1 Design Principles

A good software should have certain standards it should fulfill. The quality of a software can be measured.

Quality Assurance

To create software we should first create a quality assurance plan, which is integrated into the software development process.

- Constantly asses design quality (quantitative and qualitative characteristics)
- Apply time-tested desing principles where applicable
- Use tools and design techniques that help to achieve quality
- Use design patterns (designs used across multiple projects, problems, etc.)
- Systematically verify correctness & performance
- Validate fulfillment of Requirements

1.1 Good Software

What is good software?

- Internal quality factors
  - Perceivable only by computer professionals
  - White box view
  - Code, databases, documentation, etc.
- External quality factors
  - Perceived by the customer / user
  - Depend on internal quality factors
  - Black box view
  - UI, speed, ease of use, etc.

Internal quality factors

- **Modularity:** How easy is it to modify the software?
- **Comprehensibility:** Is the software easy to understand?
- **Cohesion:** Is it clear what each component does?
- **Concision:** How concise is the code? (Code duplication, overly lengthy code)
- **Correctness:** Does the software work as intended?

External quality factors

- **Validity:** Does the software work as according to the requirements?
  - Needs precise requirements
  - Depends on correct design
  - Often conditional / codependant on correctness of internal quality factors
- **Robustness:** How well does the software handle abnormal conditions and errors?
- **Extensibility:** Can the software be extended to fulfill new requirements?
  - Architecture must be flexible / extensible
  - Often dependant on modularity of internal quality factors
- **Reusability:** How well can the software be reused in different contexts?
- **Compatibility:** How well does the software work with other software?
- **Portability:** How well does the software work on different platforms (hardware & software)?
- **Efficiency:** How fast and resource-efficient is the software?
  - Often depends on algorithms and data structures
  - Should be implemented for the common case
- **Usability:** How easy is it to use the software?
- **Functionality:** How far does the software usage extend?
  - Features should be consistent in usage and design

Overall the most important quality factors of good software are:

- **Maintainability:** Can be adjusted over time to new requirements
- **Efficiency:** Is reasonably fast and resource-efficient
- **Usability:** Is relatively easy to use and responsive
- **Dependability:** Does not cause physical or economical damage in case of system failure

1.2 Measuring Software Quality

In general, there are no universal way to measure quality as different software varies wildly. Oftentimes some metrics need to be negelected in favor of others, depending on the context (Usability over Security, Modularity over Concision, etc.).

What can be done is to define standards / heuristics to indicate quality of code. These are usually called **software metrics** or **code metrics**.

Software Metrics Pros

- Can be computed mechanically
- Can be used to indicate bad design

Software Metrics Cons

- Does not take semantics into account
- False sense of correctness

Code Metrics

- **Fan-in / Fan-out:**
  - Fan-in: Number of functions that call a specific function
  - Fan-out: Number of functions that are called by a specific function
- **Length of code:** Number of lines of code, indicates complexity
- **Cyclomatic complexity:** Number of decision points in code (control-flow graph)
- **Depth of conditional nesting:** Number of nested conditional statements, hard to understand, hard to test
- **Weighted methods per class:** How many functions are in a class, functions are weighted dependend on size / complexity
- **Depth of Inheritance:** Number of levels of inheritance, hard to understand

1.2.1 Control-Flow Graph (CFG)

A CFG represents all execution sequences of a program.

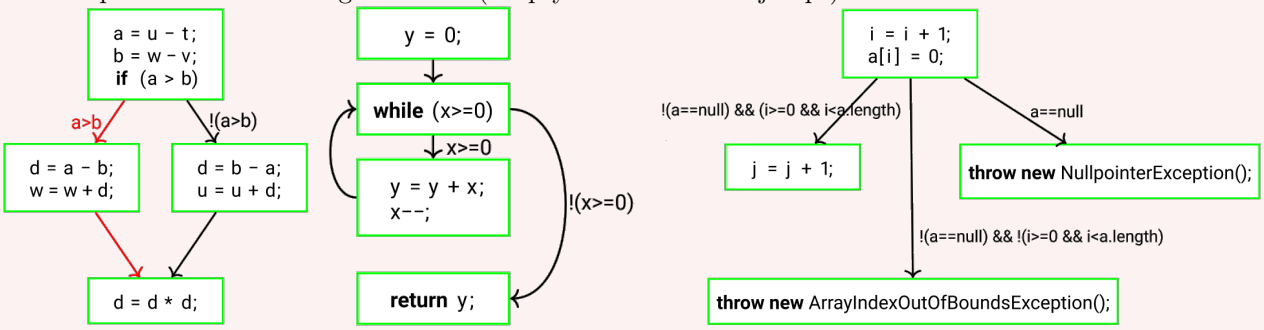
Basic Blocks in a CFG

A basic block is a maximal sequence of non-branching statements or instructions that are always executed together.
The execution of a basic block starts with the first statement, only the final statement can be a jump (branch or return).

```
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;
```

Control-Flow Graph

A Control-Flow Graph CFG(P) = (N, E, Label) of program P is a labeled directed graph, with nodes n ∈ N which represent the basic blocks of P and edges e ∈ E which represent the control flow of P. Hereby each edge e = (n_i, lb, n_j) ∈ E with n_i, n_j ∈ N and lb ∈ Label is a transition from n_i to n_j with label lb. The labels represent the branching condition (Empty for returns and jumps).



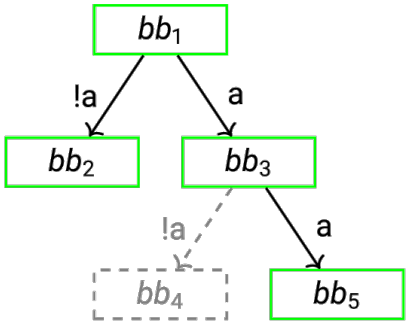
Red represents one possible execution sequence

In many cases the definition of explicit initial and exit nodes is important.

As there might be different end states sometimes the different exit nodes are important. For other reasons, like code metric computation, sometimes the exit nodes should be handled as a single node.

Sometimes code can result in unreachable nodes in one specific execution sequence. In this case the unreachable

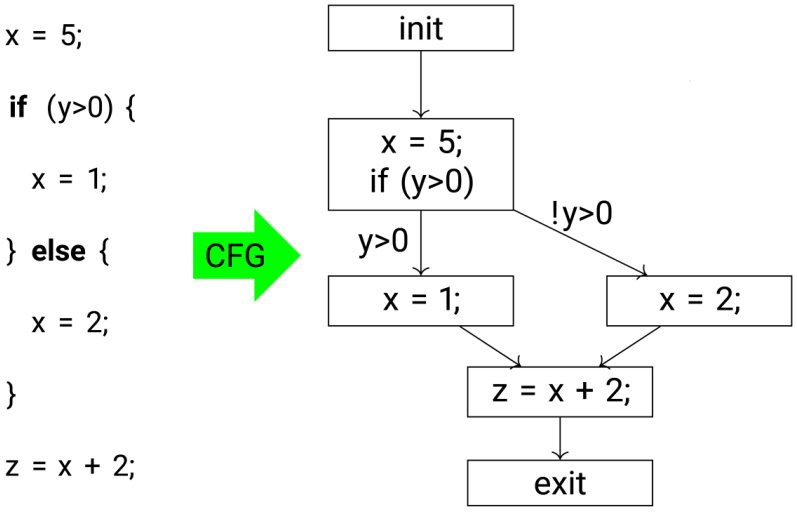
node / inactive edge should not be displayed.



Assumes bb_3 does not modify a

1.2.2 Code Metric: Cyclomatic Complexity

The cyclomatic complexity defines the number of independent paths through the code. It requires a CFG with a single exit node.



Hereby the number of independent paths through the code is 2. This can be calculated like follows:

$$C := E - N + 2P$$
with E edges, N nodes and P connected components

So in this example it would be:

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

In general, a cyclomatic complexit C of 10 or higher is considered to be complex - Rethinking design and coding might be beneficial.

1.2.3 Code Metric: Class and Interface Coupling

A class or interface C is coupled to a class or interface D if C requires D directly or indirectly. Hereby a class or interface that depends on 2 classes is considered looser than one that depends on 8 classes.

Common types of Coupling in OOP

- Attribute referal: X has an attribute of Y
  - `class X { Y y; }`
- Expression referal: X contains an expression of Y
  - `class X { Object o = new Y(); }`
  - `class X { void m() { …if (o instanceof Y) …} }`
- Method referal: X calls a method of Y
  - `class X { void m() { …y.m(); …} }` (Object method)
  - `class X { void m() { …Y.m(); …} }` (Static method)
- Method-Instance referal: X has a method that references an instance of Y
  - `class X { X(Y y) { …} }` (Parameter)
  - `class X { Y f() { …} }` (Return type)
  - `class X { void m() { Y y = …; } }` (Local variable)
  - `class X { void m() { Object o = new Y(); } }` (Local Expression)
- Inheritance: X inherits from Y
  - `class X extends Y { …}` (Extension)
  - `class X implements Y { …}` (Implementation)

Design Principles: Tight and Loose Coupling

Tight coupling is generally undesirable
- Changes in couples classes may cause undesired changes in other classes
- Tight coupling makes it hard to understand a class in isolation
- Tight coupling makes it hard to reuse a class
- Tight coupling results in low modularity

Generic classes with high reusability must have very loose coupling. However, very loose coupling or no coupling in general is also undesirable.
- Goes aginst OOP principles
- Loose coupling may require a huge number of active objects, decreasing performance

However, the tightness of couplings needs to be determined on a case-by-case basis.

1.2.4 Code Metric: Cohesion

Cohesion measures the strength of the relation among elements of a class. All operations and data in a class should ”naturally” belong to the concept modelled by the class.

Types of Cohesion

Ordered from undesirable to ideal:
- **Coincidental**: No meaningful relation among elements
- **Temporal**: Class elements are executed together
- **Sequential**: Result of one method in input of another
- **Communicational**: All functions access the same input or output
- **Functional**: All elements contribute to achieve a single, well-defined purpose: **Ideal**

Lack of Cohesion of Methods (LCOM)

Cohesion is often evaluated by the **Lack of Cohesion of Methods (LCOM)** metric. Hereby, a class C is defined as a set of instance fields F and methods M (excluding constructors). This set is then used to define an undirected Graph G(M,E) with vertices M and Edges E.

$$E = \{\langle m_1, m_2\rangle \in M \times M | \exists f \in F : m_1 \text{ and } m_2 \text{ access } f, m_1 \neq m_2\}$$

The LCOM(C) is then defined as the number of **connected components (CC)** of G(M,E). This means that for a class C with |M| = n, the LCOM(C) $\in [0, n]$. Therefore a high LCOM value indicates low cohesion. An issue with this metric is that its definition needs to be refined for special methods, like the constructors, hashCode, toString etc. methods. While these are technically part of the class, they are considered standard components of each class and therefore do not count towards the LCOM.

Low Cohesion is generally undesirable. As the classes can be hard to comprehend, reuse and maintain. Low cohesion also often indicates too-corse abstraction, meaning classes take responsibility for too many tasks, that should be handled by other classes.

As a rule of thumb: A class with high cohesion can often be described in a single sentence.