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

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

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

Real-World Example: Users do not write to bank accounts directly, only bank clerks do.

- Information hiding and data encapsulation — when enforced technically (examples later) — usually come at the price of worse efficiency.
- It is more efficient to read a component's data directly than calling an operation to provide the value: there is an overhead of one operation call.
- Knowing how a component works internally may enable more efficient operation.
- Example: if a sequence of data items is stored as a singly-linked list, accessing the data items in list-order may be more efficient than accessing them in reverse order by position.
- Good modules give usage hints in their documentation (e.g. C++ standard library).
- Example: if an implementation stores intermediate results at a certain place, it may be tempting to "quickly" read that place when the intermediate results is needed in a different context.
  - maintenance nightmare — If the result is needed in another context, add a corresponding operation explicitly to the interface.
- Yet with today's hardware and programming languages, this is hardly an issue any more; at the time of (Parnas, 1972), it clearly was.
A Problem of Abstraction: Plane Linkage

1. Data Encapsulation

- Information hiding and data encapsulation by modules.
- Examples:
  - Mathematical functions, transformations
  - Data object modules
  - Functional modules

2. Operations

- Include <iostream>
- Examples:
  - Include data type modules

3. Invoked Functions

- Lower bound (names.begin(), names.end())
- Insertion (names.erase(names.begin() + i))
- Return ≤ 0

4. Real-World Example

- Users do not write to bank accounts directly, only bank clerks do.
- 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.

5. Examples

- Evolution, decomposition, reuse, data encapsulation and information hiding by modules.
- Users do not write to bank accounts directly, only bank clerks do.
- 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.

6. Do not externalize
c o u n t . i n s e r t ( i t , n ) ;

the data-structure

else (*i t != n ) {

t h i s -> c o u n t . i n s e r t ( i t , n ) ;

}

< s t r i n g >

W e r n e r s e n : 1

S c h u l z : 1

Neumann : 1

B e r g e r : 1

Output:

Abstract Data Type
Information hiding and data encapsulation not enforced, → negative effects when requirements change, enforcing information hiding and data encapsulation by modules, abstract data types, object oriented without information hiding and data encapsulation, object oriented with information hiding and data encapsulation.

Principles of (Good) Design

• modularity, separation of concerns
• information hiding and data encapsulation
• abstract data types, object orientation
• . . . by example

Architecture Patterns

• Layered Architectures, Pipe-Filter, Model-View-Controller.

Design 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.
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.

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 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.

  7. Application
  6. Presentation
  5. Session
  4. Transport
  3. Network
  2. Data link
  1. Physical

  • 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.

  GNOME etc. Applications
  • GTK+
  • GDK ATK
  • Cairo
  • GLib
  • GIO
  • Pango
Example: Three-Tier Architecture

- Presentation 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.

Layered Architectures: Discussion

- 7. Application
- 6. Presentation
- 5. Session
- 4. Transport
- 3. Network
- 2. Data link
- 1. Physical

- Advantages:
  - Protocol-based: only neighbour layers are coupled, i.e. components of these layers interact, coupling is low, data usually encapsulated, changes have local effect (only neighbour layers affected), protocol-based: distributed implementation often easy.

- Disadvantages:
  - Performance (as usual) — nowadays often not a problem.

Example: Pipe-Filter

- Controller
- View
- Model

- Model sees
- View uses

- Manipulation of data
- Notification of updates
- Access to data

Example: Model-View-Controller

- Sourcecode
- Objectcode
- Error messages
- ASCII Tokens
- AST
- dAST

Example: Compiler

- Lexical analysis (lexer)
- Syntactical analysis (parser)
- Semantical analysis
- Code generation

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

- Advantages:
  - Advanced data exchange is supported without changing the components of the data exchange system, components can be added/removed without changing the system.

- Disadvantages:
  - 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: Pipe-Filter
Example: Model-View-Controller
– 14 – 2019-07-08 – Smvc – 33/70

- Controller
  - Uses
    - Change of
      - Visualisation
    - Manipulation of
      - Data
  - Notification of
    - Updates
  - Access to data

https://commons.wikimedia.org/wiki/File:Maschinenleitstand_KWZ.jpg

Dergenaue, CC-BY-SA-2.5

Example: Model-View-Controller
– 14 – 2019-07-08 – Smvc – 33/70

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
– 14 – 2019-07-08 – Sdespat – 35/70

- 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
– 14 – 2019-07-08 – Sdespat – 36/70

- Painter
  - SimpleUpdateStrategy
  - DrawingView
- Tool
  - CreationTool
  - SelectionTool
- Drawing
  - Figure

Strategy:
- Strategy
  - Strategy
  - ConcreteStrategy
  - StrategyContext

ConcreteStrategy1
  - algorithm()

ConcreteStrategy2
  - algorithm()

Observer:
- Observer
  - Subject

Mediator:
- Mediator
  - Colleague
  - ConcreteState
  - Mediator
  - ConcreteState

State:
- StateContext
  - State

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

Example: Strategy
– 14 – 2019-07-08 – Sdespat – 37/70

- 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
  - StrategyContext + contextInterface()
    - Strategy + algorithm()
      - ConcreteStrategy1 + algorithm()
      - ConcreteStrategy2 + algorithm()
State.

Fields) in a graphical user interface (GUI) should be consistent in each interaction. Appearance and state of different means of interaction (menus, buttons, input fields) should be consistent in each interaction.

Example: The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.

The behavior of an object depends on its (internal) state.

Problem:

State:

Undo mechanism.

Example:

State:

Algorithms needs to be archived in a way that allows to reconstruct this state without violating the principle of data encapsulation.

Problem:

State:

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

Solution:

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

Example: Pattern usage and documentation.

Diagram:

Other pattern: Singleton and Memento.

Example: Pattern usage and documentation.

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

**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, 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.
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.

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)
• WATER

Send event $1 - k \{ \tau = \sigma \}$

\[
\text{In}\{\text{expected outcomes}\} \text{of test case } A \text{ consisting of input sequences of sets of finite } \Sigma \text{ and an interpretation } \pi \text{ over test case } A \text{ in a description } \{\sigma_1, r, s\} = (\text{a description}) \text{ of the form } \{\sigma, r, s\} \text{ where } \sigma, r, s \in \Sigma.
\]

\[
\text{Computations} \text{ in } \Sigma \text{ part, i.e., return value: } \Sigma \times (\Sigma \cup \Sigma) = (\Sigma \cup \Sigma) \times (\Sigma \cup \Sigma).
\]

\[
\text{Execution the Cases}
\]

\[
\text{Testing the Cases}
\]

\[
\text{Execution the Cases}
\]
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.

Consider the test case \( T = (\emptyset, 0) \) for procedure \( \text{strlen} \). "Empty string has length 0."

A tester observes the following software behaviour:

\[ \pi = \{ s \mapsto \text{NULL}, r \mapsto 0 \} \]

\( \sigma_0 \tau \rightarrow \text{program-abort} \)

\( \sigma_1 \tau \rightarrow \text{program-exec} \)

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).

Testing (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.

Environmental Conditions

Strictly speaking, a test case is a triple \( (\text{In}, \text{Soll}, \text{Env}) \) comprising a description \( \text{Env} \) of (environmental) conditions. \( \text{Env} \) describes any aspects which could have an effect on the outcome of a test execution and cannot be specified as part of \( \text{In} \), 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?

→ 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
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:

1. Checking procedure shows no error
2. Report error
3. Artefact has error
4. Yes
5. False negative
6. True positive
7. No
8. True negative
9. False positive

“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:

12345678
+ 27
7890456
123

- 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 \( 10^{16} = 10^{16} \) 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,
- 99.99999999\% 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:

$\epsilon$ $\delta$

For sufficiently small $\epsilon$-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 cannot conclude from some values to others.

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...