Softwaretechnik / Software-Engineering

Lecture 11: Architecture & Design

2015-06-16

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

Albert-Ludwigs-Universität Freiburg, Germany

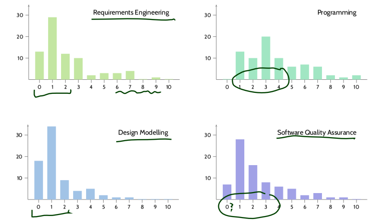
Topic Area Architecture & Design: Content

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

design principles

how to describe design

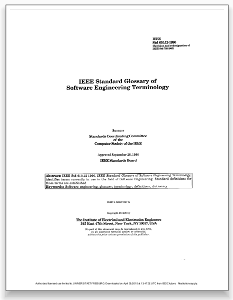
Survey: Previous Experience



Content

- **Vocabulary**
  - (software) system, component
  - module, interface
  - design, architecture
- **Principles of (Good) Design**
  - modularity
  - separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
- **information hiding / data encapsulation /** etc. by example
- **Software Modelling**
  - model
  - views & viewpoints
  - the 4+1 view
  - model-driven software engineering
- An outlook on **UML**

Introduction


IEEE Standard Glossary of Software Engineering Terminology

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 system's functionality and/ or data,
(2) restricts access to that subset via an explicitly defined interface, and
(3) has explicitly defined dependencies on its required execution context.
(Taylor et al., 2010)

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 visible from the outside only in so far as explicitly permitted by the programmers. ~ SW component (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

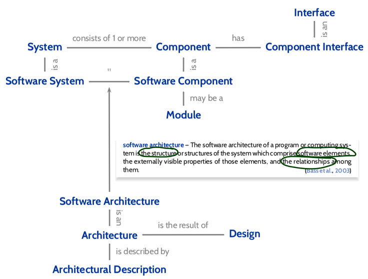
design–
(1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.
(2) The result of the process in (1). **IEEE 610.12** (1990)

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)

Once Again, Please

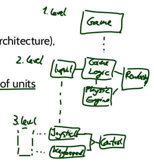


Goals and Relevance of Design

- 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":

Content

- **Vocabulary**
  - (software) system, component
  - module, interface
  - design, architecture
- **Principles of (Good) Design**
  - modularity
  - separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
- **information hiding / data encapsulation /** etc. by example
- **Software Modelling**
  - model
  - views & viewpoints
  - the 4+1 view
  - model-driven software engineering
- An outlook on **UML**

Principles of (Architectural) Design

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.

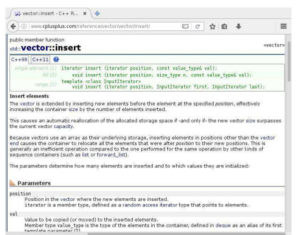
- **Advantages** / **goals**:
  - Hidden solutions may be **changed** without other components noticing, as long as the visible behaviour stays the same (e.g. the employed sorting algorithm). IOW: other components cannot (**unintentionally**) depend on details they are not supposed to.
  - Components 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.

4.) Data Encapsulation

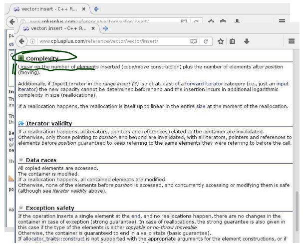
- Similar direction: **data encapsulation** (examples later).
  - Do not access data (variables, files, etc.) directly where needed, but encapsulate the data in a component which offers operations to access (read, write, etc.) the data.
    Real-World Example: Users do not write to bank accounts directly,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



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`.

Example

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

(ii) → negative effects when requirements change,

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

(iv) **abstract data types**,

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

(vi) **object oriented with** information hiding and data encapsulation.

Example: Module 'List of Names'

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

A Possible Implementation: Plain List, no Duplicates

```
#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];
}
```

```
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;
}
```

Output:

```
Berger
Meyer
Neumann
Schulz
Wernersen

Berger
Mayer
Neumann
Schulz
Wernersen

Berger
Mayer
Naumann
Schulz
Wernersen
```

access is bypassing the interface – no problem, so far

Change Interface: Support Duplicate Names

- **Task**: in addition, $count(n)$ should tell how many n's we have.
- **Operations**: (in interface of the module)
  - $insert($ `string` $n)$;
    - **pre-condition**: $N = n_0, \dots, n_i, n_{i+1}, \dots, n_{m-1}, m \in \mathbb{N}_0, \forall\, 0 \le j < m \bullet n_j <_{lex} n_{j+1}$
    - **post-condition**:
      - if $n_i <_{lex} n <_{lex} n_{i+1}$, $N = n_0, \dots, n_i, n, n_{i+1}, \dots, n_{m-1}$, $count(n) = 1$
      - if $n = n_i$ for some $0 \le i < m$, $N = old(N)$, $count(n) = old(count(n)) + 1$.
  - $remove($ `int` $i)$;
    - **pre-condition**: $N = n_0, \dots, n_{i-1}, n_i, n_{i+1}, \dots, n_{m-1}, m \in \mathbb{N}_0, 0 \le i < m$,
    - **post-condition**:
      - if $count(n_i) = 1$, $N = n_0, \dots, n_{i-1}, n_{i+1}, \dots, n_{m-1}$.
      - if $count(n_i) > 1$, $N = old(N)$, $count(n_i) = old(count(n_i)) - 1$.
  - $get($ `int` $i)$: `string`; and $dump()$;
    → unchanged contract

Changed Implementation: Support Duplicates

```
std::vector<int> count;
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()) {
    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 {
      ++(*( count.begin() +
            (it - names.begin()) ));
    }
  }
}

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];
}
```

```
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;
}
```

Output:

```
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
```

access is bypassing the interface – and corrupts the data-structure

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 ) {
```

```
#include "mod_deih.h"

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;
}
```

```
mod_deih_main.cpp: In function 'int main()':
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();




  remove( 2 );
  insert( "Naumann" );

  dump();

  return 0;
}
```

Output:

```
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

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 );
```

```
#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 );


  names[2] = "Naumann";




  dump( names );

  return 0;
}
```

```
mod_adt_main.cpp: In function 'int main()':
mod_adt_main.cpp:22:10: warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]
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 );




  remove( names, 2 );
  insert( names, "Naumann" );

  dump( names );

  return 0;
}
```

Output:

```
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;
}
```

Output:

```
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

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;
}
```

Output:

```
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
```

access is bypassing the interface – and corrupts the data-structure

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 {
    ++(*( count.begin() +
```

```
#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();


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




  names->dump();

  return 0;
}
```

```
In file included from mod_oo_deih_main.cpp:1:0:
mod_oo_deih.h: In function 'int main()':
mod_oo_deih.h:9:28: error: 'std::vector<std::basic_string<char> > Names::names' is private
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 {
      ++(*( count.begin() +
```

```
#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();


                                  ;

  names->remove( 2 );
  names->insert( "Naumann" );

  names->dump();

  return 0;
}
```

Output:

```
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) **enforcing** information hiding and data encapsulation by modules,

(iv) **abstract data types**,

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

(vi) **object oriented** **with** information hiding and data encapsulation.

Content

- **Vocabulary**
  - (software) system, component
  - module, interface
  - design, architecture
- **Principles of (Good) Design**
  - modularity
  - separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
- **information hiding** / **data encapsulation** / etc. by example
- **Software Modelling**
  - model
  - views & viewpoints
  - the 4+1 view
  - model-driven software engineering
- An outlook on **UML**

Software Modelling

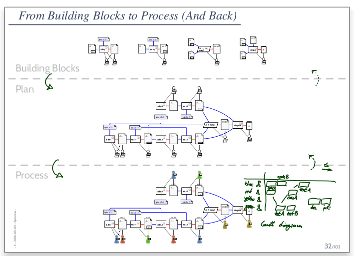
Model

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.

Example: Process Model

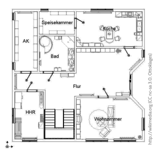


Example: Design-Models in Construction Engineering

1. Requirements

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

2. Designmodel



3. System

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

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

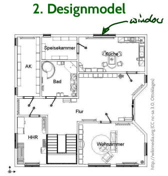
→ architects can efficiently work on appropriate level of abstraction

Example: Design-Models in Construction Engineering

1. Requirements

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

2. Designmodel



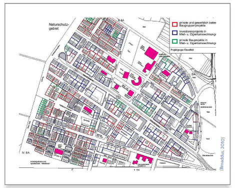
3. System

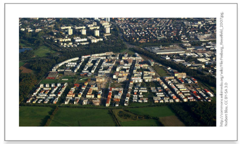
Observation (2): Floorplan **preserves**/**determines** certain system properties, e.g.,

- house and room extensions (to scale),
- presence/absence of windows and doors,
- placement of subsystems (such as windows).

→ find design errors before building the system (e.g. bathroom windows)

A Better Analogy is Maybe Regional Planning





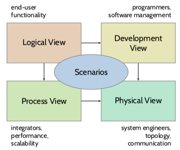
Views and Viewpoints

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.

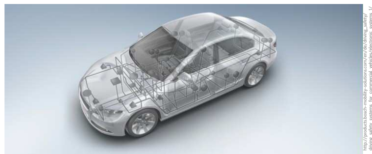
dynamic view (∼ **process view**): how and when are components instantiated and how do they work together at runtime.

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." ?!

too strict

Process and Physical View



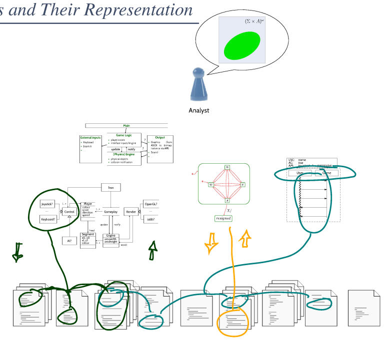
http://products.bosch-mobility-solutions.com/en/de/driving_safety/driving_safety_systems_for_commercial_vehicles/electronic_systems_2/electronic_systems_3.html – Robert Bosch GmbH

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 the employed framework (→ later) – so no need for (extensive) particular documentation.

Views and Their Representation



Structure vs. Behaviour

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

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

where

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

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

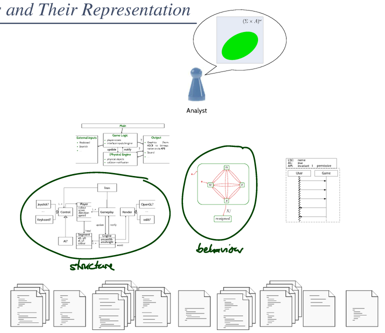
- **Form of the states** in Σ (also actions A): **structure** of S
- **Computation paths** π of S: **behaviour** of S

(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."
  → 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."
  → what should (or should not) be computed.

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

Views and Their Representation

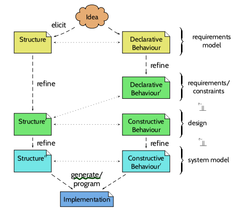


Model-Driven Software Engineering

- (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.

Model-Driven Software Engineering

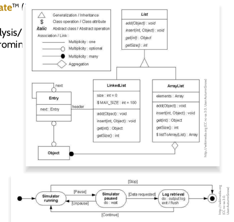
- (Jacobson et al., 1992): "System development is model building."
- Model **driven** software engineering (MDSE): **everything** is a model.
- Model **based** software engineering (MBSE): **some** models are used.



An Outlook to UML

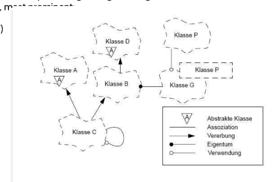
A Brief History of the Unified Modelling Language (UML)

- 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)

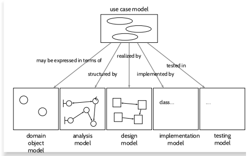
- Boxes/lines and finite automata are used to visualise software **for ages**.
- **1970's, Software Crisis**™
  – Idea: learn from engineering disciplines to handle growing complexity.
  Modelling languages: **Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams**
- Mid **1980's**: **Statecharts**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



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 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)

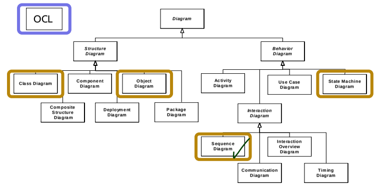


Figure A.5 - The taxonomy of structure and behavior diagram

Dobing and Parsons (2006)

Topic Area Architecture & Design: Content

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

Tell Them What You've Told Them…

- **Design** structures a system into **manageable units**.
- **Principles of (Good) Design**:
  - modularity, separation of concerns,
  - information hiding / data encapsulation
- **Model**: a concrete or mental image or archetype with
  - image attribute,
  - reduction attribute,
  - pragmatic attribute,

  here: abstract, formal, mathematical description.
- **Software Modelling**: views and viewpoints, e.g. 4+1
- **Model-driven** Software Engineering
- **Unified Modelling Language**:
  - a family of **modelling languages**.

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.

Glinz, M. (2008). Modellierung in der Lehre an Hochschulen: Thesen und Erfahrungen. *Informatik Spektrum*, 31(5):425–434.

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 (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.

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