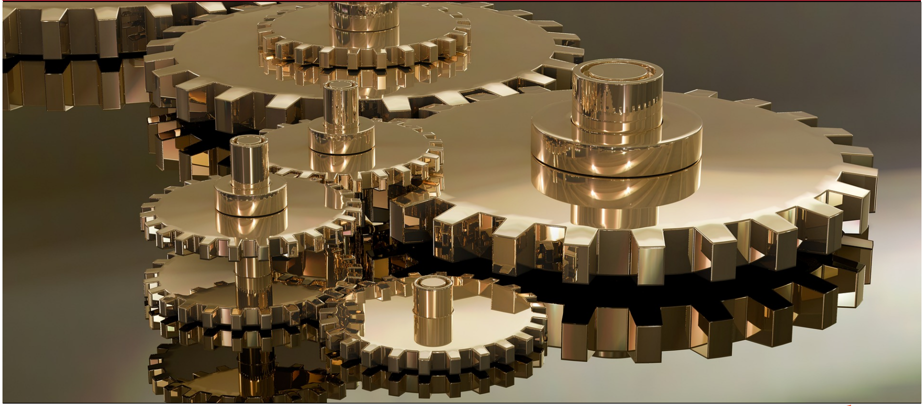
Software Engineering

Design Principles for Software Quality

Prof. Dr. Reiner Hähnle
Fachgebiet Software Engineering



Part I

Dimensions of Software Quality

Software Quality

How to **assure** software quality? The big picture

Define a quality assurance (QA) plan ("Plan zur Qualitätssicherung")
(must be integrated into the software development process)

- Constantly assess design quality (quantitative, qualitative)
- Be proficient in applying time-tested design principles
- Use tools and design techniques that help to achieve quality
- Use design patterns to learn from and reuse proven solutions to recurring problems
- Systematically verify correctness & performance of design
- Validate fulfillment of requirements

First two topics = this lecture, remaining ones next three lectures

Internal and External Quality Factors

What is good software?

Internal quality factors — Perceivable only by computer professionals

- White box view of a software system

External quality factors — Perceived by the customer / user

- Black box view of a software system
- Depend on internal quality factors

Software Quality: Internal Factors

Some important internal quality factors

- Modularity
- Comprehensibility ("Verständlichkeit"), not overly complex
- Cohesion: clear responsibilities
- Concision ("Prägnanz"): little code duplication, clear code
- Correctness

We will talk about how to judge and how to ensure these quality factors

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Validity

The ability of software to perform as defined in its requirements

- Precise requirements definition is needed
- Validity depends on correct design
- Validity is often conditional to correctness of lower layers: Compiler, OS, virtual machine, hardware, etc.

Software Quality: External Factors

TECHNISCHE UNIVERSITÄT DARMSTADT

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Robustness

The ability of software systems to react appropriately to abnormal conditions outside of the operational specification

Software Engineering Group

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Extensibility

Ease of adapting software products to extended requirements

Architecture	Can the architecture be easily adapted?
Modularity	Loose coupling (to be defined precisely) makes changes easy

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Reusability

The ability of software elements to serve for the construction of further applications

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Compatibility / Interoperability

Ease of combining software elements with other applications:

- Compatibility to standards, protocols
- Backwards compatible (to itself)
- Interoperable with other (legacy) applications

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Portability

Ease of transferring software products
to other hardware and software environments

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Efficiency

The ability of a software system to use hardware resources economically

- CPU time/load, internal/external memory usage, power, bandwidth, etc
- Often depends on choice of algorithm (different course)
- Invest in common case: Efficiency often has to be traded off

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Usability

How easily people with different backgrounds and qualifications can learn to use software and apply it to solve problems

Software Quality: External Factors

External quality factors, perceived by Customer or User

- Validity
- Robustness
- Extensibility
- Reusability
- Compatibility / Interoperability
- Portability
- Efficiency
- Usability
- Functionality (meets expectations)

Features

Extent of possible usage modes provided by a system

- Avoid “featurism”: ensure consistency of new features with existing ones

Main Attributes of “Good” Software

Maintainability	Constructed in such a way that it may evolve to meet changing requirements of customers
Efficiency	Does not waste system resources: Processing time, memory utilisation, energy, bandwidth, etc
Usability	Responsiveness, must be usable by the intended users
Dependability	“Verlässlichkeit” Does not cause physical or economic damage in the event of system failure Sub-properties: repairability, survivability, fault tolerance Belongs to system engineering rather than SE

Part II

Quantitative Measures of Software Quality

From Software Quality to Code Quality

How to **measure** software quality?

There is no generally accepted measure of software quality!

- There are many qualities in software (as seen)
  Some of these are at odds to each other (for example, usability—security)
- What we can do:
  Define heuristics that indicate the quality of code
- These heuristics are called software metrics (better: code metrics)

Software Metrics Pro

- Can be computed mechanically
- May indicate bad design

Software Metrics Contra

- No notion of semantics of code
- False sense of correctness

Code Metrics to Identify Anomalies

Some software metrics

Fan in/fan out: Fan-in of method $m = \#$ of methods calling m

Fan-out of method $m = \#$ of method called by m

Length of code: Length of source code (indicator of complexity)

Cyclomatic complexity:

Independent paths through code (in control-flow graph)

Depth of conditional nesting:

Deep nesting hard to understand: Increased test effort

Weighted methods per class:

Method weight depends on size/complexity, summed up per class

Depth of inheritance tree:

Deep inheritance tree: Classes depend on many design constraints

Control-Flow Graph (CFG)

Basic Block

Control-flow graph (CFG) represents all execution sequences of a program *P*

Basic Block (in a CFG)

A basic block is a maximal sequence of non-branching program statements (or instructions) that are always executed together or not at all.
Execution of a basic block starts with the first statement,
only the final statement may branch or return (jump).

```
 a = u − t;
 b = w − v;
 if (a > b) {
   d = a − b;
   w = w + d;
 } else {
   d = b − a;
   u = u + d;
 }
 d = d * d;
```

Basic Block?

No, contains a conditional statement (branching)

Control-Flow Graph (CFG)

Basic Block

Control-flow graph (CFG) represents all execution sequences of a program *P*

Basic Block (in a CFG)

A basic block is a maximal sequence of non-branching program statements (or instructions) that are always executed together or not at all.
Execution of a basic block starts with the first statement,
only the final statement may branch or return (jump).

```
a = u - t;
b = w - v;
if (a > b) {
  d = a - b;
  w = w + d;
} else {
  d = b - a;
  u = u + d;
}
d = d * d;
```

Basic Block?

No, not of maximal length

Control-Flow Graph (CFG)

Basic Block

Control-flow graph (CFG) represents all execution sequences of a program *P*

Basic Block (in a CFG)

A basic block is a maximal sequence of non-branching program statements (or instructions) that are always executed together or not at all.
Execution of a basic block starts with the first statement,
only the final statement may branch or return (jump).

```
a = u - t;
b = w - v;
if (a > b) {
  d = a - b;
  w = w + d;
} else {
  d = b - a;
  u = u + d;
}
d = d * d;
```

Basic Block?

Still not of maximal length

Control-Flow Graph (CFG)

Basic Block

Control-flow graph (CFG) represents all execution sequences of a program *P*

Basic Block (in a CFG)

A basic block is a maximal sequence of non-branching program statements (or instructions) that are always executed together or not at all.
Execution of a basic block starts with the first statement,
only the final statement may branch or return (jump).

```
a = u - t;
b = w - v;
if (a > b) {
  d = a - b;
  w = w + d;
} else {
  d = b - a;
  u = u + d;
}
d = d * d;
```

Basic Block?

Yes (last statement is allowed to branch)
Basic block needs not constitute sub-program

Control-Flow Graph (CFG)

Basic Block

Control-flow graph (CFG) represents all execution sequences of a program *P*

Basic Block (in a CFG)

A basic block is a maximal sequence of non-branching program statements (or instructions) that are always executed together or not at all.
Execution of a basic block starts with the first statement,
only the final statement may branch or return (jump).

```
 a = u - t;
 b = w - v;
 if (a > b) {
   d = a - b;
   w = w + d;
 } else {
   d = b - a;
   u = u + d;
 }
 d = d * d;
```

Basic Block?

All basic blocks of example

Control-Flow Graph (CFG)

Definition

Control-flow graph (CFG) represents all execution sequences of a program P

Control-Flow Graph

The control-flow graph $CFG(P) = (N, E, \text{Label})$ of P is a labeled directed graph, whose nodes $n \in N$ represent the basic blocks of P.

Each edge $e = (n_i, lb, n_j) \in E$ with $n_i, n_j \in N$ and $lb \in \text{Label}$ represents a possible transfer of control from node n_i to node n_j.

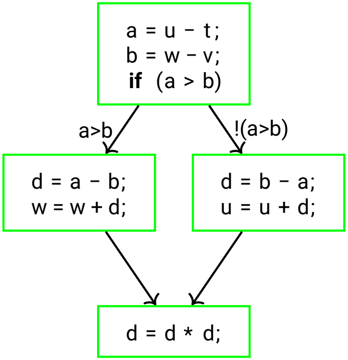
Edge label lb is the branching condition (or empty in case of a return or jump).

Control-Flow Graph (CFG)
Example

Control-flow graph (CFG) represents all execution sequences of a program *P*

```
a = u - t;
b = w - v;
if (a > b) {
  d = a - b;
  w = w + d;
} else {
  d = b - a;
  u = u + d;
}
d = d * d;
```

$\rightsquigarrow$

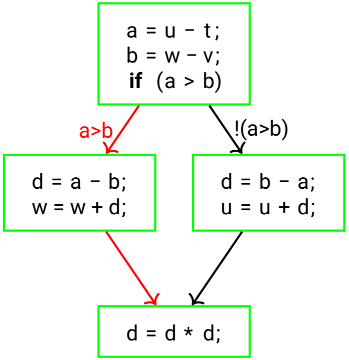


Control-Flow Graph (CFG)

Control-flow graph (CFG) represents all execution sequences of a program *P*

```
a = u - t;
b = w - v;
if (a > b) {
  d = a - b;
  w = w + d;
} else {
  d = b - a;
  u = u + d;
}
d = d * d;
```

$\rightsquigarrow$



Path through *CFG*(*P*) from the initial to an exit node represents one execution sequence of *P*

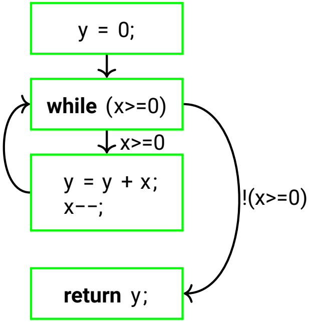
Control-Flow Graph (CFG)

Complex Control Flow

Loops

```
y = 0;
while (x>=0) {
  y = y + x;
  x--;
}
return y;
```

$\rightsquigarrow$



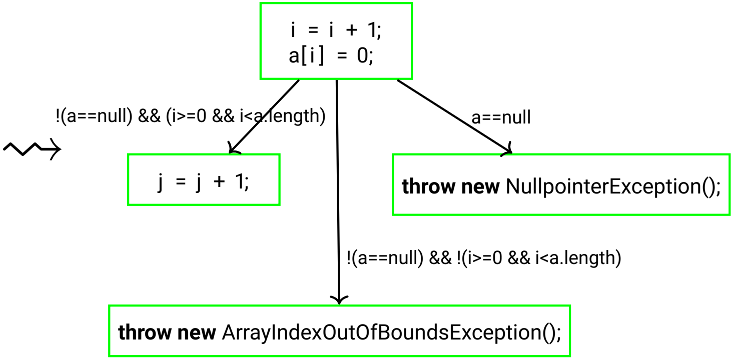
Loop guard not in initial basic block: can be independently executed

Control-Flow Graph (CFG)

Complex Control Flow

Exceptions

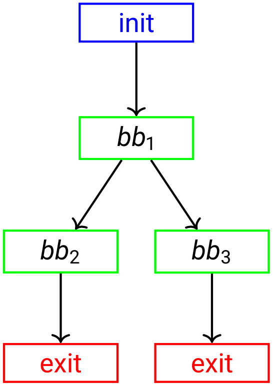
```
i = i + 1;
a[i] = 0;
j = j + 1;
```



i = i + 1;
a[i] = 0;
!(a==null) && (i>=0 && i<a.length)
a==null
j = j + 1;
throw new NullpointerException();
!(a==null) && !(i>=0 && i<a.length)
throw new ArrayIndexOutOfBoundsException();

Basic block ends, whenever a statement can throw an exception (branching)
(Example also illustrates CFG with multiple exit nodes)

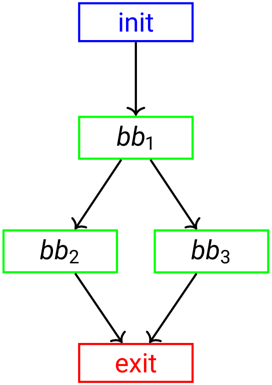
Control-Flow Graph (CFG)
Variant: Explicit Initial / Exit Nodes


$$\begin{tikzcd}
 & \text{init} \arrow[d] & \\
 & bb_1 \arrow[dl] \arrow[dr] & \\
bb_2 \arrow[d] & & bb_3 \arrow[d] \\
\text{exit} & & \text{exit}
\end{tikzcd}$$

Why Explicit initial and exit nodes?

Control-Flow Graph (CFG)

Variant: Explicit Initial / Exit Nodes

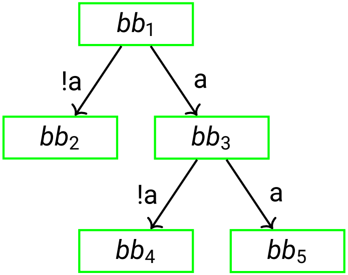

$$\begin{tikzcd}
 & \text{init} \arrow[d] & \\
 & bb_1 \arrow[dl] \arrow[dr] & \\
bb_2 \arrow[dr] & & bb_3 \arrow[dl] \\
 & \text{exit} &
\end{tikzcd}$$

Why Explicit initial and exit nodes?

Some CFG-based analyses require a single exit node

Control-Flow Graph (CFG)

Removal of Inactive Edges

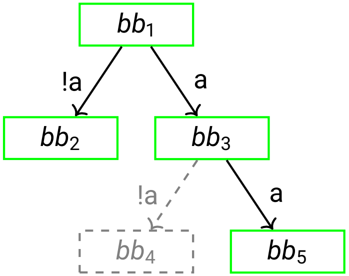

$$\begin{tikzcd}
 & bb_1 \arrow[dl, "!a"'] \arrow[dr, "a"] & & \\
bb_2 & & bb_3 \arrow[dl, "!a"'] \arrow[dr, "a"] & \\
 & bb_4 & & bb_5
\end{tikzcd}$$

For more efficient analysis
$\Rightarrow$ Remove inactive edges
and unreachable nodes

Assumption in this example:
bb_3 does not modify value of a

Control-Flow Graph (CFG)

Removal of Inactive Edges



For more efficient analysis
⇒ Remove inactive edges
and unreachable nodes

Assumption in this example:
bb_3 does not modify value of a

Control-Flow Graph (CFG)

Removal of Inactive Edges

$$\begin{tikzcd}
& \boxed{bb_1} \arrow[dl, "!a"'] \arrow[dr, "a"] & & \\
\boxed{bb_2} & & \boxed{bb_3} \arrow[dr, "a"] & \\
& & & \boxed{bb_5}
\end{tikzcd}$$

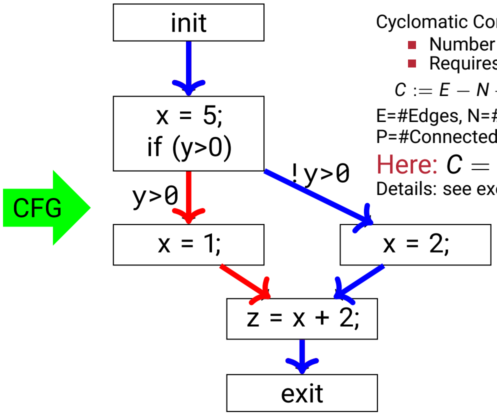
For more efficient analysis
$\Rightarrow$ Remove inactive edges
and unreachable nodes

Assumption in this example:
bb_3 does not modify value of a

Back to code metrics!

Code Metric: Cyclomatic Complexity

```
x = 5;

if (y>0) {

   x = 1;

} else {

   x = 2;

}

z = x + 2;
```



Cyclomatic Complexity C

- Number of independent paths
- Requires CFG with single exit node

$C := E - N + 2P$ (McCabe,1976)

E=#Edges, N=#Nodes,
P=#Connected Components

Here: $C = 6 - 6 + 2 * 1 = 2$

Details: see exercise session

Heuristics: $C > 10$ rethink design/coding to reduce complexity

Code Measure: Class and Interface Coupling

Coupling…

…indicates the amount of dependence between classes and packages

Definition (Class and Interface Coupling)

A class or interface C is coupled to a class or interface D if C requires D directly or indirectly.

- A class that depends on 2 other classes has a looser coupling than a class that depends on 8 other classes

Like cyclomatic complexity, coupling is an evaluative principle

Common Occurrences of Coupling in OOP

- Type X has an attribute that refers to a type Y

```
class X { private Y y = ... }
class X { private Y[] y = ... }
```

- Type X contains a (sub-)expression of a type Y

```
class X { private Object o = new Y(); }
class X { void m() { ... if (o instanceof Y) ... }
class X { void m() {... x = o.a.count + 3; ... } }
                              // where o.a is of type Y
```

- A type X object calls methods of a type Y

```
class X { void g() { Y.f(); }}
where class Y { static void f() { ... } }
```

Common Occurrences of Coupling in OOP

- Type X has a method that references an instance of type Y (as method parameter, local variable, return type, . . .)

```
class X { X(Y y) { ... } }
class X { Y f() { ... } }
class X { void g() { Y y = ... ; } }
class X { void h() { Object o = new Y(); } }
where class Y {}
```

- Type X is a subtype of type Y

```
class Y { ... }
class X extends Y { ... }
class X implements Y { ... }
```

Coupling in JAVA: An example

```
package de.tud.simpletexteditor;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class QuitAction
       implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
}
```

Class `QuitAction` is coupled directly with . . .

- ActionListener
- ActionEvent
- java.lang.Override
- java.lang.System
- java.lang.Object
  (any class in Java without an **`extends`** clause inherits directly from `java.lang.Object`)

Design Principle: Avoid Tight Coupling
Avoid Very Loose Coupling

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Classes with tight coupling are undesirable

- Changes in coupled classes may result in cascade of changes
- Tightly coupled classes hard to understand in isolation
- Tightly coupled classes hard to reuse elsewhere
  (because all coupled classes required as well)
- Tightly coupled classes result in low modularity

Generic classes with high chance of reuse must have very loose coupling

But: Very little or no coupling in general also undesirable!

- Goes against central OO metaphor:
  A system of connected objects that communicate via messages
- Loose coupling taken to excess results in active objects doing all work

Software
Engineering
Group

Tight Coupling vs Loose Coupling

Tight coupling to stable design elements and libraries is not a problem by itself*

(*: It still depends on the kind of elements to which it is coupled)

Warning:

The quest for loose coupling to achieve reusability for a (hypothetical!) project may lead to unnecessary complexity

Unwarranted decoupling can be one source of undue class proliferation: (“Ausuferung”)

Too many classes for the size of a given problem

Class Proliferation: Classes vs. Object Roles

Two ways to model behavior

As Class with own attributes and operations

As Role of an existing class, i.e. association

When to create dedicated classes?

Class Proliferation: Classes vs. Object Roles

Variant A	Variant B

class Person { ... }
class Father **extends** Person { ... }
class Mother **extends** Person { ... }
	class Person {
	 Person mother;
	 Person father;

	void init(...) {
	  mother = **new** Person(...);
	  father = **new** Person(...);
	  ...
	 }
	}

- It depends on the domain model!
  Do Mother/Father exhibit different behavior from Person?
- New classes should be warranted by truly different behavior

Ende der Vorlesung

Design Principles Part I

–

Fortsetzung nächste Woche