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

Topic Area Architecture & Design: Content

- 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
Principles of (Architectural) Design

- Principles of (Good) Design Contd
  - 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
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.
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 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.


**Task:** store a list of names in $N$ of type “list of string”.

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

- **insert** (string $n$);
  - **pre-condition:**
    
    \[ N = n_0, \ldots, n_i, n_{i+1}, \ldots, n_m - 1, m \in \mathbb{N}, 0 \leq j < m \wedge n_j < n_{j+1} \]
  - **post-condition:**
    
    \[ N = n_0, \ldots, n_i, n_i, n_{i+1}, \ldots, n_m - 1, m \in \mathbb{N}, 0 \leq i < m \]

- **remove** (int $i$);
  - **pre-condition:**
    
    \[ N = n_0, \ldots, n_{i-1}, n_i, n_i, n_{i+1}, \ldots, n_m - 1, m \in \mathbb{N}, 0 \leq i < m \]
  - **post-condition:**
    
    \[ N = n_0, \ldots, n_i, n_i, n_{i+1}, \ldots, n_m - 1 \]

- **get** (int $i$): string;
  - **pre-condition:**
    
    \[ N = n_0, \ldots, n_{i-1}, n_i, n_i, n_{i+1}, \ldots, n_m - 1, m \in \mathbb{N}, 0 \leq i < m \]
  - **post-condition:**
    
    \[ N = \text{old}(N), \text{retval} = n_i \]

- **d dump();**
  - **pre-condition:**
    
    \[ N = n_0, \ldots, n_m - 1, m \in \mathbb{N} \]
  - **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("Naumann");
  dump();
  remove(1);
  insert("Mayer");
  dump();
  names[2] = "Naumann";
  dump();
  return 0;
}
```

Output:

Berger
Mayer
Neumann
Schulz
Wernersen
Wernersen

---

A possible implementation of a list of names in C++ using a `std::vector` and handling duplicates.
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( "Bergen" );
    insert( "Schulz" );
    insert( "Neumann" );
    insert( "Meyer" );
    insert( "Wernersen" );
    dump();
    remove( 1 );
    insert( "Mayer" );
    dump();
    names[2] = "Naumann";
    dump();
    return 0;
}
```

Output:

1. Bergen
2. Meyer
3. Neumann
4. Schulz
5. Wernersen
6. Berger
7. Mayer
8. Neumann
9. Schulz
10. Wernersen

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**: $N = n_0, \ldots, n_i, n_{i+1}, \ldots, n_{m-1}, m \in \mathbb{N}, 0 \leq j < m \land n_j < \text{lex } n_{j+1}$
    - **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 $0 \leq i < m, N = \text{old}(N), \text{count}(n) = \text{old}(\text{count}(n)) + 1$.
  - `remove(int i);`
    - **pre-condition**: $N = n_0, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_{m-1}, m \in \mathbb{N}, 0 \leq i < m$,
    - **post-condition**: 
      - if $\text{count}(n_i) = 1, N = n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_{m-1}$.
      - if $\text{count}(n_i) > 1, N = \text{old}(N), \text{count}(n_i) = \text{old}(\text{count}(n_i)) - 1$.
  - `get(int i): string;` and `dump();` 
    - unchanged contract
Changed Implementation: Support Duplicates

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

void insert(const 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.insert(count.begin()) + (it - names.begin()) + 1;
    }
    names.insert(it, n);
}

void remove(int i) {
    if (--count[i] == 0) {
        names.erase(names.begin() + i);
    } else {
        ++count[i] + (it - names.begin()) + 1;
    }
}

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
Mayer: 1
Neumann: 2
Schulz: 5
Wernersen: 1
```

access is bypassing the interface – and corrupts the data-structure
```cpp
#include <vector>
#include <string>

struct Names {
  std::vector<std::string> count;
  std::vector<std::string> names;

  Names();
  ~Names();
  void dump();
  void insert(std::string n);
  void remove( std::string get( int i );
  int lower_bound();
}

int main() {
  Names names;
  std::vector<std::string> names;
  names = { "Neumann", "Mayer", "Wernerсен", "Schulz", "Bergen" };
  int n = names.size();
  int i = 0;
  int count = 0;

  while (i < n)
  {
    names.insert( names.begin() + lower_bound(), names[i]);
    count += 1;
    i += 1;
  }

  return 0;
}
```

Output:
```cpp
#include "mod_oo.h"

int main() {
  Names names = new Names();
  names.insert("Bergen");
  names.insert("Schulz");
  names.insert("Mayer");
  names.insert("Wernerсен");
  names.insert("Neumann");
  names.insert("Neumann");
  names.insert("Mayer");
  names.insert("Wernerсен");
  names.insert("Bergen");
  names.remove(1)
  names.insert("Mayer");
  names.remove(1)
  names.insert("Naumann");
  names.remove(1)
  names.dump();
  return 0;
}
```

Output:
- Bergen: 1
- Mayer: 1
- Neumann: 2
- Schulz: 1
- Wernerсен: 1

Access is bypassing the interface and corrupts the data-structure.
# Data Encapsulation / Information Hiding

```cpp
#include "mod_oo_deih.h"

class Names {

private:
  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"

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.begin() ) + ( it - names.begin() )
  } else {
    count.insert( count.end() ) + ( it - names.begin() );
  }
}

test main() {
  Names names = new Names();
  names.insert( "Berger" );
  names.insert( "Schulz" );
  names.insert( "Naumann" );
  names.insert( "Mayer" );
  names.insert( "Wernersen" );

  names.dump();
  names.remove( 1 );
  names.insert( "Mayer" );
  names.remove( 2 );
  names.insert( "Naumann" );

  names.dump();
  names.remove( 1 );
  names.insert( "Mayer" );
  names.remove( 1 );
  names.insert( "Maeyer" );

  names.dump();
  names.remove( 2 );
  names.insert( "Maeyer" );
  names.remove( 1 );
  names.insert( "Wernersen" );

  names.dump();
  names.remove( 2 );
  names.insert( "Naumnn" );

  names.dump();
  return 0;
}
```

**Output:**

```
1  Berger: 1
2   Mayer: 1
3  Naumann: 2
4    Schulz: 2
5   Wernersen: 1
6
7  Berger: 1
8   Mayer: 1
9  Naumann: 2
10   Schulz: 1
11  Wernersen: 1
12
13 0
```
(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 (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
Design Approaches
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)

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.
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.
- **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.
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.
Example: Pipe-Filter

Example: Compiler

Example: UNIX Pipes

ls -1 | 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.
Example: Model-View-Controller

controller
uses
sees
change of visualisation
notification of updates
manipulation of data
access to data

view
model

https://commons.wikimedia.org/wiki/File:Maschinenleitstand_KWZ.jpg
Dergenaue, CC-BY-SA-2.5
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

- 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: Strategy

<table>
<thead>
<tr>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
</tr>
<tr>
<td>The only difference between similar classes is that they solve the same problem by different algorithms.</td>
</tr>
</tbody>
</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
  + contextInterface()   + algorithm()

ConcreteStrategy1
  + algorithm()

ConcreteStrategy2
  + algorithm()
```

---

**Strategy**

**Problem:** The only difference between similar classes is that they solve the same problem by different algorithms.

**Solution:**

**Structure**

- **Strategy Context**
  - contextInterface()
  - strategy()

- **ConcreteStrategy1**
  - algorithm()

- **ConcreteStrategy2**
  - algorithm()

---

Example: Pattern Usage and Documentation

Observer

**Problem**
Multiple objects need to adjust their state if one particular other object is changed.

**Example**
All GUI objects displaying a file system need to change if files are added or removed.

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.

Mediator

<table>
<thead>
<tr>
<th>Problem</th>
<th>Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</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

Singleton

<table>
<thead>
<tr>
<th>Problem</th>
<th>Of one class, exactly one instance should exist in the system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Print spooler.</td>
</tr>
</tbody>
</table>

Memento

<table>
<thead>
<tr>
<th>Problem</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Undo mechanism.</td>
</tr>
</tbody>
</table>
The development of design patterns is considered to be one of the most important innovations of software engineering in recent years.

[Luwendig 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

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

- **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
### 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**
Quotes On Testing

"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 $[S]$ of (finite or infinite) computation paths of the form \( \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \) where
- $\sigma_i \in \Sigma, i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A, i \in \mathbb{N}_0$, is called action (or event).
The (possibly partial) function $[\cdot] : S \mapsto [S]$ is called interpretation of $S$.

- From now on, we assume that states consist of an input and an output/internal part, i.e., there are $\Sigma_{\text{in}}$ and $\Sigma_{\text{out}}$ such that $\Sigma = \Sigma_{\text{in}} \times \Sigma_{\text{out}}$.
- Computation paths are then of the form $\pi = \left( \begin{array}{c} \sigma_0 \\ \sigma_0' \end{array} \right) \xrightarrow{\alpha_1} \left( \begin{array}{c} \sigma_1 \\ \sigma_1' \end{array} \right) \xrightarrow{\alpha_2} \cdots$
- We use $\pi \downarrow \Sigma_{\text{in}}$ to denote $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \cdots$, i.e. the projection of $\pi$ onto $\Sigma_{\text{in}}$.

**Test Case**

**Definition.** A test case $T$ over $\Sigma$ and $A$ is a pair $(\text{In}, \text{Soll})$ consisting of
- a description In of sets of finite input sequences,
- a description Soll of expected outcomes,
and an interpretation $[\cdot]$ of these descriptions:
- $[\text{In}] \subseteq (\Sigma_{\text{in}} \times A)^*$, \quad $[\text{Soll}] \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$
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 $[\cdot]$ of these descriptions:
- $[In] \subseteq (\Sigma_{in} \times A)^*$, $[Soll] \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$.

Examples:
- Test case for procedure `strlen`: String $\rightarrow$ N, $s$ denotes parameter, $r$ return value:
  
  $T = (s = "abc", r = 3)$

  $[s = "abc"] = \{ \sigma_0 \xrightarrow{t} \sigma_1 | \sigma_0(s) = "abc" \}$, $[r = 3] = \{ \sigma_0 \xrightarrow{t} \sigma_1 | \sigma_1(r) = 3 \}$.

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

- “Call `strlen()` with string "abc", expect return value 3.”

- Test case for vending machine:
  
  $T = (C50, WATER; DWATER)$

  $[C50, WATER] = \{ \sigma_0 \xrightarrow{C50} \sigma_1 \xrightarrow{WATER} \cdots \xrightarrow{WATER} \sigma_j \}$,
  $[DWATER] = \{ \sigma_0 \xrightarrow{\alpha_k} \cdots \xrightarrow{\alpha_{k-1}} \sigma_k \xrightarrow{DWATER} \sigma_k | k \leq 10 \}$.

- “Send event $C50$ and any time later $WATER$, expect $DWATER$ 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:

$\llbracket In \rrbracket \subseteq (\Sigma_{in} \times A)^*$, \hspace{1em} $\llbracket Soll \rrbracket \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^\omega$

**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
  
  $\pi = \left( \sigma_0^\ell, \alpha_1, \sigma_1^\ell, \alpha_2, \ldots \right) \llbracket In \rrbracket$
  
  from $\llbracket S \rrbracket$ is called **execution** of test case $(In, Soll)$ if and only if

  - there is $n \in \mathbb{N}$ such that $\sigma_0^\ell \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \sigma_n \downarrow \Sigma_{in} \in \llbracket In \rrbracket$.

  ("A prefix of $\pi$ corresponds to an input sequence").

Execution $\pi$ of test case $T$ is called

- **successful** (or **positive**) if and only if $\pi \notin \llbracket Soll \rrbracket$.
  - 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 $\pi \in \llbracket Soll \rrbracket$.
  - Intuition: no error has been discovered.
  - Alternative: test item $S$ **passed the test**.
  - Okay: "test passed".
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:
  \[ \pi = \{ s \mapsto \text{NULL}, r \mapsto 0 \} \]
  \[ \xrightarrow{e_{10}} \text{program-abortion} \]

- 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**
Software Examination (in Particular 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:

- Checking procedure shows no error: reports no error.

   - true negative
   - true positive

   - false negative
   - false positive

(Ludewig and Lichter, 2013)
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 0.00000000001% of the possible inputs are covered, 99.9999999999% not touched.

- In diagrams: (red: uncovered, blue: covered)
Point vs. Range Errors

- 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**
• 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