Softwaretechnik / Software-Engineering

Lecture 14: Architecture & Design Patterns, Software Quality Assurance

2019-07-08

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
Introduction and Vocabulary

Software Modelling
- model; views / viewpoints; 4+1 view

Modelling structure
- (simplified) Class & Object diagrams
- (simplified) Object Constraint Logic (OCL)

Modelling behaviour
- Communicating Finite Automata (CFA)
- Uppaal query language
- CFA vs. Software
- Unified Modelling Language (UML)
  - basic state-machines
  - an outlook on hierarchical state-machines

Model-driven/-based Software Engineering

Principles of Design
- modularity, separation of concerns
- information hiding and data encapsulation
- abstract data types, object orientation

Design Patterns
Content (Part I: Architecture & Design)

- Principles of (Good) Design Cont’d
  - 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
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.
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.
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.

![Insert](image.png)

**Insert elements**

The `vector` is extended by inserting new elements before the element at the specified `position`, effectively increasing the container `size` by the number of elements inserted.

This causes an automatic reallocation of the allocated storage space if-and only if-the new vector `size` surpasses the current vector `capacity`.

Because vectors use an array as their underlying storage, inserting elements in positions other than the `vector` end causes the container to relocate all the elements that were after `position` to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as `list` or `forward_list`).

The parameters determine how many elements are inserted and to which values they are initialized:

- **Parameters**
  - **position:** Position in the `vector` where the new elements are inserted.
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.
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)
  - \( \text{insert}(\text{string } n) \);
    - **pre-condition:** \( N = n_0, \ldots, n_i, n_{i+1}, \ldots, n_{m-1}, m \in \mathbb{N}_0, \forall 0 \leq j < m \cdot n_j < \text{lex } n_{j+1} \)
    - **post-condition:** \( N = n_0, \ldots, n_i, n, n_{i+1}, \ldots, n_{m-1} \) if \( n_i < \text{lex } n < \text{lex } n_{i+1} \), \( N = \text{old}(N) \) otherwise.

  - \( \text{remove}(\text{int } i) \);
    - **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:** \( N = n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_{m-1} \).

  - \( \text{get}(\text{int } i) : \text{string} \);
    - **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:** \( N = \text{old}(N), \text{retval} = n_i \).

  - \( \text{dump}() \);
    - **pre-condition:** \( N = n_0, \ldots, n_{m-1}, m \in \mathbb{N}_0 \),
    - **post-condition:** \( N = \text{old}(N) \).
    - **side-effect:** \( n_0, \ldots, n_{m-1} \) 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];
}

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:
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
```
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];
}

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:

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

access is bypassing the interface – no problem, so far
Change Interface: Support Duplicate Names

- **Task**: in addition, \( \text{count}(n) \) should tell how many \( n \)'s we have.

- **Operations**: (in interface of the module)

  - \( \text{insert}( \text{string } n ) \);
    - **pre-condition**: 
      \[ N = n_0, \ldots, n_i, n_{i+1}, \ldots, n_{m-1}, m \in \mathbb{N}_0, \forall 0 \leq j < m \cdot n_j <_{\text{lex}} n_{j+1} \]
    - **post-condition**:  
      - if \( n_i <_{\text{lex}} n <_{\text{lex}} n_{i+1} \), \( N = n_0, \ldots, n_i, n, n_{i+1}, \ldots, n_{m-1}, \text{count}(n) = 1 \)
      - if \( n = n_i \) for some \( 0 \leq i < m \), \( N = \text{old}(N), \text{count}(n) = \text{old}(\text{count}(n)) + 1 \).

  - \( \text{remove}( \text{int } i ) \);
    - **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**:  
      - if \( \text{count}(n_i) = 1 \), \( N = n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_{m-1} \).
      - if \( \text{count}(n_i) > 1 \), \( N = \text{old}(N), \text{count}(n_i) = \text{old}(\text{count}(n_i)) - 1 \).

  - \( \text{get}( \text{int } i ) : \text{string}; \) and \( \text{dump}() \);
    - \( \rightarrow \) unchanged contract
Changed Implementation: Support Duplicates

```cpp
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("Naumann");
    dump();
    remove(1);
    insert("Mayer");
    dump();
    names[2] = "Naumann";
    dump();
    return 0;
}
```

Output:

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berger</td>
<td>1</td>
</tr>
<tr>
<td>Meyer</td>
<td>1</td>
</tr>
<tr>
<td>Neumann</td>
<td>2</td>
</tr>
<tr>
<td>Schulz</td>
<td>1</td>
</tr>
<tr>
<td>Wernersen</td>
<td>1</td>
</tr>
<tr>
<td>Berger</td>
<td>1</td>
</tr>
<tr>
<td>Mayer</td>
<td>1</td>
</tr>
<tr>
<td>Neumann</td>
<td>2</td>
</tr>
<tr>
<td>Schulz</td>
<td>1</td>
</tr>
<tr>
<td>Wernersen</td>
<td>1</td>
</tr>
</tbody>
</table>
# Changed Implementation: Support Duplicates

```cpp
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];
}
```

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

access is bypassing the interface – and corrupts the data-structure
Data Encapsulation + Information Hiding

### Header

```cpp
#include <string>

void dump();
void insert( std::string n );
void remove( int i );
std::string get( int i );
```

### Source

```cpp
#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];
}
```

### Compilation Error

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

```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;
}
```
# Data Encapsulation + Information Hiding

```cpp
#include <string>
#include <algorithm>
#include <vector>
#include <iostream>
#include "mod_deih.h"

int main() {
    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];
    }

    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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berger</td>
<td>1</td>
</tr>
<tr>
<td>Meyer</td>
<td>1</td>
</tr>
<tr>
<td>Neumann</td>
<td>2</td>
</tr>
<tr>
<td>Schulz</td>
<td>1</td>
</tr>
<tr>
<td>Wernersen</td>
<td>1</td>
</tr>
<tr>
<td>Mayer</td>
<td>1</td>
</tr>
<tr>
<td>Naumann</td>
<td>1</td>
</tr>
<tr>
<td>Neumann</td>
<td>1</td>
</tr>
<tr>
<td>Schulz</td>
<td>1</td>
</tr>
<tr>
<td>Wernersen</td>
<td>1</td>
</tr>
</tbody>
</table>
Abstract Data Type

```cpp
#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 );
```

```cpp
#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 );
    return 0;
}
```

```cpp
#define AVOID_PROBLEM

int main() {
    Names names = new_Names();
    names[2] = "Naumann";
}
```

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

```cpp
#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 );
```

```cpp
#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
Mayer:1
Neumann:2
Schulz:1
Wernersen:1
```

```
Berger:1
Mayer:1
Neumann:2
Schulz:1
Wernersen:1
```
# Object Oriented

```cpp
#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);
};
```

```cpp
#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->dump();
  names->names[2] = "Naumann";
  names->dump();
  return 0;
}
```

Output:

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berger</td>
<td>1</td>
</tr>
<tr>
<td>Meyer</td>
<td>1</td>
</tr>
<tr>
<td>Schulz</td>
<td>1</td>
</tr>
<tr>
<td>Neumann</td>
<td>2</td>
</tr>
<tr>
<td>Wernersen</td>
<td>1</td>
</tr>
<tr>
<td>Mayer</td>
<td>1</td>
</tr>
<tr>
<td>Naumann</td>
<td>2</td>
</tr>
<tr>
<td>Schulz</td>
<td>1</td>
</tr>
<tr>
<td>Wernersen</td>
<td>1</td>
</tr>
</tbody>
</table>
# Object Oriented

```cpp
#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);
};
```

```cpp
#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()));
        }
    }
}
```

```cpp
#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
#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);
};

#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;
}
### Header

```cpp
#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

```cpp
#include "mod_oo_deih.h"

int main() {
    Names* names = new Names();
    names->names = "Berger";
    names->insert( "Schulz" );
    names->insert( "Neumann" );
    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
Mayer:1
Neumann:2
Schulz:1
Wernersen:1
Berger:1
Mayer:1
Neumann:2
Schulz:1
Wernersen:1
Berger:1
Mayer:1
Naumann:1
Schulz:1
Wernersen:1
```
(i) **information hiding** and **data encapsulation not enforced**, 
(ii) $\rightarrow$ 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.
Principles of (Good) Design Cont’d
- 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
Design Approaches
Software Modelling

\[ \sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \sigma_2 \xrightarrow{\alpha_3} \sigma_3 \]

\[ \epsilon \subseteq E \]

Analyzer

Constructive

Reflective

Image

Pre-image
Development Approaches

- **top-down** risk: needed functionality hard to realise on target platform.
- **bottom-up** risk: lower-level units do not “fit together”.
- **inside-out** risk: user interface needed by customer hard to realise with existing system,
- **outside-in** risk: elegant system design not reflected nicely in (already fixed) UI.
Architecture Patterns
Over decades of software engineering, many clever, proved and tested designs of solutions for particular problems emerged.

**Question**: can we generalise, document and re-use these designs?

**Goals**:
- "don’t re-invent the wheel",
- benefit from “clever”, from “proven and tested”, and from “solution”.

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

*Buschmann et al. (1996)*
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
(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.

![Diagram showing layered architectures with GNOME, GTK+, Pango, GDK, ATK, GIO, Cairo, and GLib]
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.

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

(Ludewig and Lichter, 2013)
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

- Controller uses View.
- View sees Model.
- Model manipulates data.
- Controller receives updates from Model.
- View accesses data from Model.
- Controller manipulates visualisation.
- View notifies updates.

Image: https://commons.wikimedia.org/wiki/File:Maschinenleitstand_KWZ.jpg
License: CC-BY-SA-2.5
Example: Model-View-Controller

- **Controller**
  - Sees
  - Uses
  - Change of visualisation
  - Notification of updates
  - Manipulation of data
  - Access to data

- **View**
  - Change of visualisation
  - Notification of updates

- **Model**
  - Change of visualisation
  - Notification of updates
  - Access to data

![Diagram of Model-View-Controller](https://commons.wikimedia.org/wiki/File:Maschinenleitstand_KWZ.jpg)
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*)
Pattern usage in JHotDraw framework ([JHotDraw, 2007](#))(Diagram: ([Ludewig and Lichter, 2013](#)))
## Example: Strategy

<table>
<thead>
<tr>
<th>Problem</th>
<th>The only difference between similar classes is that they solve the same problem by different algorithms.</th>
</tr>
</thead>
</table>
| 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 | ![StrategyContext](image) ➔ ![Strategy](image) ➔ ![ConcreteStrategy1](image) ➔ ![ConcreteStrategy2](image) |
**Example: Pattern Usage and Documentation**

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

<table>
<thead>
<tr>
<th>Strategy</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
<td>The only difference between similar classes is that they solve the same problem by different algorithms.</td>
</tr>
<tr>
<td>Solution</td>
<td>…</td>
</tr>
<tr>
<td>Structure</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StrategyContext</th>
<th>ConcreteStrategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ contextInterface()</td>
<td>+ algorithm()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>ConcreteState</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ algorithm()</td>
<td>+ algorithm()</td>
</tr>
</tbody>
</table>
Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))
Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

<table>
<thead>
<tr>
<th>Problem</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple objects need to adjust their state if one particular other object is changed.</td>
<td>All GUI object displaying a file system need to change if files are added or removed.</td>
</tr>
</tbody>
</table>
Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

<table>
<thead>
<tr>
<th>State</th>
<th>Problem</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
<td>The behaviour of an object depends on its (internal) state.</td>
<td>The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.</td>
</tr>
</tbody>
</table>
Pattern usage in JHotDraw framework ([JHotDraw, 2007] (Diagram: [Ludewig and Lichter, 2013]))

<table>
<thead>
<tr>
<th>Mediator</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.</td>
</tr>
<tr>
<td></td>
<td>Example</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
## Other Patterns: Singleton and Memento

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem</strong></td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memento</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem</strong></td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
</tbody>
</table>
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
(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),
- 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),
  - 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.
Tell Them What You’ve Told Them...

- **Architecture & Design Patterns**
  - allow **re-use** of practice-proven designs,
  - promise easier **comprehension** and **maintenance**.

- **Notable Architecture Patterns**
  - Layered Architecture,
  - Pipe-Filter,
  - Model-View-Controller.

- **Design Patterns**: read *(Gamma et al., 1995)*

- **Rule-of-thumb**:
  - **library modules** are called from user-code,
  - **framework modules** call user-code.
Code Quality Assurance
Introduction and Vocabulary
- Test case, test suite, test execution.
- Positive and negative outcomes.

Limits of Software Testing

Glass-Box Testing
- Statement-, branch-, term-coverage.

Other Approaches
- Model-based testing,
- Runtime verification.

Program Verification
- partial and total correctness,
- Proof System PD.

Review
Content (Part II: Code Quality Assurance)

- Introduction
  - quotes on testing,
  - systematic testing vs. ‘rumprobieren’

- Test Case
  - definition,
  - execution,
  - positive and negative.

- Test Suite

- Limits of Software Testing
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range vs. point errors

- More Vocabulary
Testing: Introduction
“Testing is the execution of a program with the goal to discover errors.”  
(G. J. Myers, 1979)

“Testing is the demonstration of a program or system with the goal to show that it does what it is supposed to do.”  
(W. Hetzel, 1984)

“Software testing can be used to show the presence of bugs, but never to show their absence!”  
(E. W. Dijkstra, 1970)

**Rule-of-thumb**: (fairly systematic) tests discover half of all errors.  
(Ludewig and Lichter, 2013)
Recall:

**Definition.** Software is a finite description $S$ of a (possibly infinite) set $\llbracket S \rrbracket$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$ where

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

The (possibly partial) function $\llbracket \cdot \rrbracket : S \mapsto \llbracket S \rrbracket$ is called interpretation of $S$.

- From now on, we assume that states consist of an input and an output/internal part, i.e., there are $\Sigma_{in}$ and $\Sigma_{out}$ such that $\Sigma = \Sigma_{in} \times \Sigma_{out}$.
- Computation paths are then of the form
  $$\pi = \begin{pmatrix} \sigma_i^0 \\ \sigma_o^0 \end{pmatrix} \xrightarrow{\alpha_1} \begin{pmatrix} \sigma_i^1 \\ \sigma_o^1 \end{pmatrix} \xrightarrow{\alpha_2} \cdots$$
- We use $\pi \downarrow \Sigma_{in}$ to denote $\pi = \sigma_0^i \xrightarrow{\alpha_1} \sigma_1^i \xrightarrow{\alpha_2} \cdots$, i.e. the projection of $\pi$ onto $\Sigma_{in}$. 
Definition. A test case $T$ over $\Sigma$ and $A$ is a pair $(In, Soll)$ consisting of

- a description $In$ of sets of finite input sequences,
- a description $Soll$ of expected outcomes,

and an interpretation $[\cdot]$ of these descriptions:

- $[In] \subseteq (\Sigma_{in} \times A)^*$,
- $[Soll] \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)\omega$
Definition. A test case $T$ over $\Sigma$ and $A$ is a pair $(In, Soll)$ consisting of

- a description $In$ of sets of finite input sequences,
- a description $Soll$ of expected outcomes,

and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions:

- $\llbracket In \rrbracket \subseteq (\Sigma_{in} \times A)^*$,
- $\llbracket Soll \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

Examples:

- Test case for procedure `strlen`: $String \to \mathbb{N}$, $s$ denotes parameter, $r$ return value:

  $T = \llbracket s = "abc" \rrbracket \times \llbracket r = 3 \rrbracket$

  $\llbracket s = "abc" \rrbracket = \{ \sigma_0 \xrightarrow{\tau} \sigma_1 | \sigma_0(s) = "abc" \}$, $\llbracket r = 3 \rrbracket = \{ \sigma_0 \xrightarrow{\tau} \sigma_1 | \sigma_1(r) = 3 \}$

  Shorthand notation: $T = ("abc", 3)$.

- “Call `strlen()` with string "abc", expect return value 3.”
**Definition.** A test case $T$ over $\Sigma$ and $A$ is a pair $(\text{In}, \text{Soll})$ consisting of

- a description $\text{In}$ of sets of finite input sequences,
- a description $\text{Soll}$ of expected outcomes,

and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions:

- $\llbracket \text{In} \rrbracket \subseteq (\Sigma_{\text{in}} \times A)^*$,
- $\llbracket \text{Soll} \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

**Examples:**

- Test case for vending machine.

  $$T = (C50, \text{WATER}; \text{DWATER})$$

  $$\llbracket C50, \text{WATER} \rrbracket = \{ \sigma_0 \xrightarrow{C50} \sigma_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} \sigma_{j-1} \xrightarrow{\text{WATER}} \sigma_j \},$$

  $$\llbracket \text{DWATER} \rrbracket = \{ \sigma_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} \sigma_{k-1} \xrightarrow{\text{DWATER}} \sigma_k \mid k \leq 10 \},$$

- “Send event $C50$ and any time later $\text{WATER}$, expect $\text{DWATER}$ after 10 steps the latest.”
Definition. A test case $T$ over $\Sigma$ and $A$ is a pair $(In, Soll)$ consisting of

- a description $In$ of sets of finite input sequences,
- a description $Soll$ of expected outcomes,

and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions:

- $\llbracket In \rrbracket \subseteq (\Sigma_{in} \times A)^*$,
- $\llbracket Soll \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

Note:

- **Input sequences** can consider
  - input data, possibly with timing constraints,
  - other interaction, e.g., from network,
  - initial memory content,
  - etc.

- **Input sequences** may leave degrees of freedom to tester.
- **Expected outcomes** may leave degrees of freedom to system.
Executing Test Cases

- A computation path

\[ \pi = \left( \sigma_0^i \sigma_0^o \right)^{\alpha_1} \left( \sigma_1^i \sigma_1^o \right)^{\alpha_2} \ldots \in [In] \]

from \([S]\) is called execution of test case \((In, Soll)\) if and only if

- there is \(n \in \mathbb{N}\) such that \(\sigma_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \sigma_n \downarrow \Sigma_{in} \in [In] \).

(“A prefix of \(\pi\) corresponds to an input sequence”).

Execution \(\pi\) of test case \(T\) is called

- **successful** (or **positive**) if and only if \(\pi \notin [Soll] \).
  
  - Intuition: an error has been discovered.
  
  - Alternative: test item \(S\) **failed to pass the test**.
  
  - Confusing: “test failed”.

- **unsuccessful** (or **negative**) if and only if \(\pi \in [Soll] \).
  
  - Intuition: no error has been discovered.
  
  - Alternative: test item \(S\) **passed the test**.
  
  - Okay: “test passed”.
A **test suite** is a finite set of test cases \( \{T_1, \ldots, T_n\} \).

An **execution** of a **test suite** is a set of computation paths, such that there is at least one execution for each test case.

An **execution** of a **test suite** is called **positive** if and only if at least one test case execution is **positive**. Otherwise, it is called **negative**.
Not Executing Test Cases

- Consider the test case
  \[ T = ("", 0) \]
  for procedure `strlen`.
  (“Empty string has length 0.”)

- A tester observes the following software behaviour:
  \[ \pi = \{ s \mapsto \text{NULL}, r \mapsto 0 \} \xrightarrow{\tau} \text{program-abortion} \]

- Test execution **positive** or **negative**?

**Note:**
- If a tester does not adhere to an allowed input sequence of \( T \), \( \pi \) is not a test execution.
  Thus \( \pi \) is neither positive nor negative (only defined for test executions).
- Same case: power outage (if continuous power supply is considered in input sequence).
Tests vs. Systematic Tests

**Test** – (one or multiple) execution(s) of a program on a computer with the goal to find errors. *(Ludewig and Lichter, 2013)*

Not (even) a test (in the sense of this weak definition):

- any **inspection** of the program (no execution),
- **demo** of the program (other goal),
- analysis by software-tools for, e.g., values of **metrics** (other goal),
- **investigation** of the program with a debugger (other goal).

**Systematic Test** – a test such that

- (environment) conditions are defined or precisely documented,
- inputs have been chosen systematically,
- results are documented and assessed according to criteria that have been fixed before. *(Ludewig and Lichter, 2013)*

(Our) Synonyms for non-systematic tests: Experiment, ‘Rumprobieren’.

In the following: **test** means systematic test; if not systematic, call it **experiment**.
So Simple?
**Environmental Conditions**

**Strictly speaking**, a test case is a triple \((In, Soll, Env)\) comprising a description \(Env\) of (environmental) conditions.

\(Env\) describes any aspects which could have an effect on the outcome of a test execution and cannot be specified as part of \(In\), such as:

- Which program (version) is tested?
- Built with which compiler, linker, etc.?
- Test host (OS, architecture, memory size, connected devices (configuration?), etc.)?
- Which other software (in which version, configuration) is involved?
- Who is supposed to test when?
- etc. etc.

→ test executions should be (as) **reproducible** and **objective** (as possible).

**Full reproducibility** is hardly possible in practice – obviously (err, why…?).

- **Steps** towards reproducibility and objectivity:
  - have a fixed build environment,
  - use a fixed test host which does not do any other jobs,
  - execute test cases **automatically** (test scripts).
Content (Part II: Code Quality Assurance)

- **Introduction**
  - quotes on testing,
  - systematic testing vs. ‘rumprobieren’

- **Test Case**
  - definition,
  - execution,
  - positive and negative.

- **Test Suite**

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range vs. point errors

- **More Vocabulary**
The Limits of Software Testing
In each examination, there are two paths from the specification to results:

- the production path (using model, source code, executable, etc.), and
- the examination path (using requirements specifications).

A check can only discover errors on exactly one of the paths.

If a difference is detected: examination result is positive.

What is not on the paths, is not checked; crucial: specification and comparison.

Recall:

<table>
<thead>
<tr>
<th></th>
<th>checking procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shows no error</td>
</tr>
<tr>
<td></td>
<td>reports error</td>
</tr>
<tr>
<td>yes</td>
<td>false negative</td>
</tr>
<tr>
<td></td>
<td>true positive</td>
</tr>
<tr>
<td>no</td>
<td>true negative</td>
</tr>
<tr>
<td></td>
<td>false positive</td>
</tr>
</tbody>
</table>

(Ludewig and Lichter, 2013)
"Software testing can be used to show the presence of bugs, but never to show their absence!"

(E. W. Dijkstra, 1970)
Why Can’t We Show The Absence of Errors (in General)?

Consider a simple pocket calculator for adding 8-digit decimals:

- **Requirement**: If the display shows \(x\), \(+\), and \(y\), then after pressing \(=\),
  - the sum of \(x\) and \(y\) is displayed if \(x + y\) has at most 8 digits,
  - otherwise “-E-” is displayed.
- With 8 digits, both \(x\) and \(y\) range over \([0, 10^8 - 1]\).
- Thus there are \(10^{16} = 10,000,000,000,000,000\) possible input pairs \((x, y)\) to be considered for exhaustive testing, i.e. testing every possible case!
- And if we restart the pocket calculator for each test, we do not know anything about problems with sequences of inputs...
  (Local variables may not be re-initialised properly, for example.)
**Observation: Software Usually Has Many Inputs**

- **Example:** Simple Pocket Calculator.

  With **ten thousand** (10,000) **different** test cases (that’s a lot!),
  9,999,999,999,990,000 of the $10^{16}$ possible inputs remain **uncovered**.

  **In other words:**
  Only 0.0000000001% of the possible inputs are covered, 99.9999999999% not touched.
Observation: Software Usually Has Many Inputs

- **Example:** Simple Pocket Calculator.

  With ten thousand \((10,000)\) different test cases (that's a lot!), 9,999,999,999,990,000 of the \(10^{16}\) possible inputs remain **uncovered**.

  **In other words:**
  Only 0.0000000001% of the possible inputs are covered, 99.9999999999% not touched.

- **In diagrams:** (red: uncovered, blue: covered)
Software is (in general) **not continuous**.

Consider a continuous function, e.g. the one to the right:

For sufficiently small $\varepsilon$-environments of an input, the outputs **differ only by a small amount** $\delta$.

Physical systems are (to a certain extent) continuous:
- For example, if a bridge endures a single car of 1000 kg, we strongly expect the bridge to endure cars of 990 kg or 1010 kg.
- And anything of weight smaller than 1000 kg can be expected to be endured.

For software, adjacent inputs may yield **arbitrarily distant** output values.

**Vocabulary**:
- **Point error**: an isolated input value triggers the error.
- **Range error**: multiple “neighbouring” inputs trigger the error.

For software, (in general, without extra information) we can not **conclude from some values to others**.
Content (Part II: Code Quality Assurance)

- Introduction
  - quotes on testing,
  - systematic testing vs. ‘rumprobieren’

- Test Case
  - definition,
  - execution,
  - positive and negative.

- Test Suite

- Limits of Software Testing
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range vs. point errors

- More Vocabulary
Tell Them What You’ve Told Them…

- Testing is about
  - finding errors, or
  - demonstrating scenarios.

- A test case consists of
  - input sequences and
  - expected outcome(s).

- A test case execution is
  - positive if an error is found,
  - negative if no error is found.

- A test suite is a set of test cases.
References
References


