Softwaretechnik / Software-Engineering

Lecture 15: Architecture and Design Patterns

2015-07-04

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

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

VL 11
- **Introduction and Vocabulary**
- **Principles of Design**
  - (i) modularity
  - (ii) separation of concerns
  - (iii) information hiding and data encapsulation
  - (iv) abstract data types, object orientation

⋮
- **Software Modelling**
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)
  - (iv) **modelling structure** (VL 12 ⋮)
    - a) (simplified) class diagrams
    - b) (simplified) object diagrams
    - c) (simplified) object constraint logic (OCL)
  - (v) **modelling behaviour** (VL 13 ⋮)
    - a) communicating finite automata
    - b) Uppaal query language
    - c) implementing CFA (VL 14 ⋮)
    - d) an outlook on **UML State Machines**

VL 15
- **Design Patterns**

⋮
- **Testing**: Introduction

Content (Part I)

- **Architecture Patterns**
  - Layered Architectures,
  - Pipe-Filter,
  - Model-View-Controller.
- **Design Patterns**
  - Strategy,
  - Observer, State, Mediator,
  - Singleton, Memento.
  - Inversion of control.
- **Libraries and Frameworks**
- **Quality Criteria on Architectures**
  - Development Approaches,
  - Software Entropy.

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

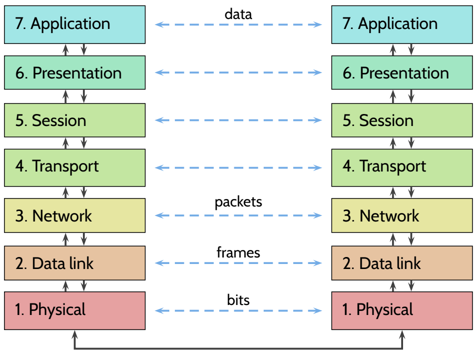
Example: Layered Architectures

- (Züllighoven, 2005):

  A **layer** whose components only interact with components of their **direct neighbour** layers is called **protocol-based** layer.

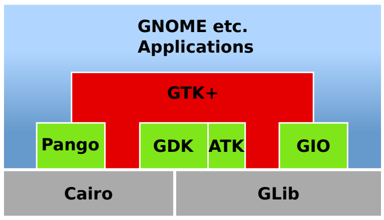
  A **protocol-based layer** hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.

- **Example: The ISO/OSI reference model.**



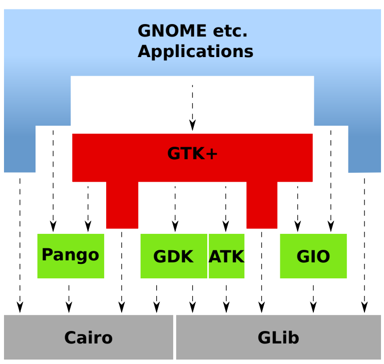
Example: Layered Architectures Cont'd

- **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: 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.



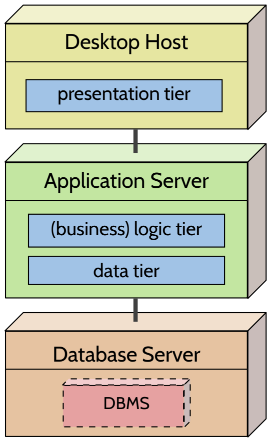
Example: Three-Tier Architecture

- **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**:

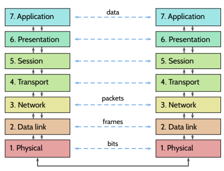
  persistent data storage; hides information about how data is organised, read, and written, offers particular chunks of information in a form useful for the logic layer.

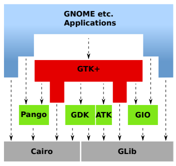


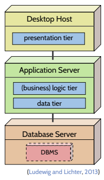
(Ludewig and Lichter, 2013)

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

Layered Architectures: Discussion





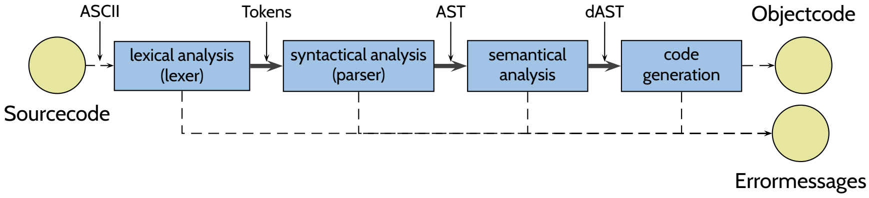


- **Advantages**:
  - **protocol-based**:
    only neighouring layers are coupled, i.e. components of these layers interact,
  - coupling is low, data usually encapsulated,
  - changes have local effect (only neighbouring layers affected),
  - **protocol-based**: **distributed** implementation often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

Pipe-Filter

Example: Pipe-Filter

Example: **Compiler**



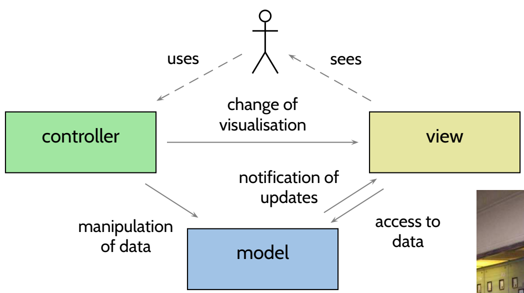
Example: **UNIX Pipes**

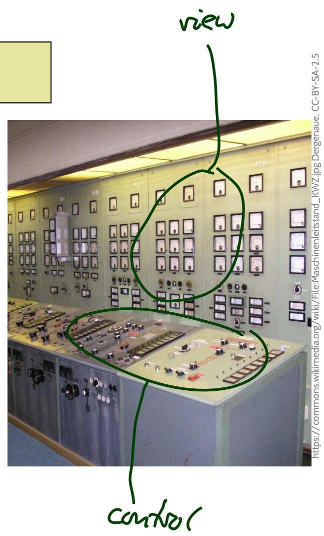
```
ls -l | grep Sarch.tex | awk '{ print $5 }'
```

- **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.

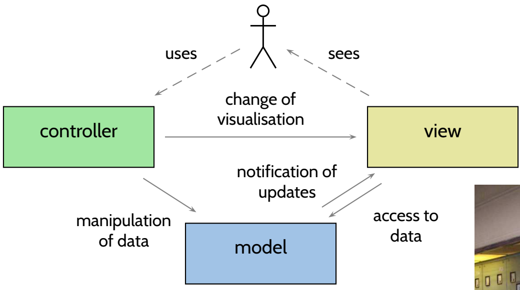
Model-View-Controller

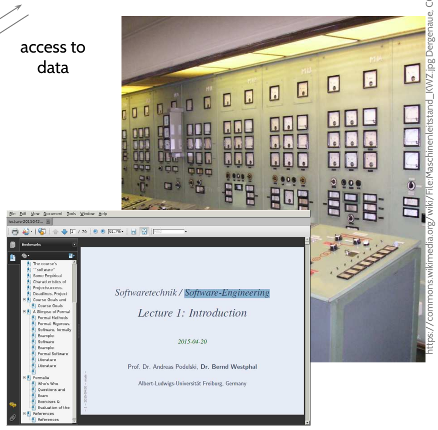
Example: Model-View-Controller





Example: Model-View-Controller



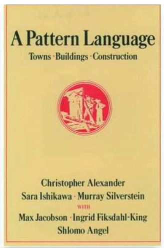


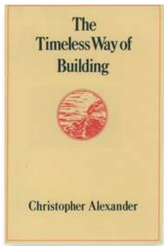
- **Advantages**:
  - one model can serve multiple view/controller pairs;
  - view/controller pairs can be added and removed at runtime;
  - model visualisation always up-to-date in all views;
  - distributed implementation (more or less) easily.
- **Disadvantages**:
  - if the view needs **a lot of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

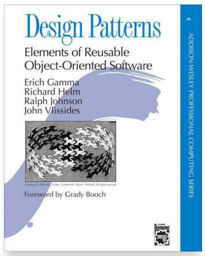
Design Patterns

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).

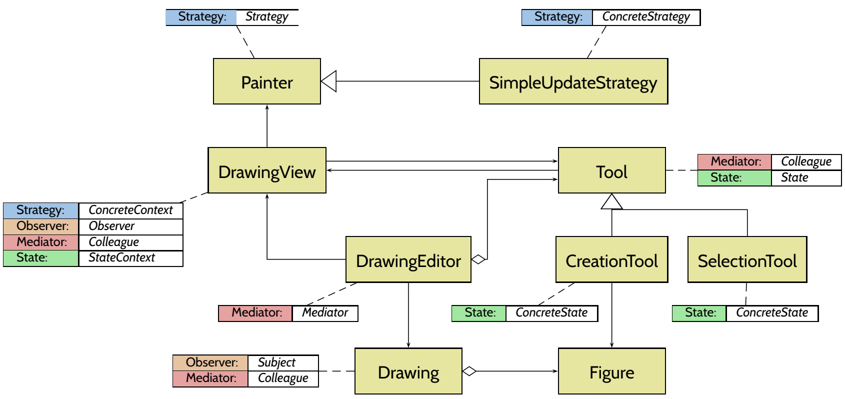






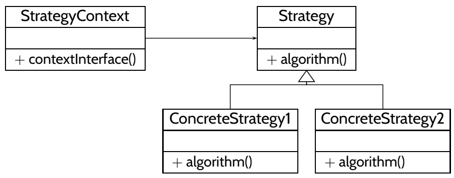
> **Design patterns** ... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
>
> A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable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

Example: Pattern Usage and Documentation

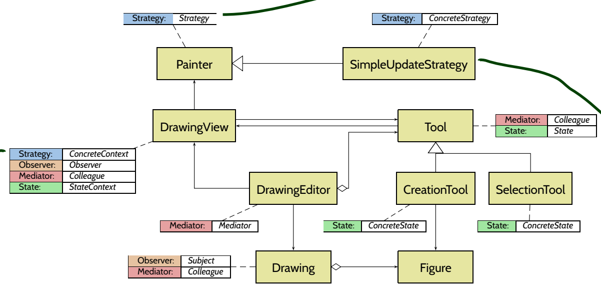


Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

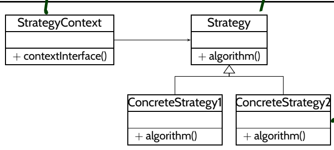
Example: Strategy

	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	

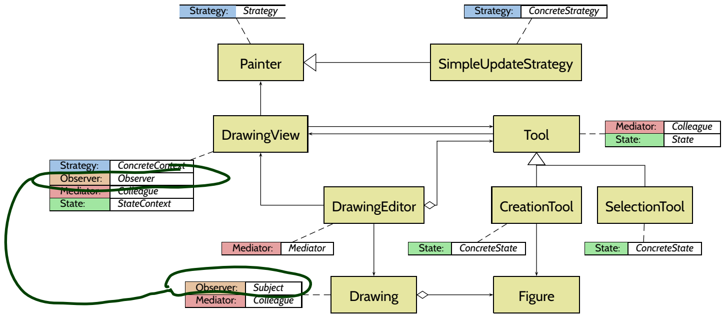
Example: Pattern Usage and Documentation



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

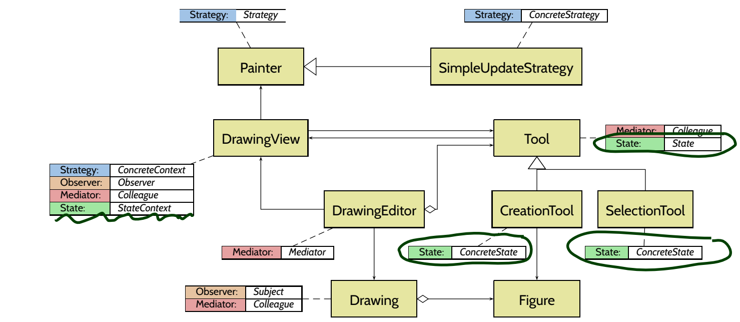


Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	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.

– 15 - 2015-07-04 - Sdespat –

Example: Pattern Usage and Documentation

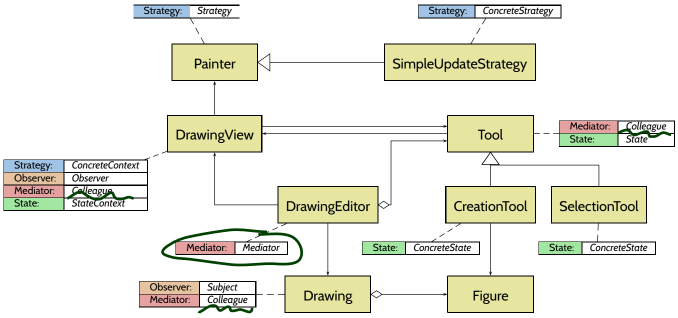


Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

	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.

– 15 - 2015-07-04 - Sdespat –

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.

Meta Design Pattern: Inversion of Control

"**don't call us, we'll call you**"

- **User interfaces**, for example:
  - define `button_callback();`
  - register method with UI-framework (→ later),
  - whenever button is pressed (handled by UI-framework), `button_callback()` is called and does its magic.

- Also found in **MVC** and **observer** patterns:
  model notifies view, subject notifies observer.

vs.

- **Classical** (small) embedded controller software:
  - ```
    while (true) {
        // read inputs
        // compute updates
        // write outputs
    }
    ```


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

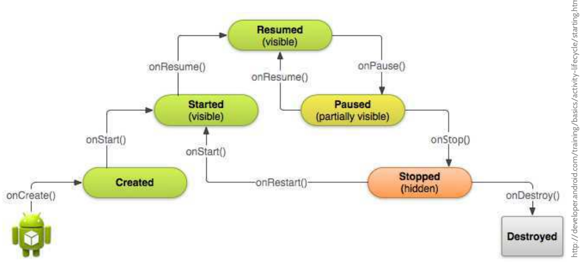
- **(Class) Library**:
  a collection of operations or classes offering generally usable functionality in a re-usable way.

  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.

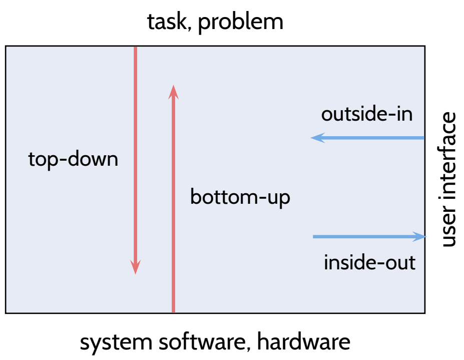
  Examples:
  - `libc` – standard C library (is in particular abstraction layer for operating system functions),
  - `GMP` – GNU multi-precision library, cf. Lecture 6.
  - `libz` – compress data.
  - `libxml` – read (and validate) XML file, provide DOM tree.
- **Framework**: class hierarchies which determine 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").
- For some application domains, there are **reference architectures** (games, compilers).

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**.

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.

Software Entropy

- **Lehman**'s Laws of Software Evolution (Lehman and Belady, 1985):
  (i) A program that is used **will be modified**.
  (ii) When a program is modified, its **complexity will increase**, provided that one does not **actively work against** this.

- (Jacobson et al., 1992): **Software entropy** E (measure of disorder), **claim**:

$$\Delta E \sim E$$

  - "when designing a system with the intention of it being maintainable, we try to give it the lowest software entropy possible from the beginning."
  - Work against disorder: **re-factoring**

    (re-assign data and operations to modules, introduce new layers generalising old and new solutions, (automatically) check that intended interfaces are not bypassed, etc.)

- Proposal (Jacobson et al., 1992):
  - use "probability for change" as guideline in (architecture) design,
  - i.e. base design on a thorough analysis of problem and solution domain.

item	*probability for change*
Object from application [domain]	Low
Long-lived information structures	Low
Passive object's attribute	Medium
Sequences of behaviour	Medium
Interface with outside world	High
Functionality	High

– 15 – 2015-07-04 – Sdesq –

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.
- Mind **Lehman's Laws** and **software entropy**.

– 15 – 2015-07-04 – Sttwytt –

Code Quality Assurance

Content (Part II)

- **Introduction**
  - quotes on testing,
  - systematic testing vs. 'rumprobieren'.
- **Test Case**
  - definition,
  - execution,
  - **positive** and **negative**.
- The **Specification** of a Software
- **Test Suite**
- More **Vocabulary**

Testing: Introduction

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)

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)

(Our) Synonyms: Experiment, 'Rumprobieren'.

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

- any inspection of the program,
- demo of the program,
- analysis by software-tools for, e.g., values of metrics,
- investigation of the program with a debugger.

> **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)

In the following: **test means** systematic test; if not systematic, call it **experiment**.

More Formally: Test Case

> **Definition.** A **test case** T 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.

Plus, **strictly speaking**, for each pair a description Env of **(environmental) conditions**:, i.e., any aspects which **could have an effect** on the outcome of the test 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-cases should be (as) **reproducible** and **objective** (as possible).

Note: **inputs** can be

- input data, possibly with timing constraints,
- other interaction, e.g., from network,
- initial memory content,
- etc.

Test Case Cont'd

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).

Executing Test Cases: Preliminaries

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 Σ_{in} and Σ_{out} such that

$$\Sigma = \Sigma_{in} \times \Sigma_{out}.$$

- **Computation paths** are then of the form

$$\pi = \begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow{\alpha_1} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix} \xrightarrow{\alpha_2} \cdots$$

$\in \Sigma = \Sigma_{in} \times \Sigma_{out}$

Executing Test Cases

- A computation path

$$\pi = \begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow{\alpha_1} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix} \xrightarrow{\alpha_2} \cdots$$

  from $[\![S]\!]$ is called **execution** of test case $(In, Soll)$
  if and only if there is $n \in \mathbb{N}_0$ such that $\sigma_0^i, \sigma_1^i, \dots, \sigma_n^i \in [\![In]\!]$.

- π is called **successful** (or **positive**) if it discovered an error, i.e., if $\pi \notin [\![Soll]\!]$.
  (Alternative: test item S **failed to pass test**; confusing: "test failed".)
- π is called **unsuccessful** (or **negative**) if it did not discover an error, i.e., if $\pi \in [\![Soll]\!]$.
  (Alternative: test item S **passed test**; okay: "test passed".)

- **Note**: if input sequence not adhered to, or power outage, etc., π is **not** (even) a test execution.

Test Case Example

- Software S is the **Java program**:

```
public int successor( int x ) { x = x + 1;   return x; }
```

- Assume that $[\![S]\!]$ just considers call and return, i.e. computation paths are of the form

$$\begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow{\tau} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix}$$

  $\sigma_0^i(x)$ is the input value for x and $\sigma_1^o(ret)$ is the **return value**.

- Example **test case**: $(In, Soll) = (27, 28)$ denoting

$$[\![27]\!] := \{\sigma_0^i(x) = 27\} \qquad [\![28]\!] := \left\{ \begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow{\tau} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix} \,\middle|\, \sigma_1^o(ret) = 28 \right\}.$$

- Then

$$\pi = \begin{pmatrix} x = 27 \\ ret = 0 \end{pmatrix} \xrightarrow{\tau} \begin{pmatrix} x = 28 \\ ret = 28 \end{pmatrix}$$

  is an **execution** of $(In, Soll)$.

- Is π **successful** or **unsuccessful**?

The Specification of a Software

- Same software S:

```
public int successor( int x ) { x = x + 1;   return x; }
```

- **Assume** 16-bit `int`, i.e. value of x is in $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$.
- Test case $(In, Soll) = (32767, 32768)$.
- What will S compute?

$$\pi = \begin{pmatrix} x = 32767 \\ ret = 0 \end{pmatrix} \xrightarrow{\tau} \begin{pmatrix} x = \cdot \\ ret = \end{pmatrix}$$

-32768 or -1 / -1 or -32768

- Is π **successful** or **unsuccessful**?
- Well, we operated S **outside** its **specification**:
  - $successor($ `int` $x\ );$
    - **pre-condition**: $x < 32767$
    - **post-condition**: $ret = old(x) + 1$

  If an input does not satisfy the **pre-condition**, S may do "whatever it wants".
  Its behaviour is **not specified** in that case (aka. **chaos**).
- Test cases are usually supposed to test that the software satisfies its **specification**.

By The Way...

- **High quality software** should be aware of its specification.
- `successor()` should check its inputs
  and "**complain**" if operated outside of specification, e.g.
  - throw an exception,
  - abort program execution,
  - (at least) print an error message,
  - etc.
- **Not**: "garbage in, garbage out"

*Wait, Why a **Set** of Inputs. . . ?*

Definition. A **test case** T 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.

- Sometimes, a test case provides a **degree of freedom** or **choices** to the person who conducts the tests.
- For example, for the vending machine

$$In = C50, WATER$$

could specify

> "At **some time** after switching on the vending machine, insert a **50 cent coin**, and **some time** later **request water**."

without fixing these **times**, thus there are many valid input sequences.

Test Suite

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

Testing Vocabulary

Specific Testing Notions

- How are the test cases **chosen**?
  - Considering only the specification (**black-box** or **function** test).
  - Considering the structure of the test item (**glass-box** or **structure** test).

- How much **effort** is put into testing?

  execution trial – does the program run at all?

  throw-away-test – invent input and judge output on-the-fly (→ "**rumprobieren**),

  systematic test – somebody (not author!) derives test cases, defines input/soll, documents test execution.

  In the long run, **systematic tests** are more **economic**.

- **Complexity** of the test item:

  unit test – a single program unit is tested (function, sub-routine, method, class, etc.)

  module test – a component is tested,

  integration test – the interplay between components is tested.

  system test – tests a whole system.

Specific Testing Notions Cont'd

- Which **property** is tested?

  function test –
  functionality as specified by the requirements documents,

  installation test –
  is it possible to **install** the software with the provided documentation and tools?

  recommissioning test –
  is it possible to **bring the system back to operation** after operation was stopped?

  availability test –
  does the system run for the required amount of time without issues,

  load and stress test –
  does the system behave as required under **high or highest load**? ... under overload?
  "Hey, let's try how many game objects can be handled!" – that's an experiment, not a test.

  regression test –
  does the new version of the software **behave like the old one**
  on inputs where no behaviour change is expected?

  resource tests –
  response time, minimal **hardware (software) requirements**, etc.

Specific Testing Notions Cont'd

- Which roles are **involved** in testing?
  - **inhouse test** –
    only developers (meaning: quality assurance roles),
  - **alpha** and **beta** test –
    selected (potential) customers,
  - **acceptance test** –
    the customer tests whether the system (or parts of it, at milestones) test whether the system is acceptable.

A First Rule-of-Thumb

- **How to choose test cases**?
  - "Everything, which is required, **must** be examined/checked. Otherwise it is **uncertain** whether the requirements have been **understood** and **realised**." (Ludewig and Lichter, 2013)
  - In other words:

    Not having at least one (systematic) test case for each (required) feature is (**grossly**?) **negligent** (Dt.: (grob?) fahrlässig).
  - In even other words:

    Without at least one test case for each feature, we can hardly speak of software **engineering**.

– 15 – 2015-07-04 – Stestvoc –

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.
- Distinguish (among others),
  - **glass-box test**: structure (or source code) of test item available,
  - **black-box test**: structure not available.

– 15 – 2015-07-04 – Sttwytt2 –

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.

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

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

Lehman, M. M. and Belady, L. (1985). *Program Evolution. Process of Software Change.* Academic Press.

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

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