Softwaretechnik / Software-Engineering

Lecture 13: Architecture and Design Patterns

2018-06-25

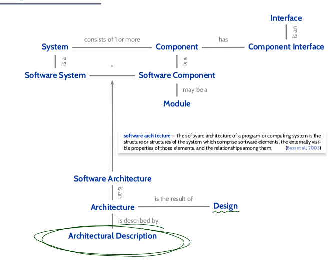
Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

- VL 11: **Introduction and Vocabulary**
- **Software Modelling**
  - model; views / viewpoints; 4+1 view
- **Modelling structure**
  - (simplified) class & object diagrams
  - VL 12: (simplified) object constraint logic (OCL)
- VL 13: **Principles of Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
- **Design Patterns**
- VL 14: **Modelling behaviour**
  - communicating finite automata (CFA)
  - Uppaal query language
  - VL 15: CFA vs. Software
  - Unified Modelling Language (UML)
    - basic state-machines
    - an outlook on hierarchical state-machines
- **Model-driven/-based Software Engineering**

Once Again, Please



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

9/55

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**.

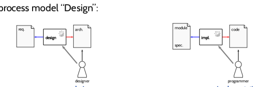
Design...

(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":



10/55

Content

- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ...by example
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

Principles of (Architectural) Design

Overview

1.) **Modularisation**
- split software into units / components of **manageable size**
- provide well-defined interface

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

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

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

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

1.) Modularisation

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

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

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

2.) Separation of Concerns

- **Separation of concerns** is a fundamental principle in software engineering:
  - each component should be **responsible for a particular area of tasks**,
  - components which try to cover different task areas tend to be unnecessarily complex, thus hard to understand and maintain.
- **Criteria** for separation/grouping:
  - in **object oriented design**, data and operations on that data are grouped into classes,
  - sometimes, functional aspects (features) like printing are realised as separate components,
  - separate **functional** and **technical** components,
    Example: logical flow of (logical) messages in a communication protocol (**functional**) vs. exchange of (physical) messages using a certain technology (**technical**).
  - assign flexible or variable functionality to own components.
    Example: different networking technology (wireless, etc.)
  - assign functionality which is expected to need extensions or changes later to own components.
  - separate system **functionality** and **interaction**
    Example: most prominently graphical user interfaces (GUI), also file input/output

3.) Information Hiding

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

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

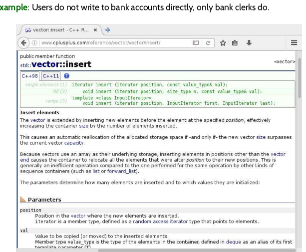
- **Note**: what is hidden is information which other components **need not know**
  (e.g., how data is stored and accessed, how operations are implemented).

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

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

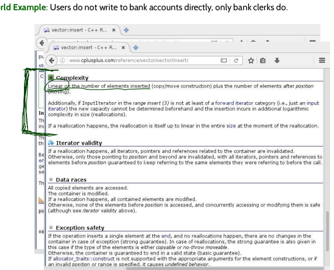
4.) Data Encapsulation

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



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

```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;
}
```

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**: $N = n_0, \dots, n_i, n_{i+1}, \dots, n_{m-1}, m \in \mathbb{N}_0, \forall\, 0 \leq j < m \bullet n_j <_{lex} n_{j+1}$
    - **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 $0 \leq i < m$, $N = old(N), count(n) = old(count(n)) + 1$.
  - $remove($ `int` $i\,)$;
    - **pre-condition**: $N = n_0, \dots, n_{i-1}, n_i, n_{i+1}, \dots, n_{m-1}, m \in \mathbb{N}_0, 0 \leq i < m,$
    - **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

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()) {
```

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

```
#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 );
      in->names.insert( it, n );
    } else {
      ++(*( in->count.begin() +
```

```
#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 );
    } else {
      ++(*( count.begin() +
            (it - names.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() +
            (it - names.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

- **Principles of (Good) Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ...by example
- **Architecture Patterns**
  - Layered Architectures, Pipe-Filter, Model-View-Controller.
- **Design Patterns**
  - Strategy, Examples
- **Libraries and Frameworks**

Architecture Patterns

Introduction

- Over decades of software engineering, many **clever**, **proved** and **tested** designs of solutions for particular problems emerged.
- **Question**: can we **generalise**, **document** and **re-use** these designs?
- **Goals**:
  - "**don't re-invent the wheel**",
  - benefit from "**clever**", from "**proven and tested**", and from "**solution**".

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

Introduction Cont'd

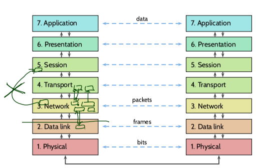
> **architectural pattern** – An architectural pattern expresses a fundamental structural organization schema for software systems.
> It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
> Buschmann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

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

Layered Architectures

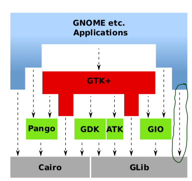
Example: Layered Architectures

- (Züllighoven, 2005):
  A **layer** whose components only interact with components of their **direct neighbour** layers is called **protocol-based** layer.
  A **protocol-based layer** hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.
- **Example: The ISO/OSI reference model.**



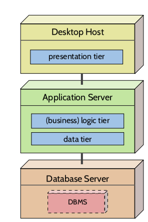
Example: Layered Architectures Cont'd

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



Example: Three-Tier Architecture

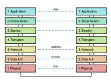
- **presentation layer** (or **tier**):
  user interface; presents information obtained from the logic layer to the user, controls interaction with the user, i.e. requests actions at the logic layer according to user inputs.
- **logic layer**:
  core system functionality; layer is designed without information about the presentation layer, may only read/write data according to data layer interface.
- **data layer**:
  persistent data storage; hides information about how data is organised, read, and written, offers particular chunks of information in a form useful for the logic layer.

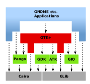


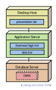
(Ludewig and Lichter, 2013)

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

Layered Architectures: Discussion





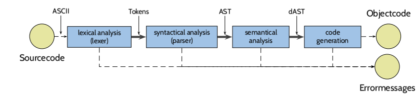


- **Advantages**:
  - **protocol-based**:
    only neighbouring layers are coupled, i.e. components of these layers interact,
  - coupling is low, data usually encapsulated,
  - changes have local effect (only neighbouring layers affected),
  - **protocol-based**: **distributed** implementation often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

Pipe-Filter

Example: Pipe-Filter

Example: Compiler



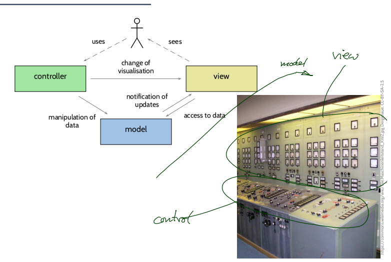
Example: UNIX Pipes

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

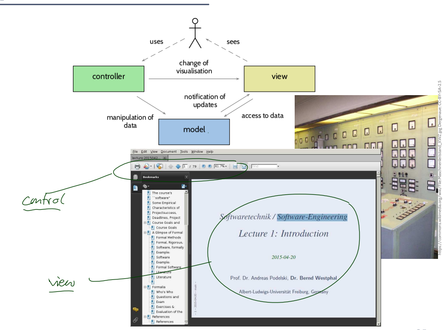
- **Disadvantages**:
  - if the filters use a common data exchange format, all filters may need changes if the format is changed, or need to employ (costly) conversions.
  - filters do not use global data, in particular not to handle error conditions.

Model-View-Controller

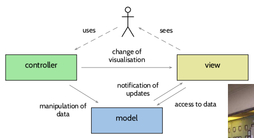
Example: Model-View-Controller

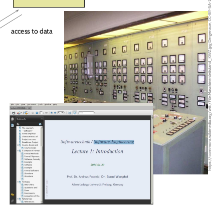


Example: Model-View-Controller



Example: Model-View-Controller



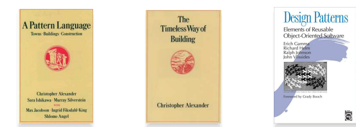


- **Advantages**:
  - one model can serve multiple view/controller pairs;
  - view/controller pairs can be added and removed at runtime;
  - model visualisation always up-to-date in all views;
  - distributed implementation (more or less) easily.
- **Disadvantages**:
  - if the view needs **a lot of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

Design Patterns

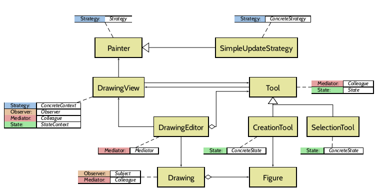
Design Patterns

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



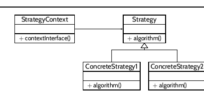
> **Design patterns** ... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
>
> A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable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

Example: Pattern Usage and Documentation

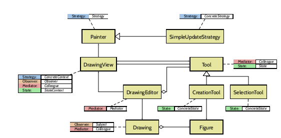


Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

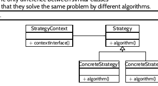
Example: Strategy

	Strategy
Problem	The only difference between similar classes is that they solve the same problem by different algorithms.
Solution	• Have one class **StrategyContext** with all common operations. • Another class **Strategy** provides signatures for all operations to be implemented differently. • From Strategy, derive one sub-class **ConcreteStrategy** for each implementation alternative. • StrategyContext uses concrete Strategy-objects to execute the different implementations via delegation.
Structure	

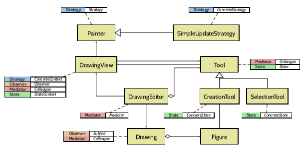
Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	Strategy
Problem	The only difference between similar classes is that they solve the same problem by different algorithms.
Solution	
Structure	

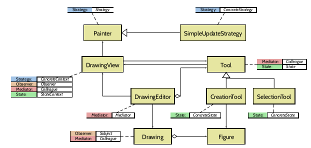
Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	Observer
Problem	Multiple objects need to adjust their state if one particular other object is changed.
Example	All GUI object displaying a file system need to change if files are added or removed.

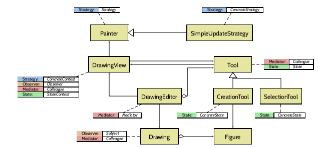
Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	State
Problem	The behaviour of an object depends on its (internal) state.
Example	The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.

Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	Mediator
Problem	Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.
Example	Appearance and state of different means of interaction (menus, buttons, input fields) in a graphical user interface (GUI) should be consistent in each interaction state.

Other Patterns: Singleton and Memento

	Singleton
Problem	Of one class, exactly one instance should exist in the system.
Example	Print spooler.

	Memento
Problem	The state of an object needs to be archived in a way that allows to re-construct this state without violating the principle of data encapsulation.
Example	Undo mechanism.

Design Patterns: Discussion

"The development of design patterns is considered to be one of the most important innovations of software engineering in recent years."

(Ludewig and Lichter, 2013)

- **Advantages**:
  - **(Re-)use** the experience of others and employ well-proven solutions.
  - Can improve on **quality criteria** like changeability or re-use.
  - Provide a **vocabulary** for the design process, thus facilitates documentation of architectures and discussions about architecture.
  - Can be combined in a flexible way, one class in a particular architecture can correspond to roles of multiple patterns.
  - Helps teaching software design.
- **Disadvantages**:
  - Using a pattern is not a value as such. Having too much global data cannot be justified by "but it's the pattern Singleton".
  - **Again**: reading is easy, writing need not be.

    Here: Understanding abstract descriptions of design patterns or their use in existing software may be easy – using design patterns appropriately in new designs requires (**surprise, surprise**) experience.

Libraries and Frameworks

Libraries and Frameworks

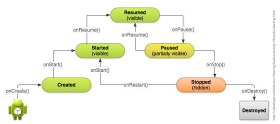
- **(Class) Library**:
  a collection of operations or classes offering generally usable functionality in a re-usable way.

  Examples:

  - `libc` – standard C library (is in particular abstraction layer for operating system functions),
  - `GMP` – GNU multi-precision library, cf. Lecture 6.
  - `libz` – compress data.
  - `libxml` – read (and validate) XML file, provide DOM tree.

- **Framework**: class hierarchies which determine a generic solution for similar problems in a particular context.

  - **Example**: Android Application Framework



Libraries and Frameworks

- **(Class) Library**:
  a collection of operations or classes offering generally usable functionality in a re-usable way.

  Examples:

  - `libc` – standard C library (is in particular abstraction layer for operating system functions),
  - `GMP` – GNU multi-precision library, cf. Lecture 6.
  - `libz` – compress data.
  - `libxml` – read (and validate) XML file, provide DOM tree.

- **Framework**: class hierarchies which determine a generic solution for similar problems in a particular context.

  - **Example**: Android Application Framework

- The difference lies in **flow-of-control**:
  library modules are called from user code, frameworks call user code.

- **Product line**: parameterised design/code
  ("all turn indicators are equal, turn indicators in premium cars are more equal").

Quality Criteria on Architectures

Quality Criteria on Architectures

- **testability**
  - architecture design should keep testing (or formal verification) in mind
    (**buzzword** "design for verification"),
  - high locality of design units may make testing significantly easier (module testing),
  - particular testing interfaces may improve testability
    (e.g. allow injection of user input not only via GUI; or provide particular log output for tests).
- **changeability, maintainability**
  - most systems that are used need to be changed or maintained,
    in particular when requirements change,
  - **risk assessment**: parts of the system with high probability for changes should be designed such that changes are possible with acceptable effort (abstract, modularise, encapsulate),
- **portability**
  - **porting**: adaptation to different platform (OS, hardware, infrastructure).
  - systems with a long lifetime may need to be adapted to different platforms over time, infrastructure like databases may change (→ introduce abstraction layer).
- **Note**:
  - a good design (model) is first of all supposed to **support the solution**,
  - it **need not be** a good **domain model**.

References

References

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

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

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

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

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

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

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

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

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

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