# 1 Verification

## 1.1 Introduction

### 1.1.1 Validation vs. Verification

**Validation**

**Validation** ensures that the software meets the costumers expectation.

Are we building the **right** system?

- Is the feature set as intended and complete?
- Does it fit into the organizations workflow?

Validation is reliant on correctly performed requirements analysis and thus cannot be done without involving the user / customer.

**Verification**

**Verification** ensures that the software is correct in respect to the system specification.

Is the system built **right**?

- Do the features work as specified?
- How does the system react to faults?

Verification is performed in solution space by the developer and thus doesn't necessarily involve the user.

### 1.1.2 Verification Techniques

**Static**

Static techniques do not require code execution.

- **Code Review:** Can be done at any stage of development
- **Static Checking:** Automated analysis of source code (Type checks, bug finders, etc.)
- **Formal Verification:** Ensures that a program satifies a formal specification

**Dynamic**

Dynamic techniques require code execution.

- **Testing:** Assert correct behavior for specific inputs
- **Runtime Monitoring:** Instrument program with safety assertions, whose violations are detected at runtime

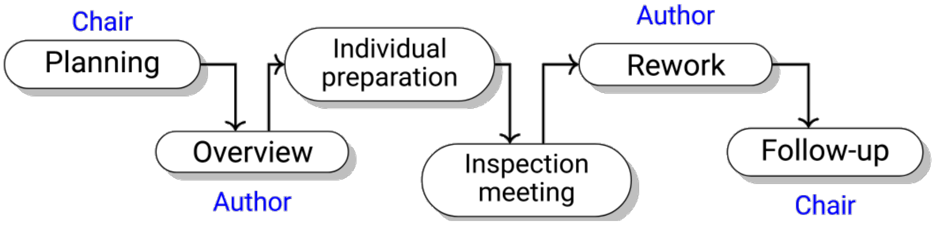
## 1.2 Code Review

A **Code Review** is an inspection process in which a team reviews project code, with the goal of identifying errors, bugs, deviations from conventions, etc.

### 1.2.1 Building Blocks

In a code review all team members take on specific roles:

- **Author / Owner:** Writing the code
- **Inspector / Reader:** Inspecting the code
- **Scribe:** Writing down comments, questions, requirements, etc.
- **Chair / Moderator:** Leads project and manages discussion and requirements



### 1.2.2 Check Lists

Code Reviews are often driven by a **Check List**, which is used to specify specific fault classes that should be checked. These are often individual to the project environment based on language, coding standards, etc.

**Check List Example**

| | |
|---|---|
| **Data Fault** | Are all variables initialized? |
| **I/O Fault** | Are all input variables used? |
| **...** | ... |

**Advantages**

- Empirical evidence that they work and save cost
- Distribute knowledge of the codebase to all team members
- Find defects before they might cause problems in tests
- Improves code quality
- Code does not need to be executed

**Disadvantages**

- Team members might feel criticized
- Time pressure might become worse because of the review
- Only works when properly conducted

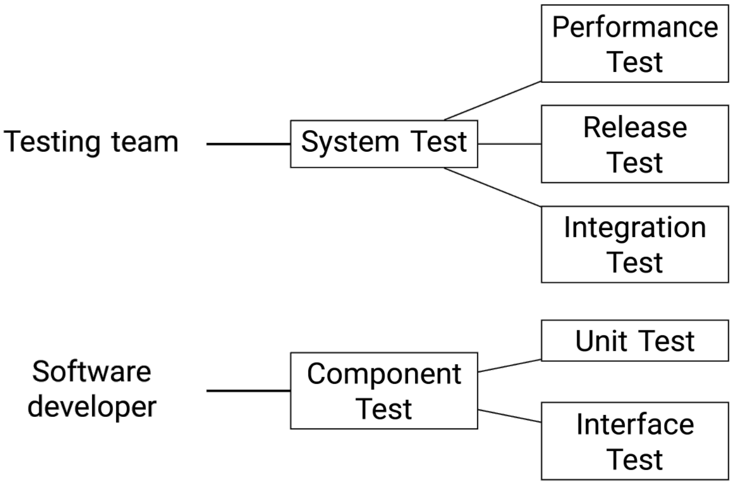
## 1.3 Software Testing

Testing can never show all faults, but it can reduce them. (Falsifiability)

### 1.3.1 Constituents

- **System or Implementation under test (SUT, IUT)**
- **Test Inputs**
- **Test Harness:** Runtime environment, preparation and execution, clean up
- **Test Verdict:** Given by **Test Oracle**

### 1.3.2 Test Levels



### 1.3.3 Test Plan

A test plan contains detailed descriptions of the testing process. It is a document intended to be read by humans.

- **Work Plan:** Phases, schedules, etc.
- **Testing Procedures**
- Explanation of the **test design**
- **Test documentation just as important as code documentation**

### 1.3.4 Test Design

1. Identify and analyze responsibilities of the IUT
   - **Pre- and Postconditions** in use cases
   - **Minimal and Success Guarantees** in use cases
   - Analyze distribution of responsibilities
2. Add test cases based on
   - Use Case-, Design- and Code Analysis
   - Suspicions, minimal success guarantees
   - General Heuristics (Domain / Expression boundaries)
3. Determine for each test case how verdict is reached: Provide expected results, programmed or human test oracle

### 1.3.5 Test Automation

Testing is very expensive; Typically about 15 - 20% of development costs are allocated towards testing.

These can be reduced automating tests:

1. **Running the tests:** Nightly, after each change, etc.
2. **Generating test cases:**
   (a) Code-driven
   (b) Model-driven
   (c) Data-driven
3. **Generating test verdict:**

(a) Test oracle as a small program, added to the test harness

(b) Oracle generated automatically from formal specification

**Tasks of a Test Automation System**

1. Set up test environment
   - Start servers, establish connection, register services
2. Start IUT
3. Bring IUT to required pretest state
   - Load required data, create required objects
4. Set tests inputs
5. Evaluate output and test verdict
6. Clean up environment
   - Delete files, stop services, reset data

Not everything is possible to automate, some manual tasks remain (Test inputs, test oracle)

### 1.3.6 Test Goal

Establish **sufficient** trust, that system is operational by exercising the interfaces between its parts.

### 1.3.7 Test Input

**Test (Data) Point**

A **Test Point** is a specific value for

- a test case input
- a state variable

The test point is selected from a domain. A **domain** is the set of values that input or state variables can assume.

**Heuristics for Test Point Selection**

- **Equivalence Classes:** Singular test point for expected equivalent outcome
- **Boundary Values:** Min/Max of ordered domain, pivot for comparison
- **Special values:** Null, other values with specific semantics

### 1.3.8 Other Definitions

**Test Case**

A **Test Case** consists of

- Pretest State of IUT
- Test Point / Conditions: test input
- Expected result

A collection of test cases is called a **Test Suite**.

**Test Run**

A **Test Run** is the execution of a test suite on a single IUT. A test whose results are equal to the expected results gets the **verdict Pass**, otherwise a **Fail**.

#### Test Driver & Test Harness

A **Test Driver** is a class or program that applies test cases to an IUT.
A **Test Harness** is a system of test drivers and other components that support test execution.

#### Fault & Failure

A **Fault** is missing or incorrect code.
A **Failure** is teh manifested inability of a system to perform a required function within specified limits (Time, memory, etc.)

### 1.3.9 JUnit Test Framework

#### Example of JUnit Test

```java
public class AccountTest {
    Account account;
    @BeforeEach // Runs before each test (Pretest State)
    public void setUp() {
        account = new Account(100);
    }
    @AfterEach // Runs after each test (Cleanup, Posttest State)
    public void tearDown() {
        account = null;
    }
    @Test
    public void successfulWithdrawTest() {
        assertTrue(account.withdraw(50)); // Delivers a passing verdict if value is true
    }
    @Test
    public void failedWithdrawTest() {
        assertFalse(account.withdraw(150)); // Delivers a passing verdict if value is false
    }
}
```

## 1.4 Test Coverage

### 1.4.1 When to Stop Testing?

**Structural Criteria (Code Structure)**

Based on the **Control Flow Graph (CFG)** of a program.

**Statement Coverage (SC)** Each **statement executed** at least once

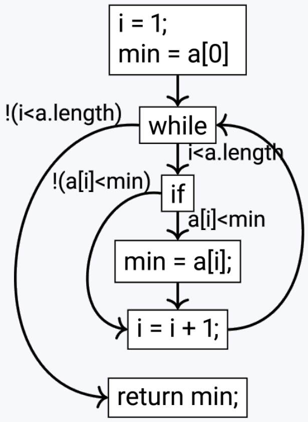
**Basic Block Coverage (BBC)** Each **basic block** executed at least once (implies **SC**)

**Branch Coverage (BC)** Each **outgoing edge** from a node in the CFG is executed at least once (implies **BBC**)

**Path Coverage (PC)** Each **path** through the CFG is executed at least once (implies **BC**), unachievable in practice, # of paths grows exponentially

**Example of Structural Coverage**

```
i = 1;
min = a[0];
While $i < n.length$ do
    If $a[i] < min$ then
        min = a[i];
    i++;
return min;
```



**Definitions**

**Execution Path** A complete path through the CFG

**Feasible** A path that can actually be taken

**Logic-Based Criteria (Logical Case Distinctions)**

**Condition Coverage (CC)** Each **condition** evaluated as true and as false in at least one test run

**Decision Coverage (DC)** Each **decision** (guard) evaluated as true and as false in at least one test run

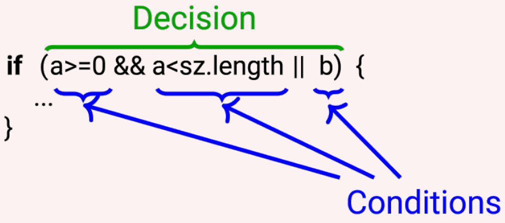
**Modified Condition Decision Coverage (MCDC)** Combines **CC** and **DC** & independence test

**Multiple-Condition Coverage (MCC)** All true-false combinations of conditions are tested in at least one test run

**Condition vs. Decision**

**Condition** A condition is a Boolean Expression & cannot be divided into sub-expressions

**Decision** A decision is a Boolean Expression constituting the guard of a conditional or loop statement



```
if  (a>=0 && a<sz.length ||  b) {
  ...
}
```

### Modified Condition Decision Coverage (MCDC)

For one occurrence of condition c **inside** decision d, MCDC is satisfied if:

1. Evaluates d at least twice
   - once where c is true
   - once where c is false
2. d evaluates differently in both cases
3. all other conditions in d evaluate identically in both cases **or** are not evaluated at all in at least one case.