may be changed without other components noticing as long as the behaviour visible via the interface 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 validated in isolation.

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 read that place when the intermediate results is needed in a different context.

    → maintenance nightmare; if needed in another context, an operation should be offered.

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

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`.

Example

(i) information hiding and data encapsulation not enforced,
(ii) negative effects when requirements change,
(iii) information hiding and data encapsulation by modules,
(iv) abstract data types,
(v) object oriented without information hiding and data encapsulation,
(vi) and with information hiding and data encapsulation,

Example: Collecting Names

- **Task**: store a list of names in N.
- **Operations**:
  - $insert($ `string` n $)$;
    - **pre-condition**:
      $N = n_0, \ldots, n_i, n_{i+1}, \ldots, n_{m-1}$, $m \in \mathbb{N}_0$, $\forall\, 0 \leq j < m \bullet n_j <_{lex} n_{j+1}$
    - **post-condition**:
      $N = n_0, \ldots, n_i, n, n_{i+1}, \ldots, n_{m-1}$ if $n_i <_{lex} n <_{lex} n_{i+1}$, $N = old(N)$ otherwise.
  - $remove($ `int` i $)$;
    - **pre-condition**: $N = n_0, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_{m-1}$, $m \in \mathbb{N}_0$, $0 \leq i < m$,
    - **post-condition**: $N = n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_{m-1}$.
  - $get($ `int` i $)$: `string`;
    - **pre-condition**: $N = n_0, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_{m-1}$, $m \in \mathbb{N}_0$, $0 \leq i < m$,
    - **post-condition**: $N = old(N)$, $retval = n_i$.
  - $dump()$;
    - **pre-condition**: $N = n_0, \ldots, n_{m-1}$, $m \in \mathbb{N}_0$,
    - **post-condition**: $N = old(N)$.
    - **side-effect**: $n_0, \ldots, n_{m-1}$ printed to standard output in this order.

Implementations: Plain

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

std::vector<std::string> names;

void insert( std::string n ) {

  std::vector<std::string>
    ::iterator it =
      lower_bound( names.begin(),
                   names.end(), n );

  if (it == names.end() || *it != n)
    names.insert( it, n );
}

void remove( int i ) {
  names.erase( names.begin() + i );
}

std::string get( int i ) {
  return names[i];
}
```

```cpp
int main() {

  insert( "Berger" );
  insert( "Schulz" );
  insert( "Neumann" );
  insert( "Meyer" );
  insert( "Wernersen" );
  insert( "Neumann" );

  dump();

  remove( 1 );
  insert( "Mayer" );

  dump();

  names[2] = "Naumann";

  dump();

  return 0;
}
```

```
1  Berger
2  Meyer
3  Neumann
4  Schulz
5  Wernersen
6
7  Berger
8  Mayer
9  Neumann
10 Schulz
11 Wernersen
12
13 Berger
14 Mayer
15 Naumann
16 Schulz
17 Wernersen
```

Implementations: Multi-List

```
1  std::vector<int> count;
2  std::vector<std::string> names;
3
4  void insert( std::string n ) {
5
6    std::vector<std::string>::iterator
7      it = lower_bound( names.begin(),
8                        names.end(), n );
9
10   if (it == names.end()) {
11     names.insert( it, n );
12     count.insert( count.end(), 1 );
13   } else {
14     if (*it != n) {
15       count.insert( count.begin() +
16                     (it - names.begin()),
17                     1 );
18       names.insert( it, n );
19     } else {
20       ++(*( count.begin() +
21             (it - names.begin()) ));
22     }
23   }
24 }
25
26 void remove( int i ) {
27   if (--count[i] == 0) {
28     names.erase( names.begin() + i );
29     count.erase( count.begin() + i );
30   }
31 }
32
33 std::string get( int i ) {
34   return names[i];
35 }
```

increment counter

```
1  int main() {
2
3    insert( "Berger" );
4    insert( "Schulz" );
5    insert( "Neumann" );
6    insert( "Meyer" );
7    insert( "Wernersen" );
8    insert( "Neumann" );
9
10   dump();
11
12   remove( 1 );
13   insert( "Mayer" );
14
15   dump();
16
17   names[2] = "Naumann";
18
19   dump();
20
21   return 0;
22 }
```

```
1  Berger:1
2  Meyer:1
3  Neumann:2
4  Schulz:1
5  Wernersen:1
6
7  Berger:1
8  Mayer:1
9  Neumann:2
10 Schulz:1
11 Wernersen:1
12
13 Berger:1
14 Mayer:1
15 Naumann:2
16 Schulz:1
17 Wernersen:1
```

Data Encapsulation + Information Hiding

header

```
1  #include <string>
2
3  void dump();
4
5  void insert( std::string n );
6
7  void remove( int i );
8
9  std::string get( int i );
```

source

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  #include "mod_deih.h"
6
7  std::vector<int> count;
8  std::vector<std::string> names;
9
10 void insert( std::string n ) {
11 }
12
13 void remove( int i ) {
14   if (--count[i] == 0) {
15     names.erase( names.begin() + i );
16     count.erase( count.begin() + i );
17   }
18 }
19
20 std::string get( int i ) {
```

not in Header

```
1  #include "mod_deih.h"
2
3  int main() {
4
5    insert( "Berger" );
6    insert( "Schulz" );
7    insert( "Neumann" );
8    insert( "Meyer" );
9    insert( "Wernersen" );
10   insert( "Neumann" );
11
12   dump();
13
14   remove( 1 );
15   insert( "Mayer" );
16
17   dump();
18
19 #ifndef AVOID_PROBLEM
20   names[2] = "Naumann";
21 #else
22   remove( 2 );
23   insert( "Naumann" );
24 #endif
25   dump();
26
27   return 0;
28 }
```

```
1  mod_deih_main.cpp: In function int main() :
2  mod_deih_main.cpp:20:3: error: names was not declared in this scope
```

Data Encapsulation + Information Hiding

header

```
#include <string>

void dump();

void insert( std::string n );

void remove( int i );

std::string get( int i );
```

source

```
#include <algorithm>
#include <iostream>
#include <vector>

#include "mod_deih.h"

std::vector<int> count;
std::vector<std::string> names;

void insert( std::string n ) {
}

void remove( int i ) {
  if (--count[i] == 0) {
    names.erase( names.begin() + i );
    count.erase( count.begin() + i );
  }
}

std::string get( int i ) {
  return names[i];
}
```

```
#include "mod_deih.h"

int main() {

  insert( "Berger" );
  insert( "Schulz" );
  insert( "Neumann" );
  insert( "Meyer" );
  insert( "Wernersen" );
  insert( "Neumann" );

  dump();

  remove( 1 );
  insert( "Mayer" );

  dump();

#ifndef AVOID_PROBLEM
  names[2] = "Naumann";
#else
  remove( 2 );
  insert( "Naumann" );
#endif
  dump();

  return 0;
}
```

```
Berger:1
Meyer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Naumann:1
Neumann:1
Schulz:1
Wernersen:1
```

Abstract Data Type

header

```
1  #include <string>
2
3  typedef void* Names;
4
5  Names new_Names();
6
7  void dump( Names names );
8
9  void insert( Names names, std::string n );
10
11 void remove( Names names, int i );
12
13 std::string get( Names names, int i );
```

get: Names x int → string

source

```
1  #include "mod_adt.h"
2
3  typedef struct {
4    std::vector<int> count;
5    std::vector<std::string> names;
6  } implNames;
7
8  Names new_Names() {
9    return new implNames;
10 }
11
12 void insert( Names names, std::string n ) {
13   implNames* in = (implNames*)names;
14
15   std::vector<std::string>::iterator
16     it = lower_bound( in->names.begin(),
17                       in->names.end(), n );
18
19   if (it == in->names.end()) {
20     in->names.insert( it, n );
21     in->count.insert( in->count.end(), 1 );
```

```
1  #include "mod_adt.h"
2
3  int main() {
4
5    Names names = new_Names();
6
7    insert( names, "Berger" );
8    insert( names, "Schulz" );
9    insert( names, "Neumann" );
10   insert( names, "Meyer" );
11   insert( names, "Wernersen" );
12   insert( names, "Neumann" );
13
14   dump( names );
15
16   remove( names, 1 );
17   insert( names, "Mayer" );
18
19   dump( names );
20
21 #ifndef AVOID_PROBLEM
22   names[2] = "Naumann";
23 #else
24   remove( names, 2 );
25   insert( names, "Naumann" );
26 #endif
27   dump( names );
28
29   return 0;
30 }
```

```
1  mod_adt_main.cpp: In function int main() :
2  mod_adt_main.cpp:22:10: warning: pointer of type void * used in arithmetic [-Wpointer-arith]
3  mod_adt_main.cpp:22:10: error: Names {aka void*} is not a pointer-to-object type
```

Abstract Data Type

header

```
#include <string>

typedef void* Names;

Names new_Names();

void dump( Names names );

void insert( Names names, std::string n );

void remove( Names names, int i );

std::string get( Names names, int i );
```

source

```
#include "mod_adt.h"

typedef struct {
  std::vector<int> count;
  std::vector<std::string> names;
} implNames;

Names new_Names() {
  return new implNames;
}

void insert( Names names, std::string n ) {
  implNames* in = (implNames*)names;

  std::vector<std::string>::iterator
    it = lower_bound( in->names.begin(),
                      in->names.end(), n );

  if (it == in->names.end()) {
    in->names.insert( it, n );
    in->count.insert( in->count.end(), 1 );
  } else {
    if (*it != n) {
      in->count.insert( in->count.begin() +
                        (it - in->names.begin()),
                        1 );
```

```
#include "mod_adt.h"

int main() {

  Names names = new_Names();

  insert( names, "Berger" );
  insert( names, "Schulz" );
  insert( names, "Neumann" );
  insert( names, "Meyer" );
  insert( names, "Wernersen" );
  insert( names, "Neumann" );

  dump( names );

  remove( names, 1 );
  insert( names, "Mayer" );

  dump( names );

#ifndef AVOID_PROBLEM
  names[2] = "Naumann";
#else
  remove( names, 2 );
  insert( names, "Naumann" );
#endif
  dump( names );

  return 0;
}
```

```
Berger:1
Meyer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Naumann:1
Neumann:1
Schulz:1
Wernersen:1
```

Object Oriented

header

```
#include <vector>
#include <string>

struct Names {

  std::vector<int> count;
  std::vector<std::string> names;

  Names();

  void dump();

  void insert( std::string n );

  void remove( int i );

  std::string get( int i );
};
```

source

```
#include "mod_oo.h"
}

void Names::insert( std::string n ) {

  std::vector<std::string>::iterator
    it = lower_bound( this->names.begin(),
                      this->names.end(), n );

  if (it == this->names.end()) {
    this->names.insert( it, n );
    this->count.insert( this->count.end(), 1 );
  } else {
    if (*it != n) {
      this->count.insert( this->count.begin() +
                          (it - this->names.begin()),
                          1 );
      this->names.insert( it, n );
    } else {
      ++(*( this->count.begin() +
            (it - this->names.begin()) ));
```

```
#include "mod_oo.h"

int main() {

  Names* names = new Names();

  names->insert( "Berger" );
  names->insert( "Schulz" );
  names->insert( "Neumann" );
  names->insert( "Meyer" );
  names->insert( "Wernersen" );
  names->insert( "Neumann" );

  names->dump();

  names->remove( 1 );
  names->insert( "Mayer" );

  names->dump();

  names->names[2] = "Naumann";

  names->dump();

  return 0;
}
```

```
Berger:1
Meyer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Naumann:2
Schulz:1
Wernersen:1
```

Object Oriented + Data Encapsulation / Information Hiding

header

```
1  #include <vector>
2  #include <string>
3
4  class Names {
5
6  private:
7    std::vector<int> count;
8    std::vector<std::string> names;
9
10 public:
11   Names();
12
13   void dump();
14
15   void insert( std::string n );
16
17   void remove( int i );
18
19   std::string get( int i );
20 };
```

source

```
1  #include "mod_oo_deih.h"
2
3  void Names::insert( std::string n ) {
4
5    std::vector<std::string>::iterator
6      it = lower_bound( names.begin(),
7                        names.end(), n );
8
9    if (it == names.end()) {
10     names.insert( it, n );
11     count.insert( count.end(), 1 );
18   } else {
```

```
1  #include "mod_oo_deih.h"
2
3  int main() {
4
5    Names* names = new Names();
6
7    names->insert( "Berger" );
8    names->insert( "Schulz" );
9    names->insert( "Neumann" );
10   names->insert( "Meyer" );
11   names->insert( "Wernersen" );
12   names->insert( "Neumann" );
13
14   names->dump();
15
16   names->remove( 1 );
17   names->insert( "Mayer" );
18
19   names->dump();
20
21 #ifndef AVOID_PROBLEM
22   names->names[2] = "Naumann";
23 #else
24   names->remove( 2 );
25   names->insert( "Naumann" );
26 #endif
27   names->dump();
28
29   return 0;
30 }
```

```
1  In file included from mod_oo_deih_main.cpp:1:0:
2  mod_oo_deih.h: In function   int main()   :
3  mod_oo_deih.h:9:28: error:   std::vector<std::basic_string<char> > Names::names   is private
4  mod_oo_deih_main.cpp:22:10: error: within this context
```

Object Oriented + Data Encapsulation / Information Hiding

header

```
#include <vector>
#include <string>

class Names {

private:
  std::vector<int> count;
  std::vector<std::string> names;

public:
  Names();

  void dump();

  void insert( std::string n );

  void remove( int i );

  std::string get( int i );
};
```

source

```
#include "mod_oo_deih.h"

void Names::insert( std::string n ) {

  std::vector<std::string>::iterator
    it = lower_bound( names.begin(),
                      names.end(), n );

  if (it == names.end()) {
    names.insert( it, n );
    count.insert( count.end(), 1 );
  } else {
    if (*it != n) {
      count.insert( count.begin() +
                      (it - names.begin()),
                      1 );
      names.insert( it, n );
    } else {
```

```
#include "mod_oo_deih.h"

int main() {

  Names* names = new Names();

  names->insert( "Berger" );
  names->insert( "Schulz" );
  names->insert( "Neumann" );
  names->insert( "Meyer" );
  names->insert( "Wernersen" );
  names->insert( "Neumann" );

  names->dump();

  names->remove( 1 );
  names->insert( "Mayer" );

  names->dump();

#ifndef AVOID_PROBLEM
  names->names[2] = "Naumann";
#else
  names->remove( 2 );
  names->insert( "Naumann" );
#endif
  names->dump();

  return 0;
}
```

```
Berger:1
Meyer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Neumann:2
Schulz:1
Wernersen:1

Berger:1
Mayer:1
Naumann:1
Neumann:1
Schulz:1
Wernersen:1
```

"Tell Them What You've Told Them"

(i) information hiding and data encapsulation **not enforced**,

(ii) **negative effects** when requirements change,

(iii) **information hiding** and **data encapsulation** by modules,

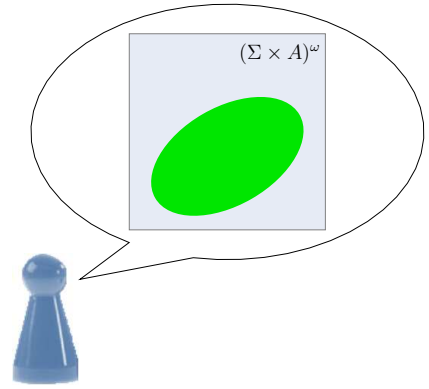
(iv) **abstract data types**,

(v) **object oriented without** information hiding and data encapsulation,

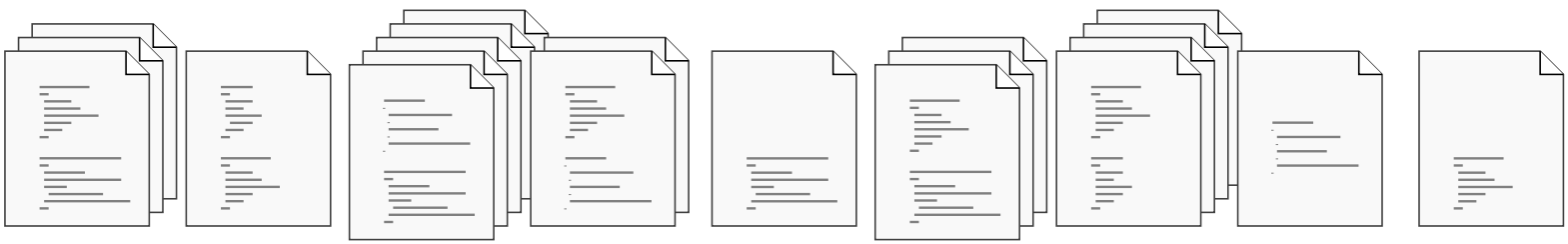
(vi) and **with** information hiding and data encapsulation,

Software Modelling

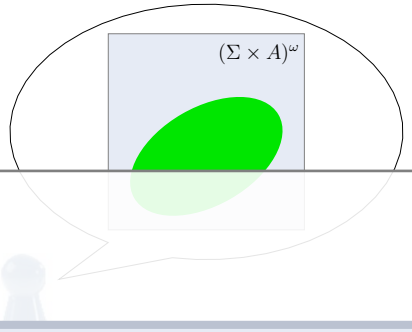
Recall: Model



Analyst



Recall: Model



Model

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

Definition. (Glinz, 2008, 425)
A **model** is a concrete or mental image (Abbild) of something or a concrete or mental archetype (Vorbild) for something.

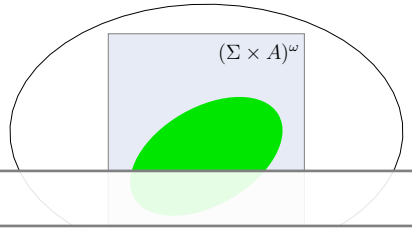
Three properties are constituent:

(i) the **image attribute** (Abbildungsmerkmal), i.e. there is an entity (called original) whose image or archetype the model is,

(ii) the **reduction attribute** (Verkürzungsmerkmal), i.e. only those attributes of the original that are relevant in the modelling context are represented,

(iii) the **pragmatic attribute**, i.e. the model is built in a specific context for a specific purpose.

Recall: Model



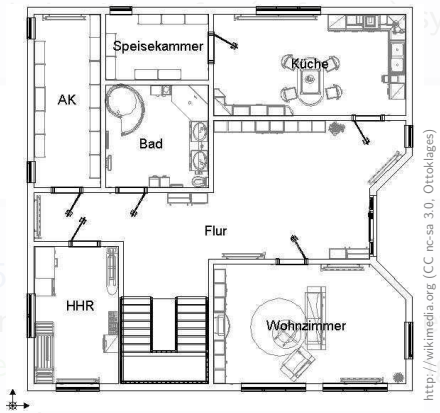
Model

Model Example: Floorplan

1. Requirements

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

2. Designmodel



http://wikimedia.org (CC nc-sa 3.0; Ottoklages)

3. System

http://wikimedia.org (CC nc-sa 3.0; Bobthebuilder82)

Observation: Floorplan **abstracts** from certain system properties, e.g. ...

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

→ architects can efficiently work on appropriate level of abstraction

21/77

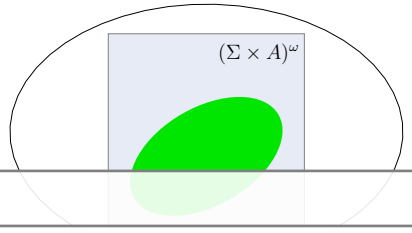
20/77

– 03 – 2015-04-30 – Smodel –

– 03 – 2015-04-30 – Smodel –

– 11 – 2015-06-22 – Sswmodel –

Recall: Model



Model

Model Example: Floorplan

1. Requirements

2. Designmodel

(Broaddus, 2010)

http://commons.wikimedia.org/wiki/File:Freiburg_Rieselfeld_2007.jpg, Norbert Blau, CC BY-SA 3.0

- water pipes/wiring, and
- wall decoration

→ architects can efficiently work on appropriate level of abstraction

21/77

20/77

– 03 – 2015-04

– 03 – 2015

– 11 – 2015-06-22 – Sswmodel –

Views and Viewpoints

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

IEEE 1471 (2000)

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

IEEE 1471 (2000)

Views and Viewpoints

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

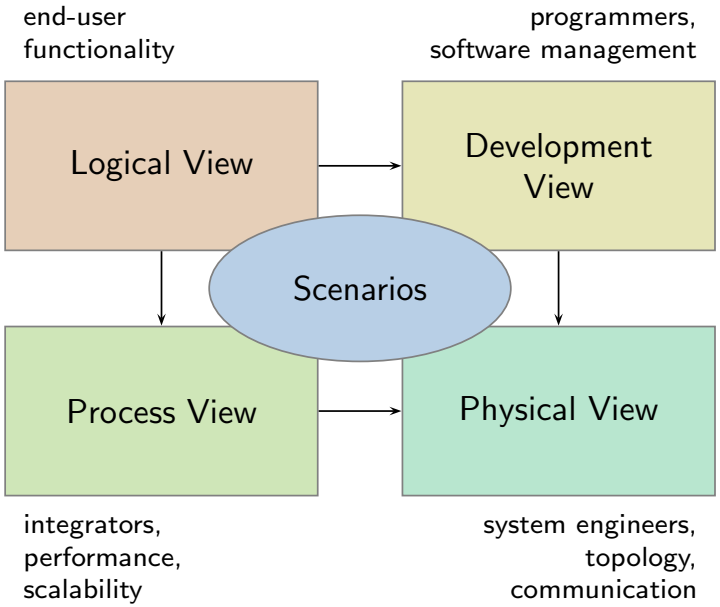
IEEE 1471 (2000)

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

IEEE 1471 (2000)

- A **perspective** is determined by concerns and information needs:
  - **team leader**, e.g., needs to know which team is working on what component,
  - **operator**, e.g., needs to know which component is running on which host,
  - **developer**, e.g., needs to know interfaces of other components.
  - etc.

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



(Ludewig and Lichter, 2013):

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

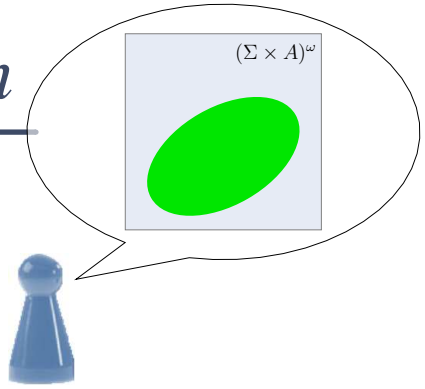
static view (∼ developer view): components of the architecture, their interfaces and relations. Possibly: assignment of development, test, etc. onto teams.

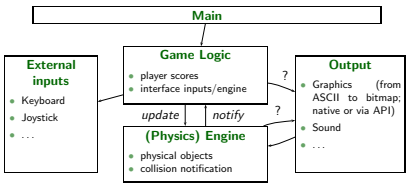
dynamic view (∼ process view): how and when are components instantiated and how do they work together at runtime.

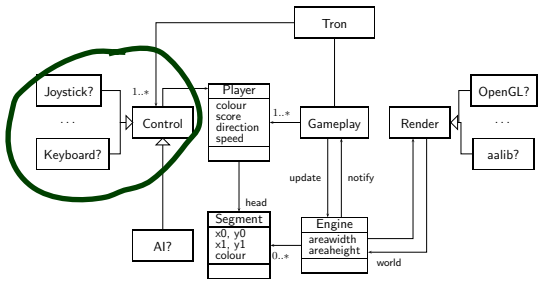
deployment view (∼ physical view): how are component instances mapped onto infrastructure and hardware units.

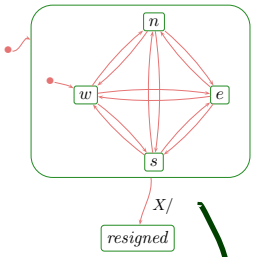
"Purpose of architecture: **support** functionality; functionality is not **part** of the architecture."

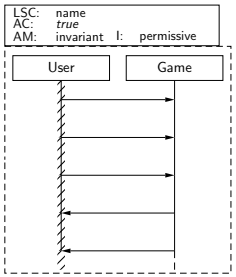
Views and Their Representation



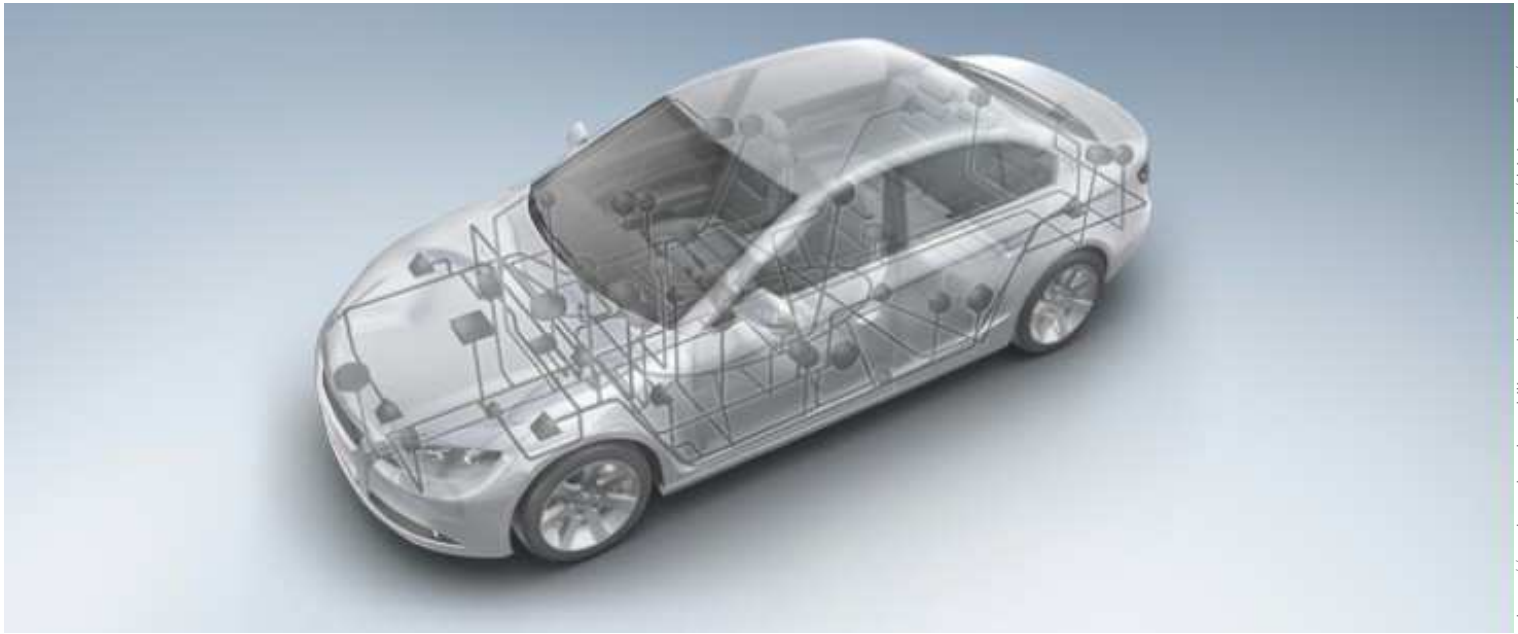








Process and Physical View

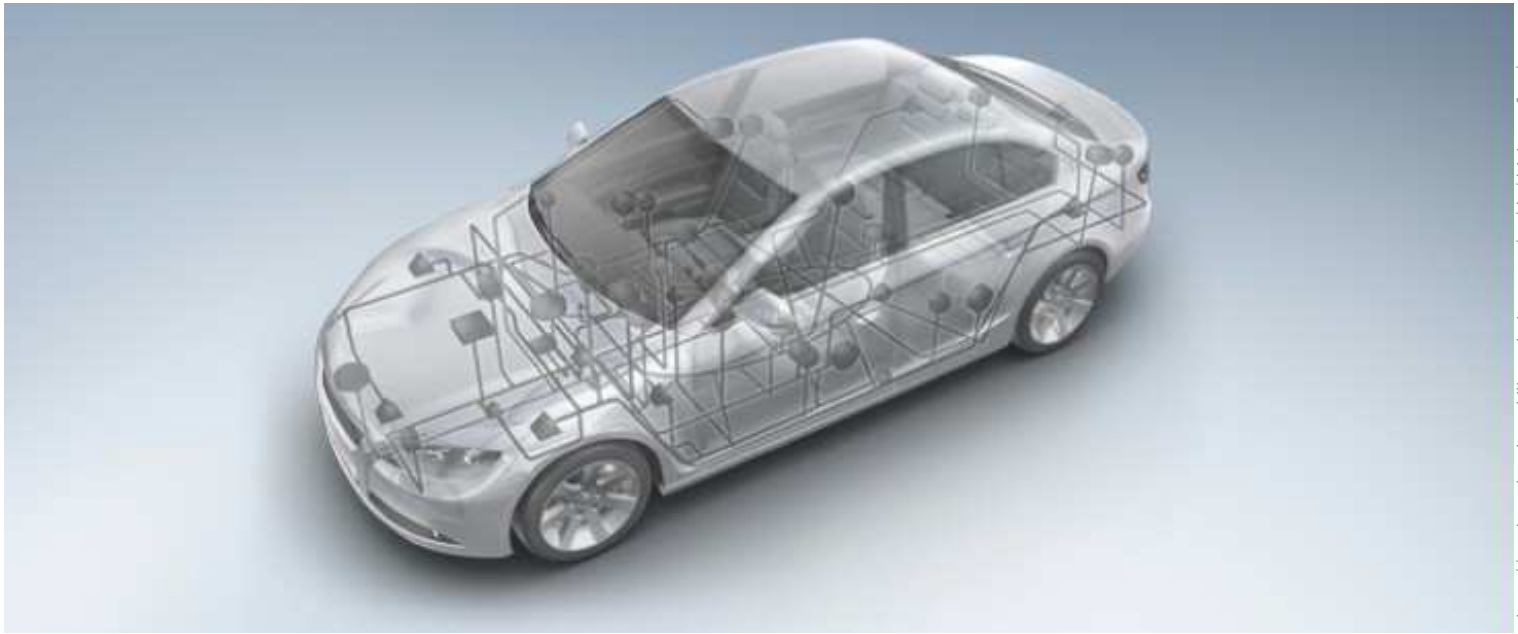


http://products.bosch-mobility-solutions.com/en/de/driving_safety/driving_safety_systems_for_commercial_vehicles/electronic_systems_1/electronic_systems_3.html — Robert Bosch GmbH

Example: modern cars

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

Process and Physical View



http://products.bosch-mobility-solutions.com/en/de/driving_safety/driving_safety_systems_for_commercial_vehicles/electronic_systems_1/electronic_systems_3.html — Robert Bosch GmbH

Example: modern cars

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

For, e.g., a simple smartphone App, process and physical view may be trivial or determined by framework ($\rightarrow$ later) — so no need for (extensive) particular documentation.

Other Views

- Form of the states σ:
  structure of S
- Computation paths π:
  behaviour of S

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

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

where

- $\sigma_i \in \Sigma$, $i \in \mathbb{N}_0$, is called **state** (or **configuration**), and
- $\alpha_i \in A$, $i \in \mathbb{N}_0$, is called **action** (or **event**).

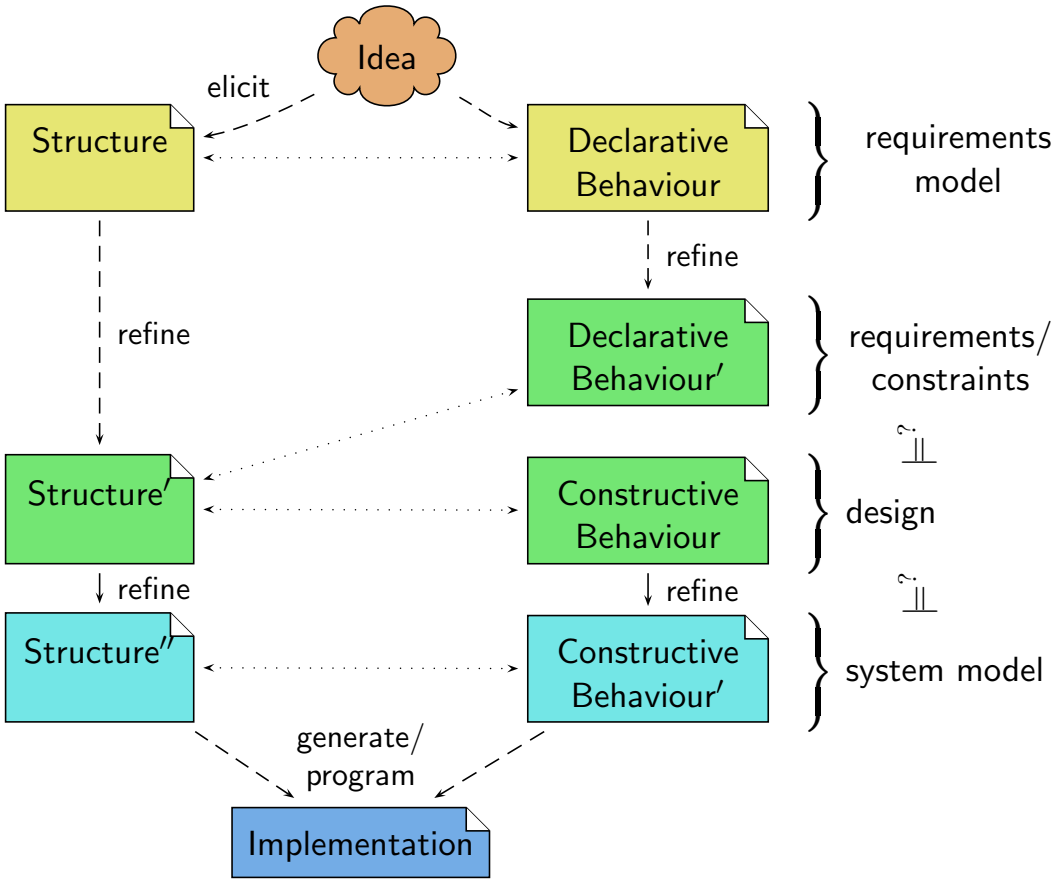
The (possibly partial) function $[\![\cdot]\!] : S \mapsto [\![S]\!]$ is called **interpretation** of S.

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

- **constructive**:
  "*constructs [of description] contain information needed in executing the model or in translating it into executable code.*" $\rightarrow$ **how things are computed**.
- **reflective** (or **assertive**):
  "*[description used] to derive and present views of the model, statically or during execution, or to set constraints on behavior in preparation for verification.*"
  $\rightarrow$ **what should (not) be computed**.

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

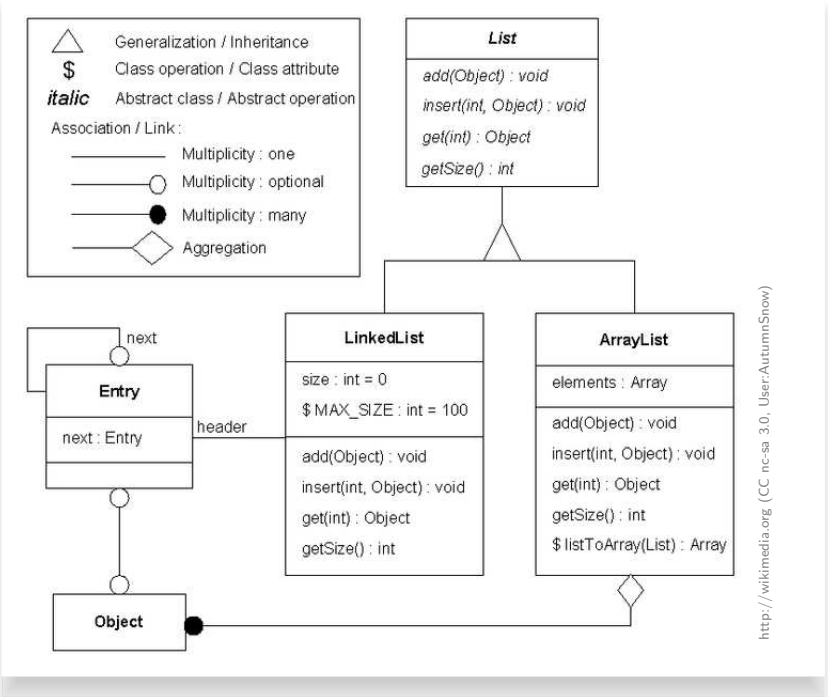
Model-Driven Software Engineering

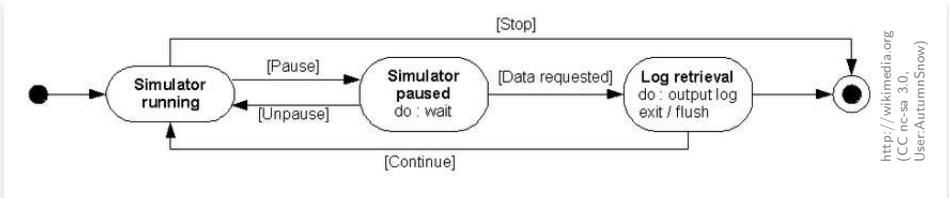


- (Jacobson et al., 1992): "System development is model building."
- Model **driven** software engineering (MDSE): **everything** is a model.
- Model **based** software engineering (MBSE): **some** models are used.

A Brief History of the Unified Modelling Language (UML)

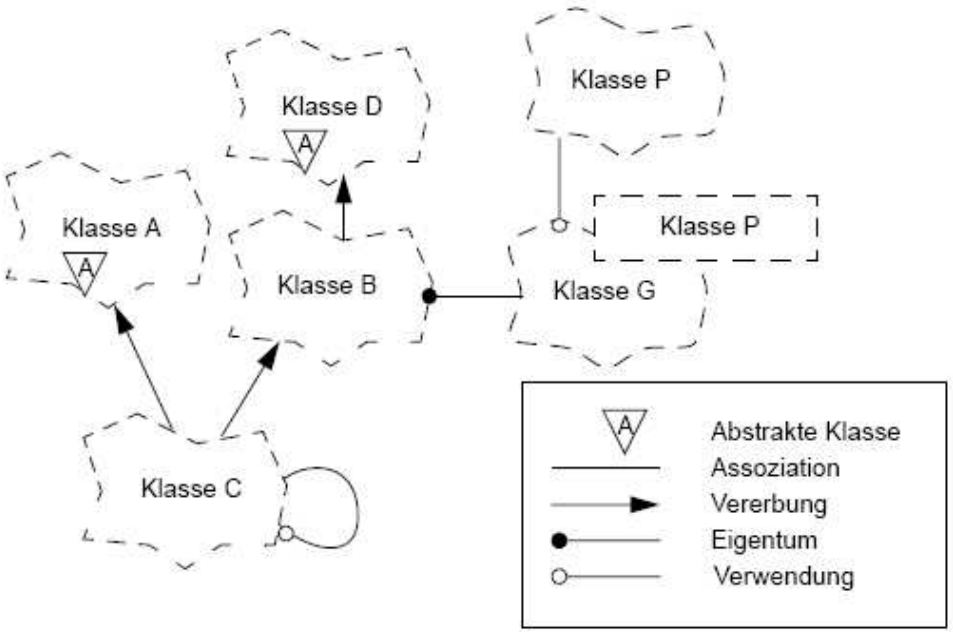
- Boxes/lines and finite automata are used to visualise software **for ages**.
- **1970's**, **Software Crisis**^{TM}
  — Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**
- Mid **1980**'s: **Statecharts**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

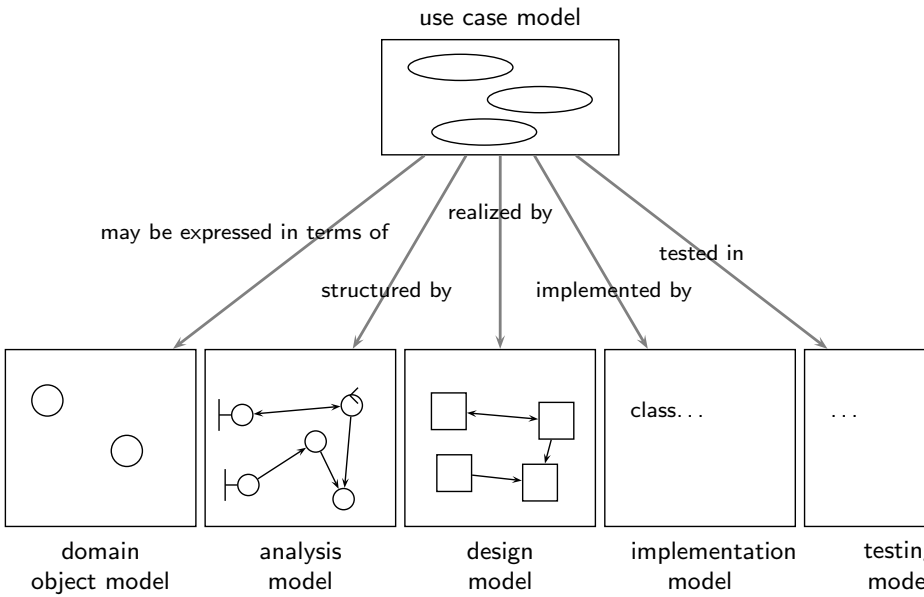


http://wikimedia.org (Public domain, Johannes Fasolt)

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...



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, 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)

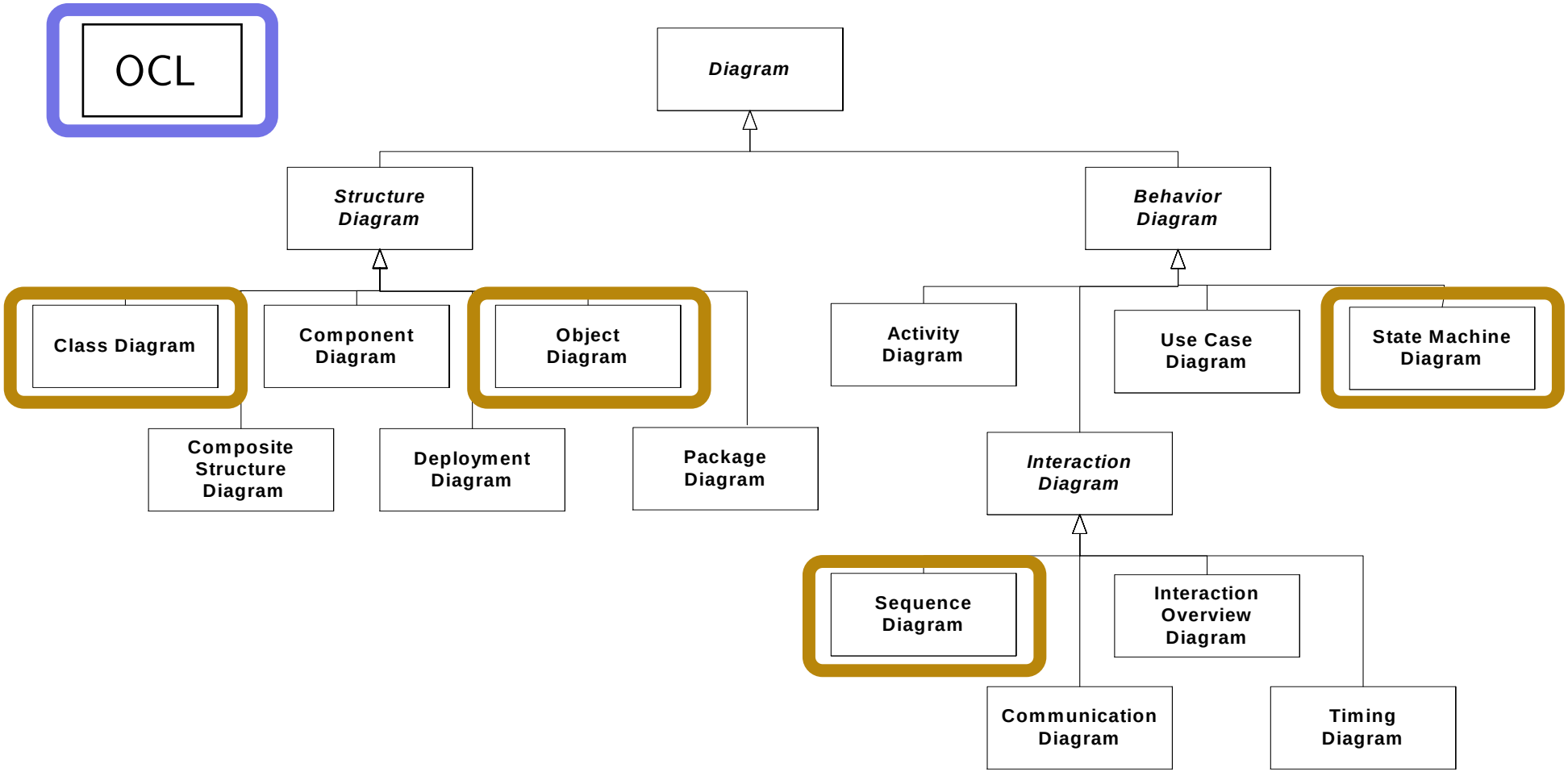


Figure A.5 - The taxonomy of structure and behavior diagram

Dobing and Parsons (2006)

References

References

Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002). Documenting software architecture: Documenting interfaces. Technical Report 2002-TN-015, CMU/SEI.

Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley, 2nd edition.

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

Broaddus, A. (2010). A tale of two eco-suburbs in Freiburg, Germany: Parking provision and car use. *Transportation Research Record*, 2187:114–122.

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

Ellis, W. J., II, R. F. H., Saunders, T. F., Poon, P. T., Rayford, D., Sherlund, B., and Wade, R. L. (1996). Toward a recommended practice for architectural description. In *ICECCS*, pages 408–413. IEEE Computer Society.

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

Harel, D. (1997). Some thoughts on statecharts, 13 years later. In Grumberg, O., editor, *CAV*, volume 1254 of *LNCS*, pages 226–231. Springer-Verlag.

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.

IEEE (2000). *Recommended Practice for Architectural Description of Software-Intensive Systems*. Std 1471.

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

Kruchten, P. (1995). The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50.

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

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.

Schumann, M., Steinke, J., Deck, A., and Westphal, B. (2008). Traceviewer technical documentation, version 1.0. Technical report, Carl von Ossietzky Universität Oldenburg und OFFIS.

Taylor, R. N., Medvidovic, N., and Dahofy, E. M. (2010). *Software Architecture Foundations, Theory, and Practice*. John Wiley and Sons.