# QA
# TDD

... do you test already
... or do you develop now?

… last comment from a developer…

… just before production.

I have looked everything very clear.
I'm sure: There aren't any fails.

After start, they was a fail at just his added position.

… a developer with compiler-DNA…

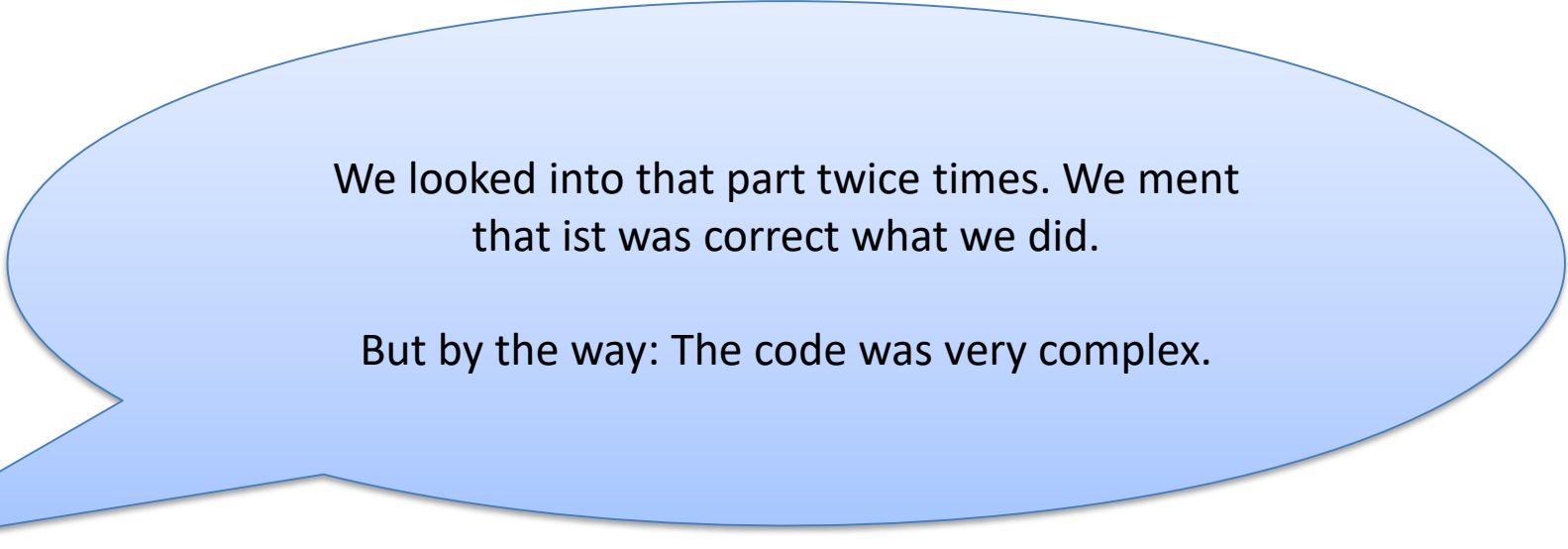I looked into the code. Line 5 will give a compiler error.

The param value isn't correct.

… answer from the project manager:
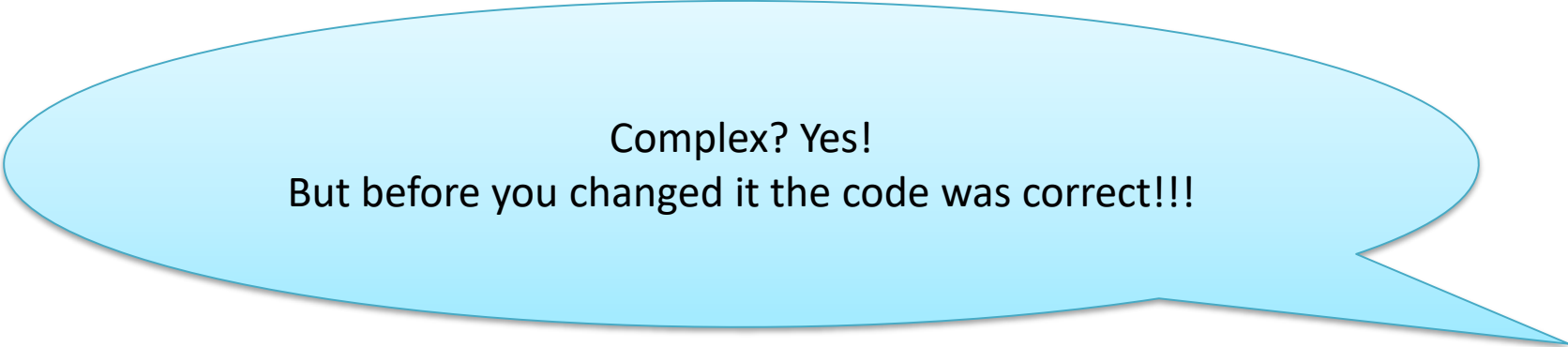
Did you know?:
JAVA-Compiler don't cost anything.

… developer after a little refactoring and a fail after life

We looked into that part twice times. We ment that ist was correct what we did.

But by the way: The code was very complex.

… answer from the projectmanager:

Complex? Yes!
But before you changed it the code was correct!!!

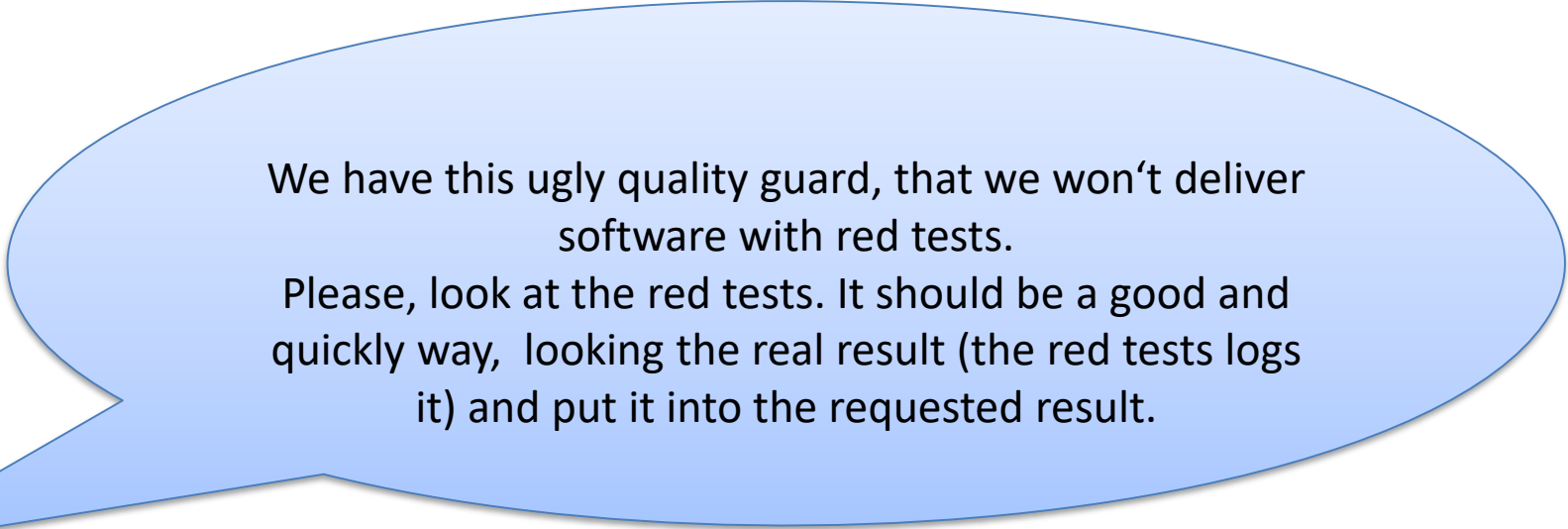… after a configuration change for a software release …

Oh oh …

Now, more than 20 tests will change to red, and I have to change the expected results.

… answer:

???
(without any word)

… just before delivery …

We have this ugly quality guard, that we won't deliver software with red tests.
Please, look at the red tests. It should be a good and quickly way, looking the real result (the red tests logs it) and put it into the requested result.
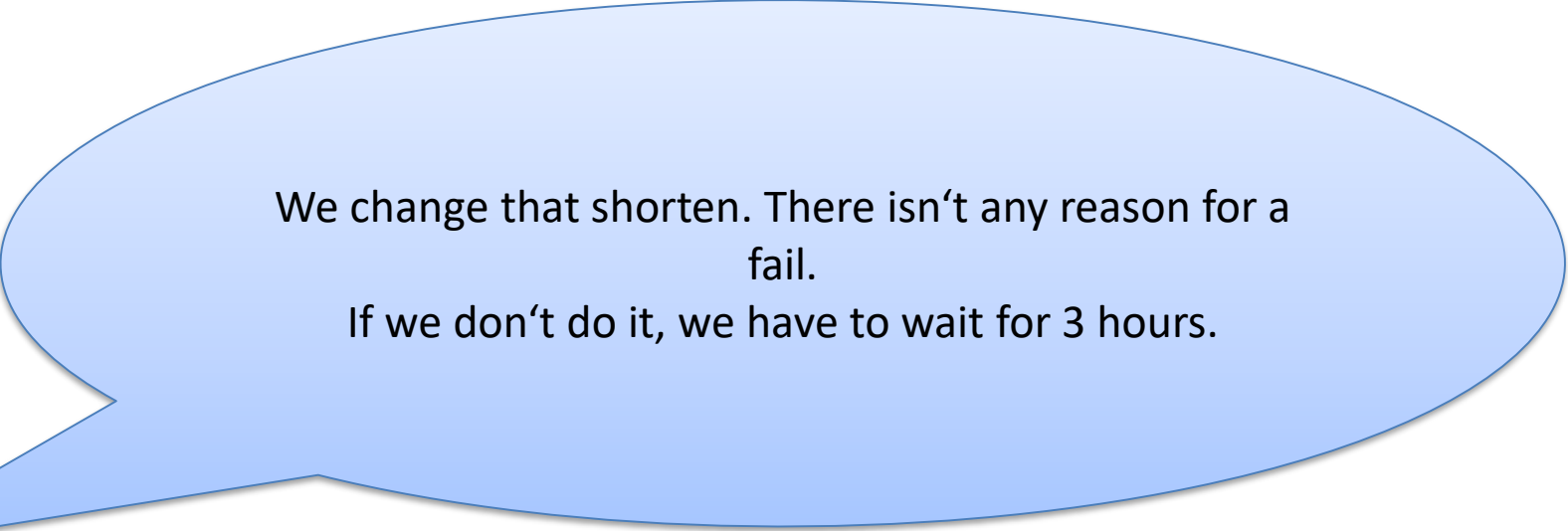
… weeks later after delivery:

Uups!

This feature doesn't work correct. Does ist ever work correct?

… just before change a XML-fragment …

We change that shorten. There isn't any reason for a fail.
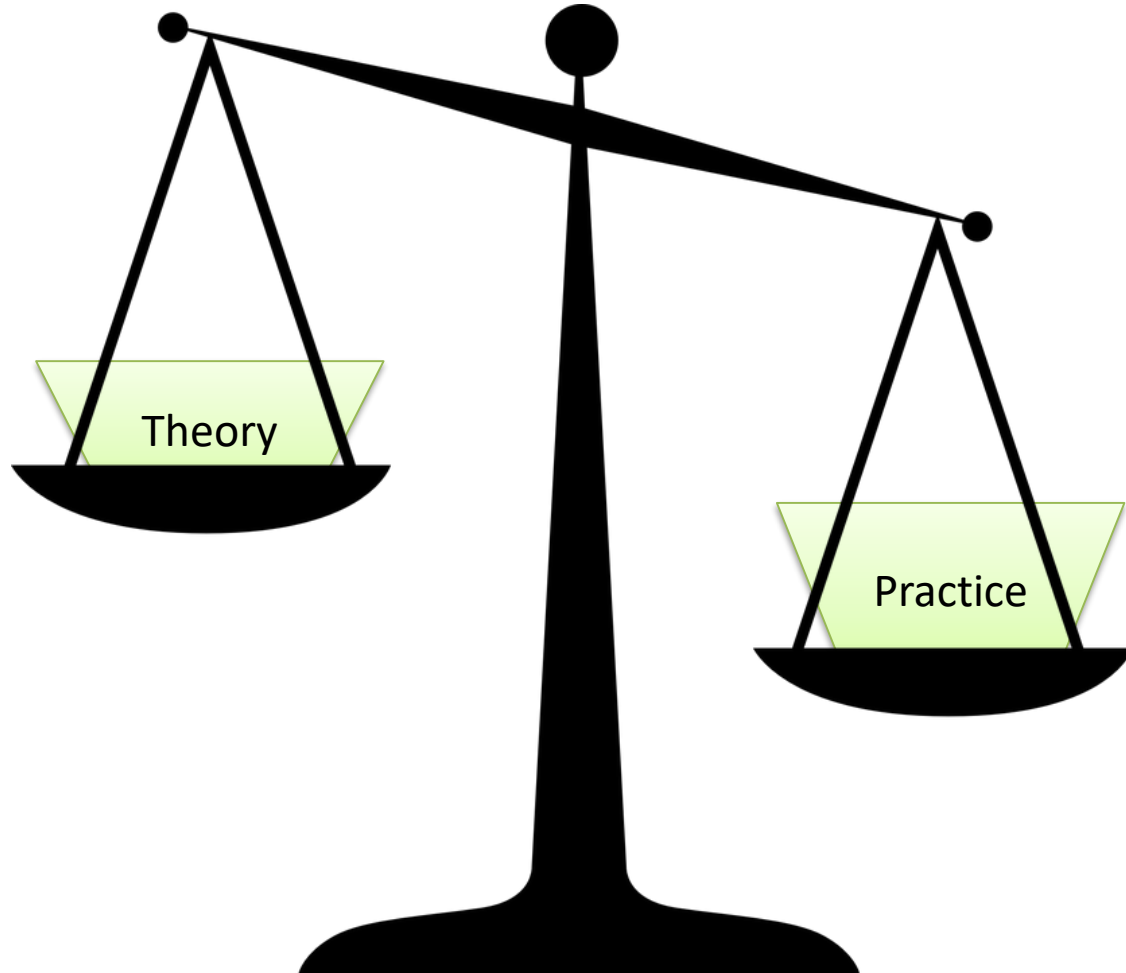If we don't do it, we have to wait for 3 hours.

… after delivery. The requested labels couldn't print, because the xml-fragment wasn't correct.

… for 3 hours, because there wasn't any possibility for re-delivery.
… costs: 1000EUR per minute

# Content

# Agenda

- Basics
  - Tests, why tests, kind of tests
  - How to test
- First action with TDD
  - What is TDD
  - Steps in TDD
  - Target with TDD

# BASICS AND DEEP DIVE INTO TESTS

# Term: Quality

Many people many definitions

# Term: Quality

Many people many definitions

Extendible

Correctness

Robustness

Easy learning

Efficient

Easy using

Portable

Security

Timeliness

Reusable

Effictive

Improved

# Term: Quality

Many people many definitions

Extendible

Correctness

Robustness

Easy learning

Efficient

Easy using

Portable

Security

Timeliness

Reusable

Effictive

Improved

# Quality: Reusable

# Quality: Effective and efficient

Not efficient            efficient

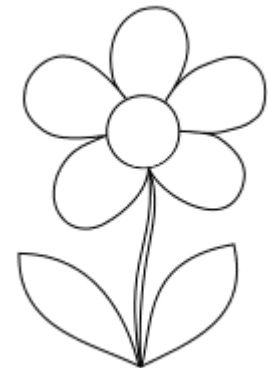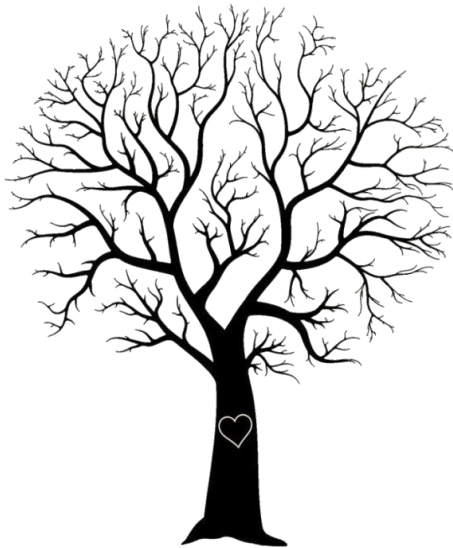Not effective
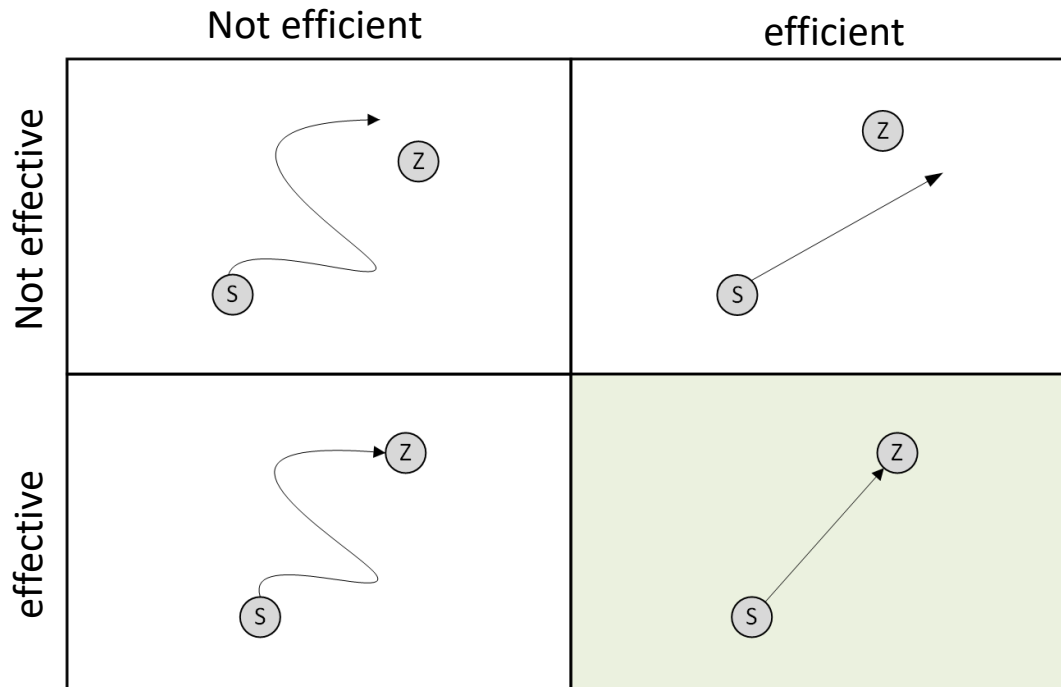
effective

From an email:

AW:AW:RE:REASON:
AW:AW:AW:RE:RE:RE:
Reply:RE:RE:RE:RE:RE:AW:AW:Answer:
RE:AW:RE:AE:AW:RE:Reasons,
why we don't work efficient

# Quality: effective and efficient

# Quality: a numeric definition

$$\text{Effectiveness} = \frac{\text{Result}}{\text{Goal}}$$

$$\text{Efficiency} = \frac{\text{Result}}{\text{Effort}}$$

# Quality: Robustness

# Quality: Reliability

0                                                                    t

- Performing quality aspects
  for a specified period of time

# Quality: Reliability

0                                                                    t

- Performing quality aspects
  for a specified period of time

Intended function
specific period of time
specified condtions
Probability
Satisfactory

# Quality: in one sentence

- Quality is the absence of
    - deficits
    - Bugs

# Quality: IEEE

- Mistakes

- Faults

- Failures

# Faults in Software: Reasons (1)

- In large system, no one understand the complete algorithm

- complexity of the code

- Complexity of the business

- Unpredictable behavior in some situations

# Faults in Software: Reasons (2)

- Unclear requirements
- Design errors
- Architecture errors
- No understanding design
- No understanding architecture
- Implementation errors
- Wrong assumptions

# Fails in Software

- Failure

- Error

- Fault

# Fails in Software

**Fault** — Mistake in programm/function

Function double(3) results in 9 (because the developer paste a „*" instead of „+")

In the start-phase of the flight the software interpreted a faulty signal and pushed the plane nose down

**The programm should works fine until some calls/times …**

Call double(2) results in correct number.

If the sensor works fine, the software works fine.

# Fails in Software

| Fault | Mistake in programm/function |
|---|---|

Function double(3) results in 9 (because the developer paste a „*" instead of „+")

In the start-phase of the flight the software interpreted a faulty signal and pushed the plane nose down

**The programm should works fine until some calls/times …**

| Error | Activated Fault |
|---|---|

| Failure | Deviation in system behaviour |
|---|---|

| Bug | Deviation in system behaviour |
|---|---|

# Fault, Error, Failure

```java
public boolean withdraw(BigDecimal money) {

    BigDecimal balance = amount.subtract(money);
    if (balance.compareTo(BigDecimal.ZERO) >0) {
        amount = balance;
        return true;
    }else {
        return false;
    }
}
```

# Fault, Error, Failure

```java
public boolean withdraw(BigDecimal money) {

    BigDecimal balance = amount.subtract(money);
    if (balance.compareTo(BigDecimal.ZERO) >0) {
        amount = balance;
        return true;
    }else {
        return false;
    }
}
```

FAULT: withdraw isn't possible if balance is zero.

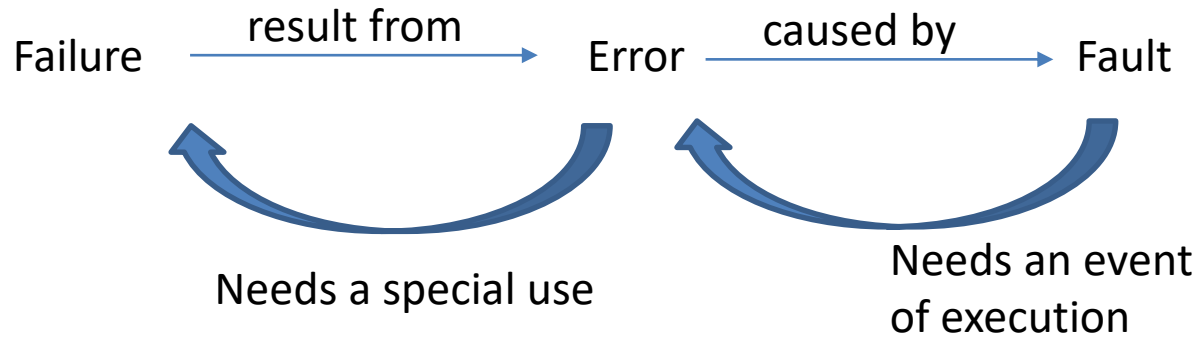ERROR: Call the function with amount-value, other calls proceed well

FAILURE: customer tries to transfer his complete credit to other bank account

# Fault, Error, Failure

```java
public boolean withdraw(BigDecimal money) {

    BigDecimal balance = amount.subtract(money);
    if (balance.compareTo(BigDecimal.ZERO) >= 0) {
        amount = balance;
        return false;
    }else {
        return false;
    }
}
```
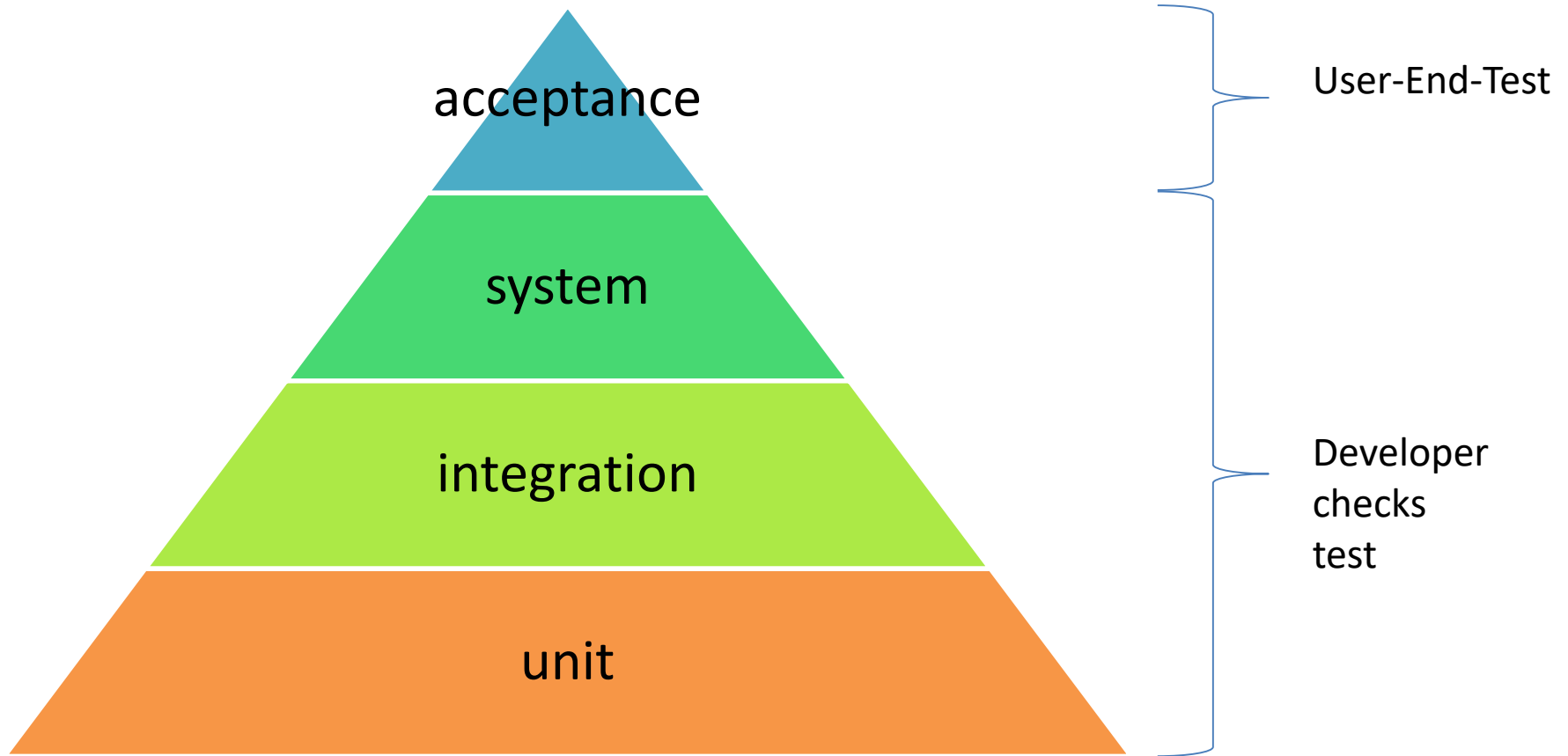
# Fault, Error, Failure

```java
public boolean withdraw(BigDecimal money) {

    BigDecimal balance = amount.subtract(money);
    if (balance.compareTo(BigDecimal.ZERO) >= 0) {
        amount = balance;
        return true;
    }else if (balance.compareTo(BigDecimal.ZERO) < 0){
        return false;
    }else {
        amount = balance;
        return true;
    }
}
```
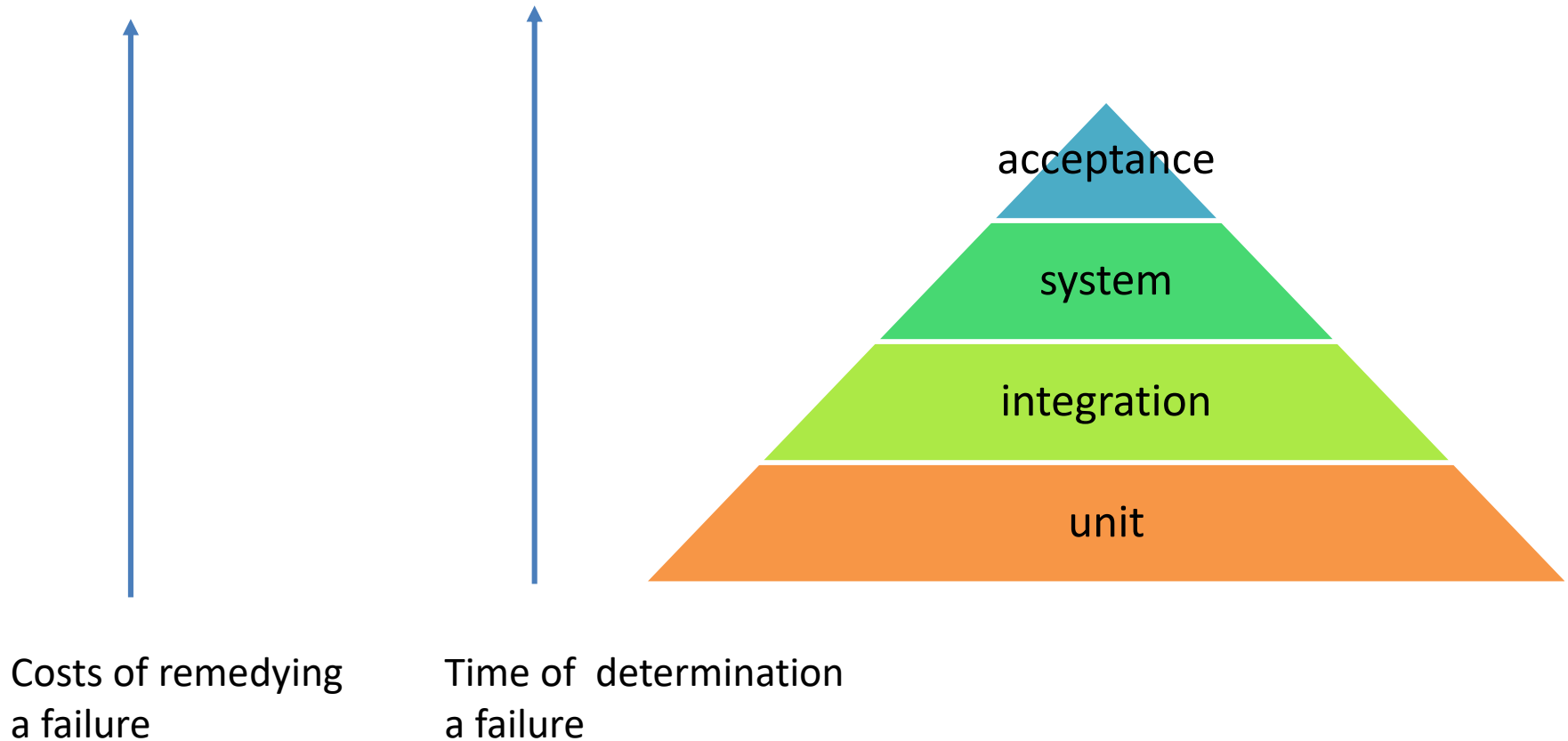
Failure → result from → Error → caused by → Fault

Needs a special use

Needs an event of execution

| Defect | Found Error in Developing/Test-Phase (by developer, tester, …) |
| --- | --- |
| Bug | Found Error after delivery (by customer, user) |

# Test-Level

# Test-Level



acceptance
system
integration
unit

Costs of remedying
a failure

Time of determination
a failure

# QA: Tests

With tests you trie to make a fail.
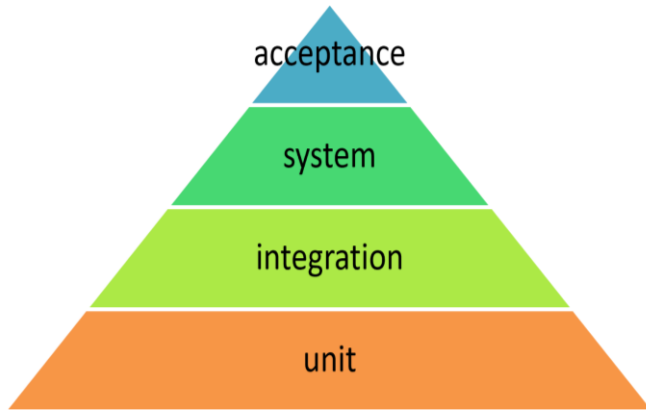
Or in other words:
Testing can only show the precence of errors, never their absence.
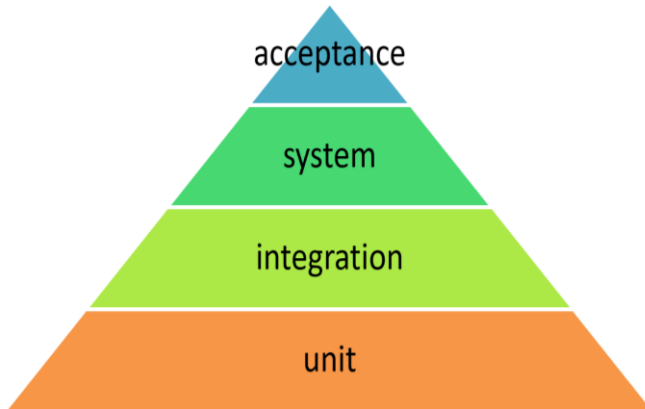{Dijkstra, 1930-2002}

# Aspects for tests

- A test is made for finding bugs
- Only a test, that previously failed is a good hint about the reliability for the test or the tested unit
- Testing stops a identification of bugs
  - Correction is „debugging"

# Cutback



- Quality
  - Absence of deficits
  - Absence of bugs

- Typical software Quality terms
  - Robustness
  - Efficience
  - Effectife
  - Reliability

# Cutback



- Errors in software
  - Requirements
    - Unclear
    - Complex
    - incomplete
  - Design
    - Error
    - Not understand
  - Architecture
    - Error
    - No understand
  - Code
    - Complex
    - Errors
    - Unpredictable behaviour sometime

- QA (quality assurance)
  - Defining quality guards
    - Ensure, that the product meets these guards
    - Improving over the lifecycle
  - Making tests
    - Try to make fails
    - Ensuring the correctness of the software
  - Testing can only show the precence of errors, never their absence.

# Go forward in making test
# for demonstrating a correct requirement

Simple step.
Make a  fail test

Note:
The test must be fixed with short code changes

```java
//a) An empty String ("") results 0
@Test
public void emptyStringResultZero() {
    StringCalculator stringCalculator = new StringCalculator();
    int result = stringCalculator.calculate("");
    assertThat(result, is(0));
}
```

This test failed:
- It gets an compile error

But, during writing a test you think about:
- Name of the class
- Name of the method
- Input of the method
- Output of the method

# Go forward in making test
# for demonstrating a correct requirement

➡️ Simple step.
Make a fail test

Note:
The test must be fixed with short code changes

```java
//a) An empty String ("") results 0
@Test
public void emptyStringResultZero() {
    StringCalculator stringCalculator = new StringCalculator();
    int result = stringCalculator.calculate("");
    assertThat(result, is(0));
}
```

➡️ Solve the test in the easiest way you can.

```java
public class StringCalculator {

    public int calculate(String string) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

# Go forward in making test
# for demonstrating a correct requirement

NOTE:

The only one task here was to make
a function which returns 0 by an empty
string.

That the function do return 0 by all
other input too doesn't matter.
Thats not part of the current task.

Stay by your task.

There are many other (additional) tasks
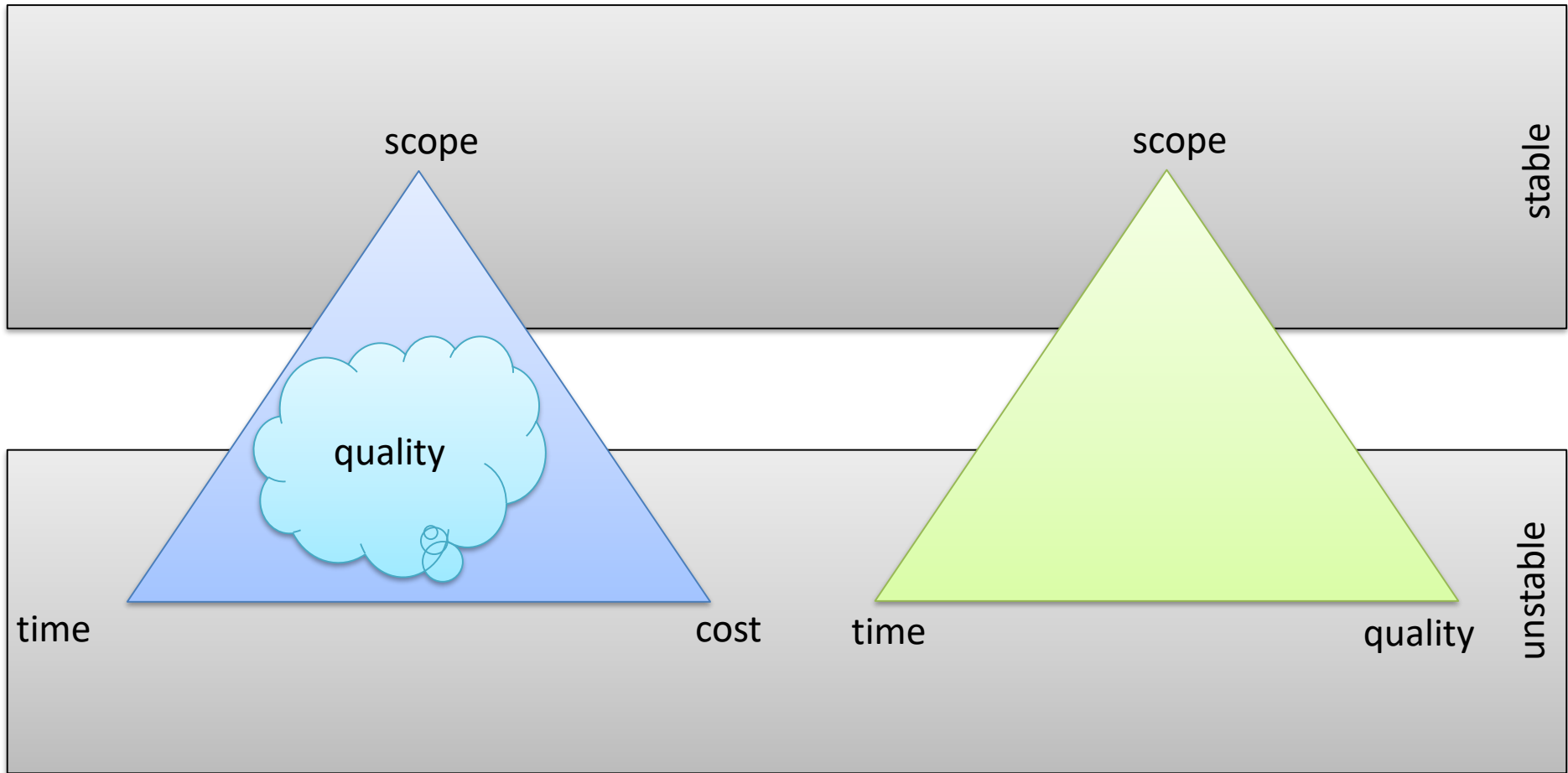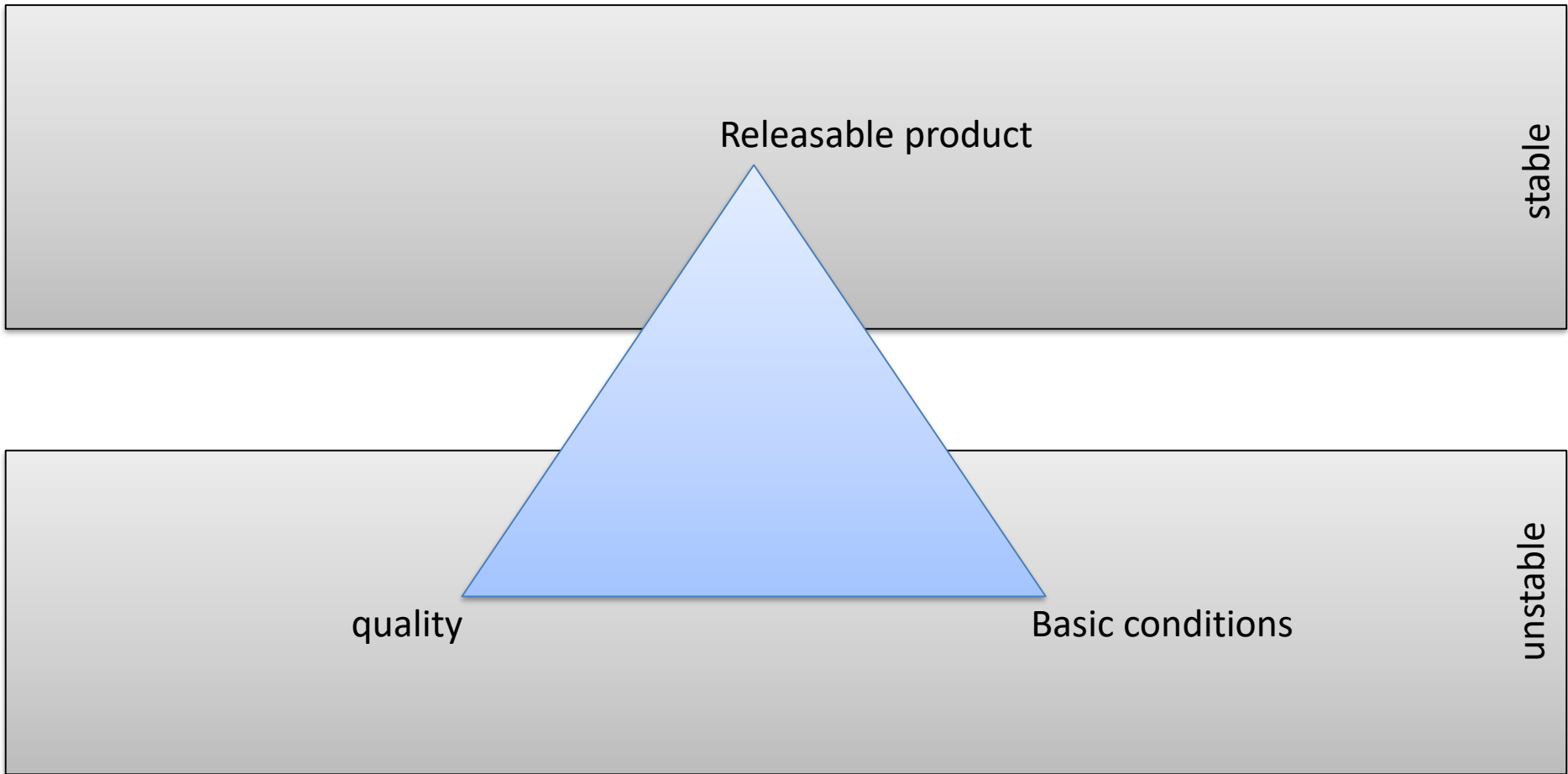defining the complete behaviour of the
calculator.

```java
public class StringCalculator {

    public int calculate(String string) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

# Quality meets Project live
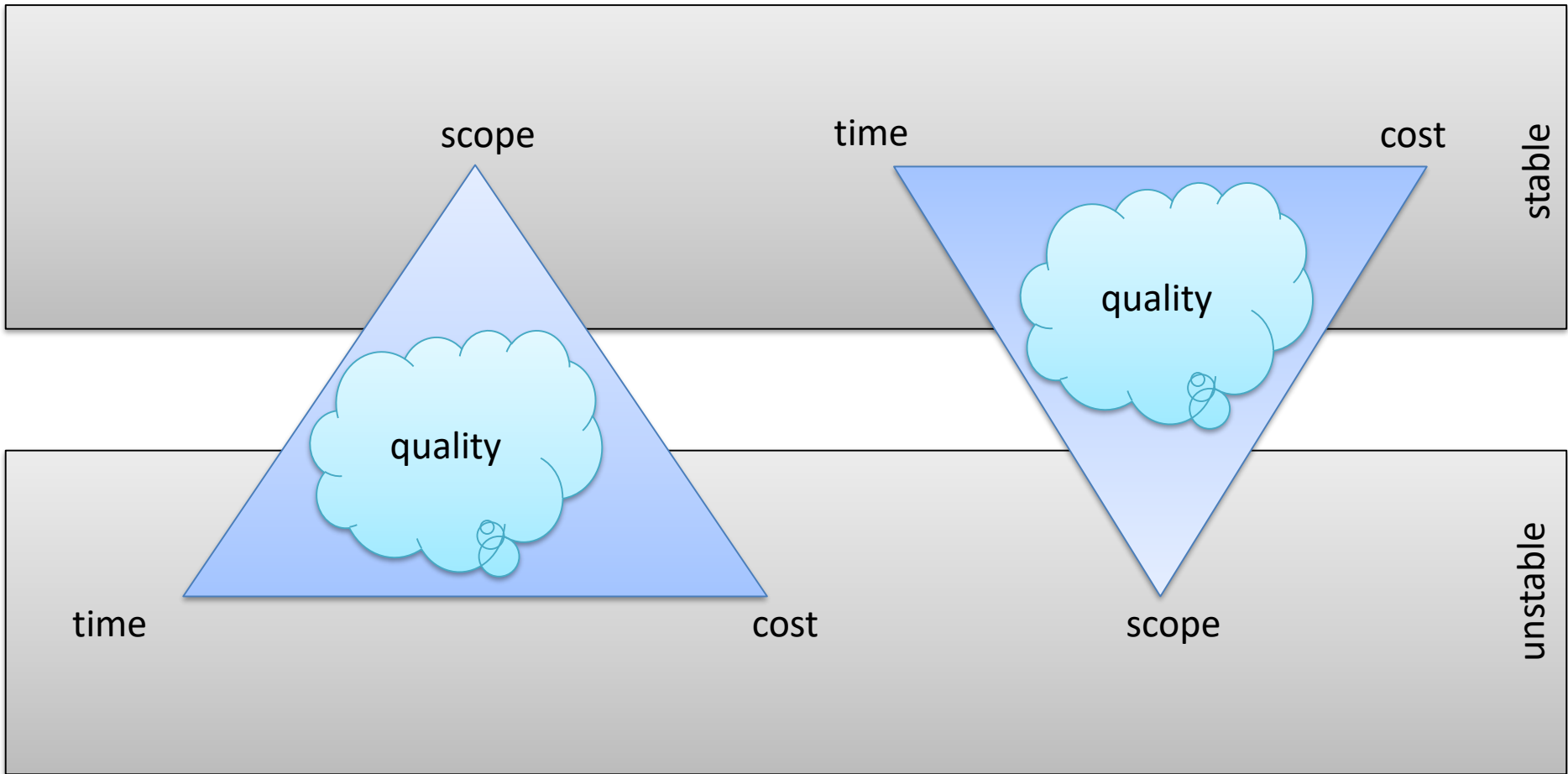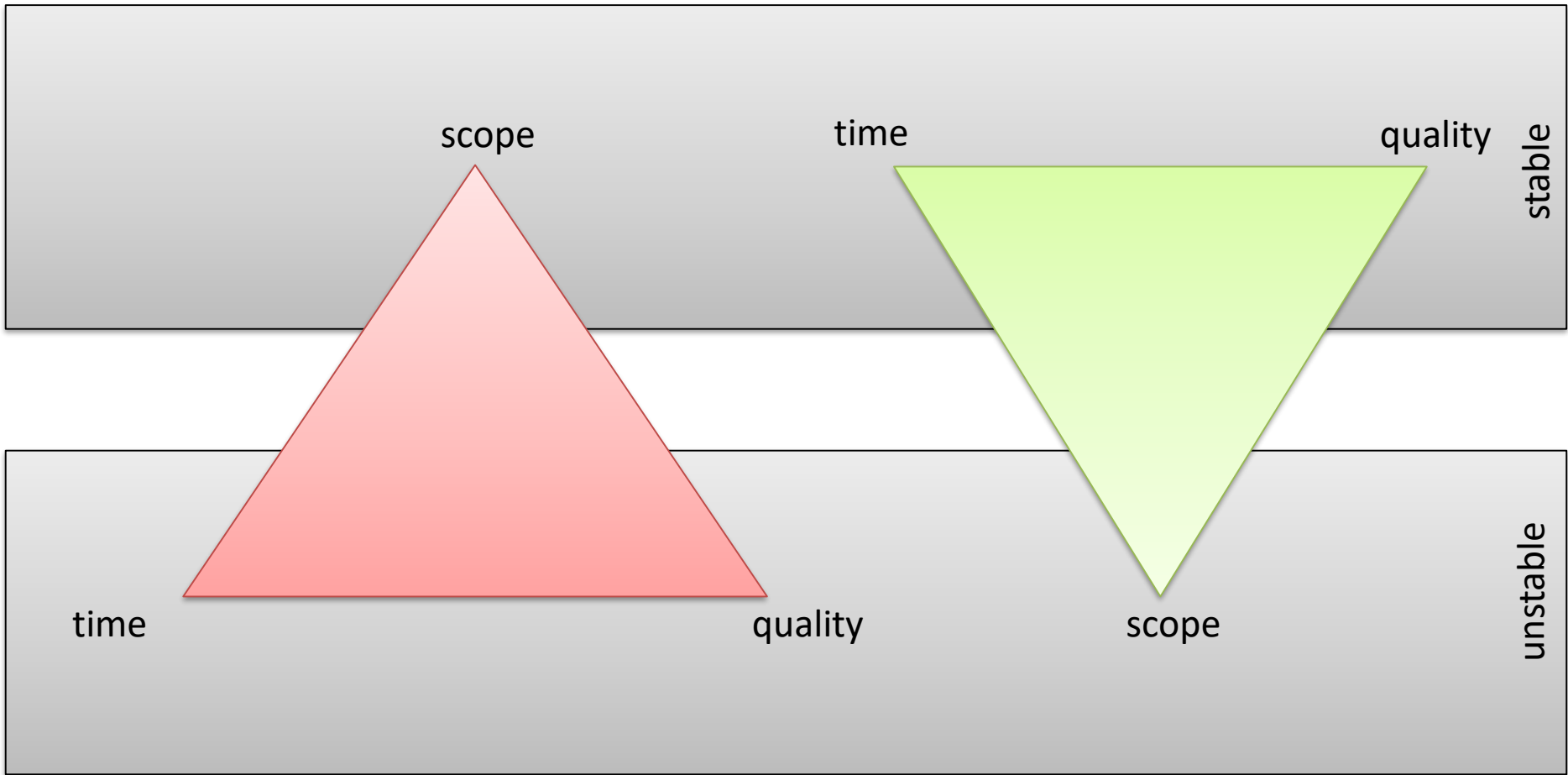
# Quality meets Project live

# Quality meets Project live

Releasable product

stable

quality

Basic conditions
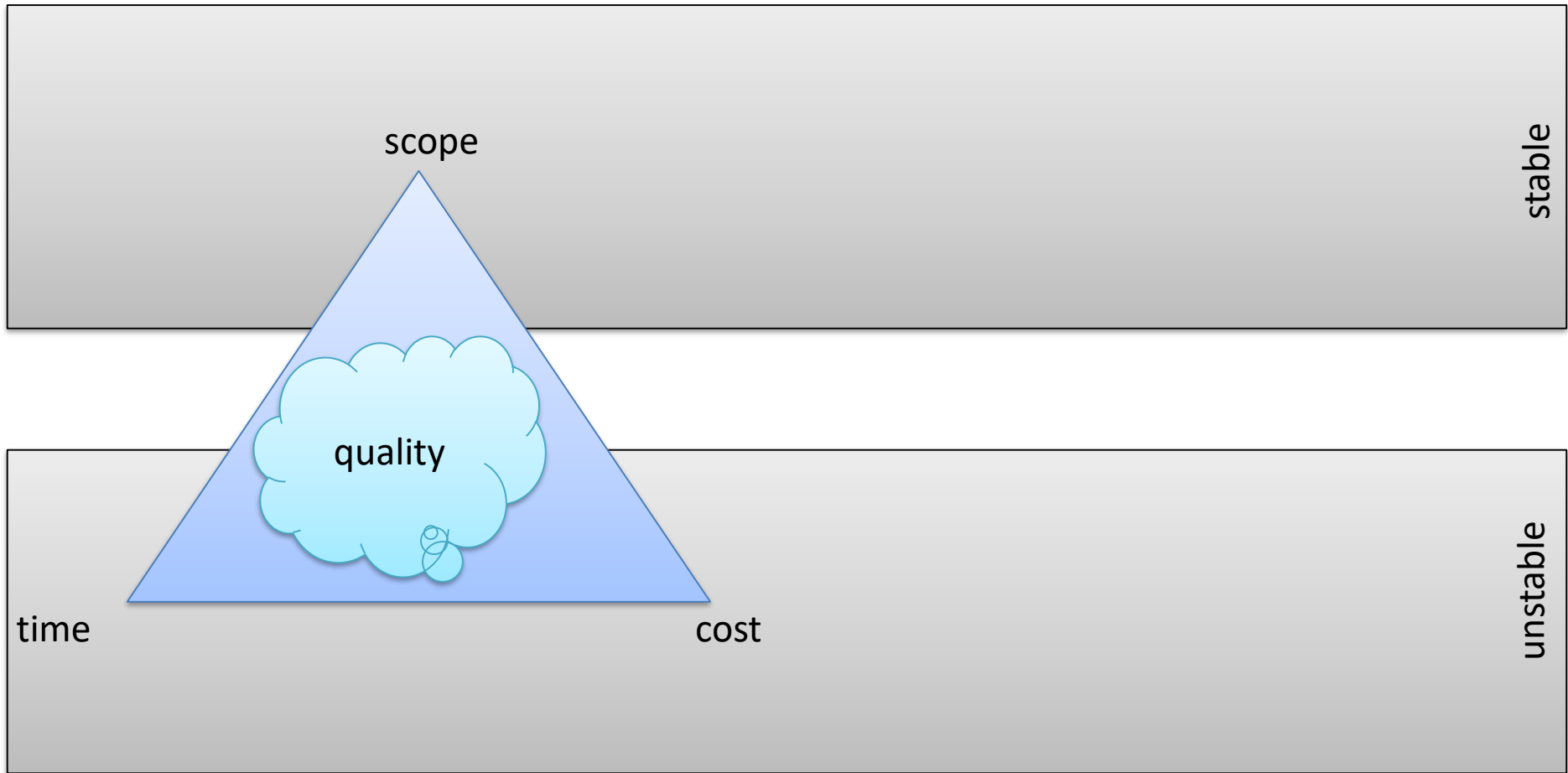
unstable

{Agile triangle, Jim Highsmith}

# Quality meets Project live

# Quality meets Project live



Typical agile view

# Verification and Validation

**Verification**

- Determine if the artifact (result of a development phase) establishes the requirements before creation
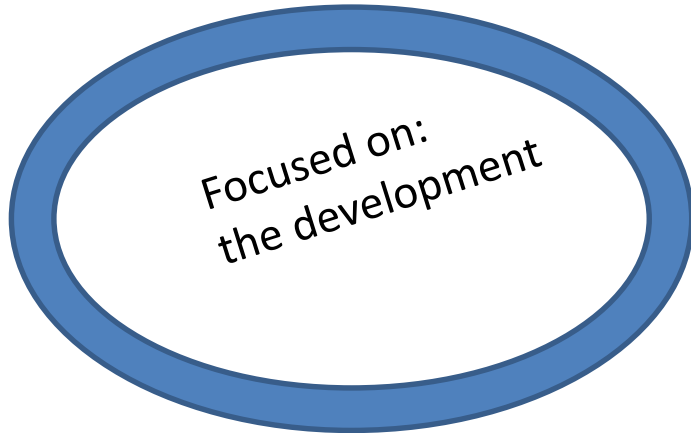
**Validation**

- Confirm that a product meets customers expectation(s)
- Confirm that a product meets its attended use

# Verification and Validation

**Verification**

- Determine if the artifact (result of a development phase) establishes the requirements before creation

Focused on: the development

**Validation**

- Confirm that a product meets customers expectation(s)
- Confirm that a product meets its attended use

Focused on: the (final) product
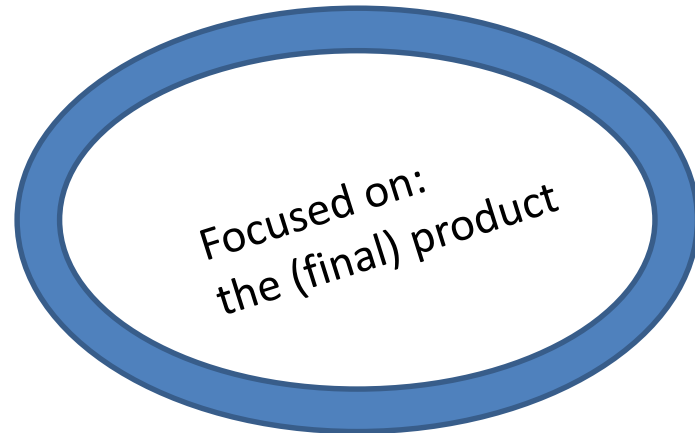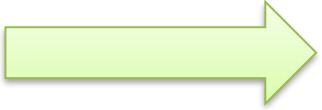
# QA: Quality Assurance

- Define quality objectives

- Define processes for meeting the defines objectives

- Improve the quality objectives over the time

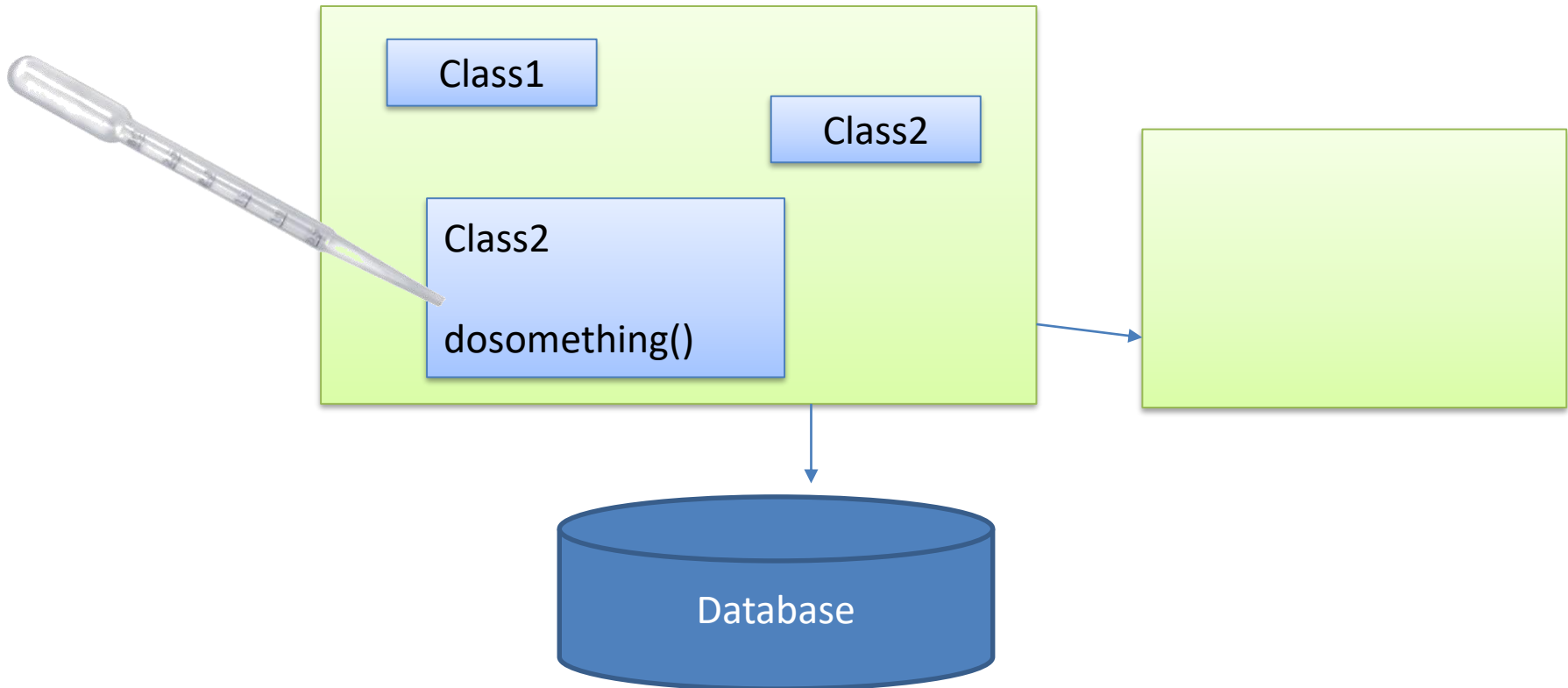# Term: Quality assurance (QA)

Activities
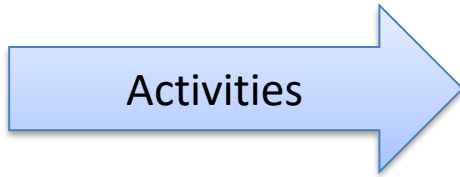
- Defining quality guards
  - Ensure, that the product meets the guards
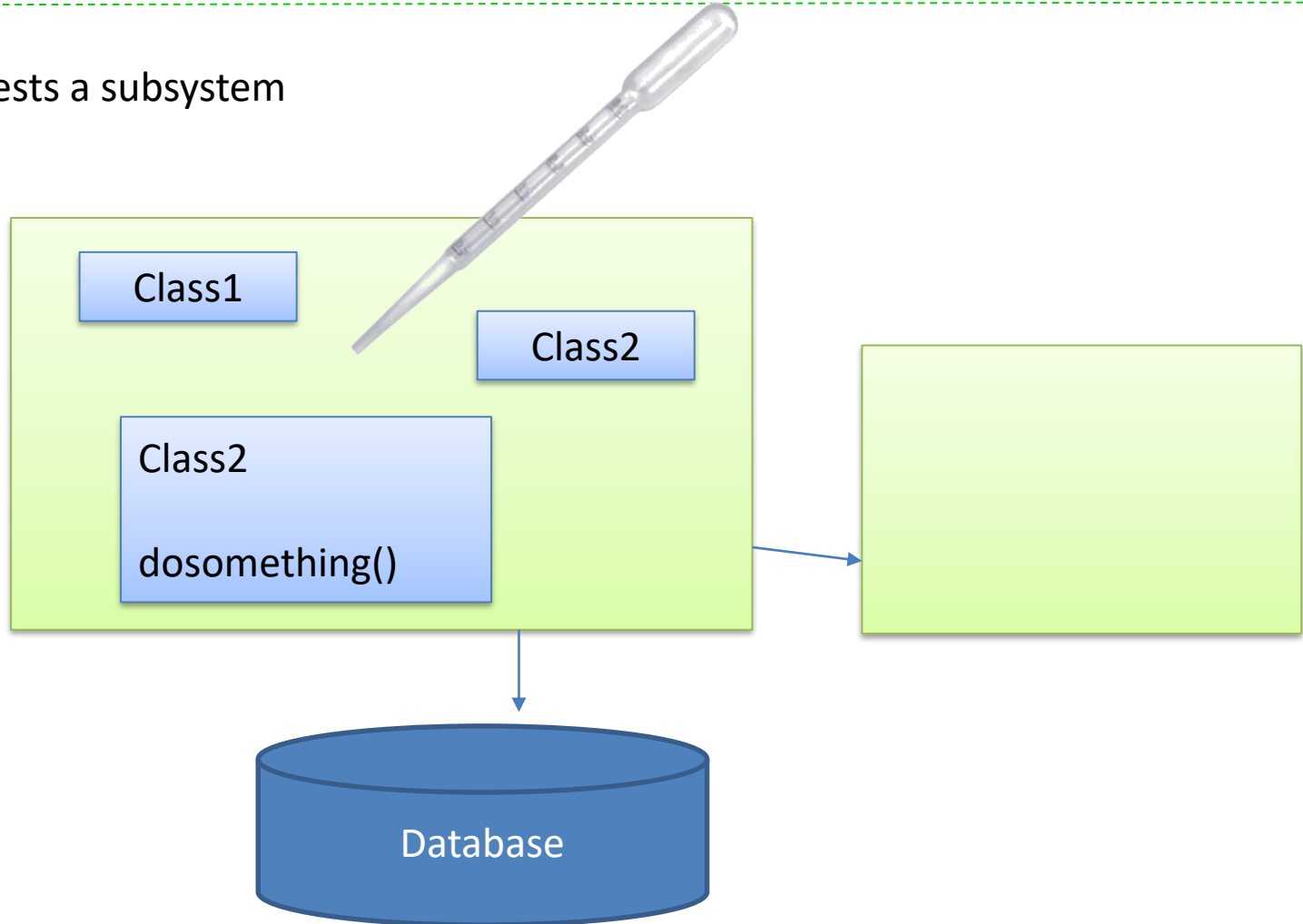  - Improve them over the lifecycle

# Testscope: Unit-Test

Tests a module/class or methode/function

Class1

Class2

Class2
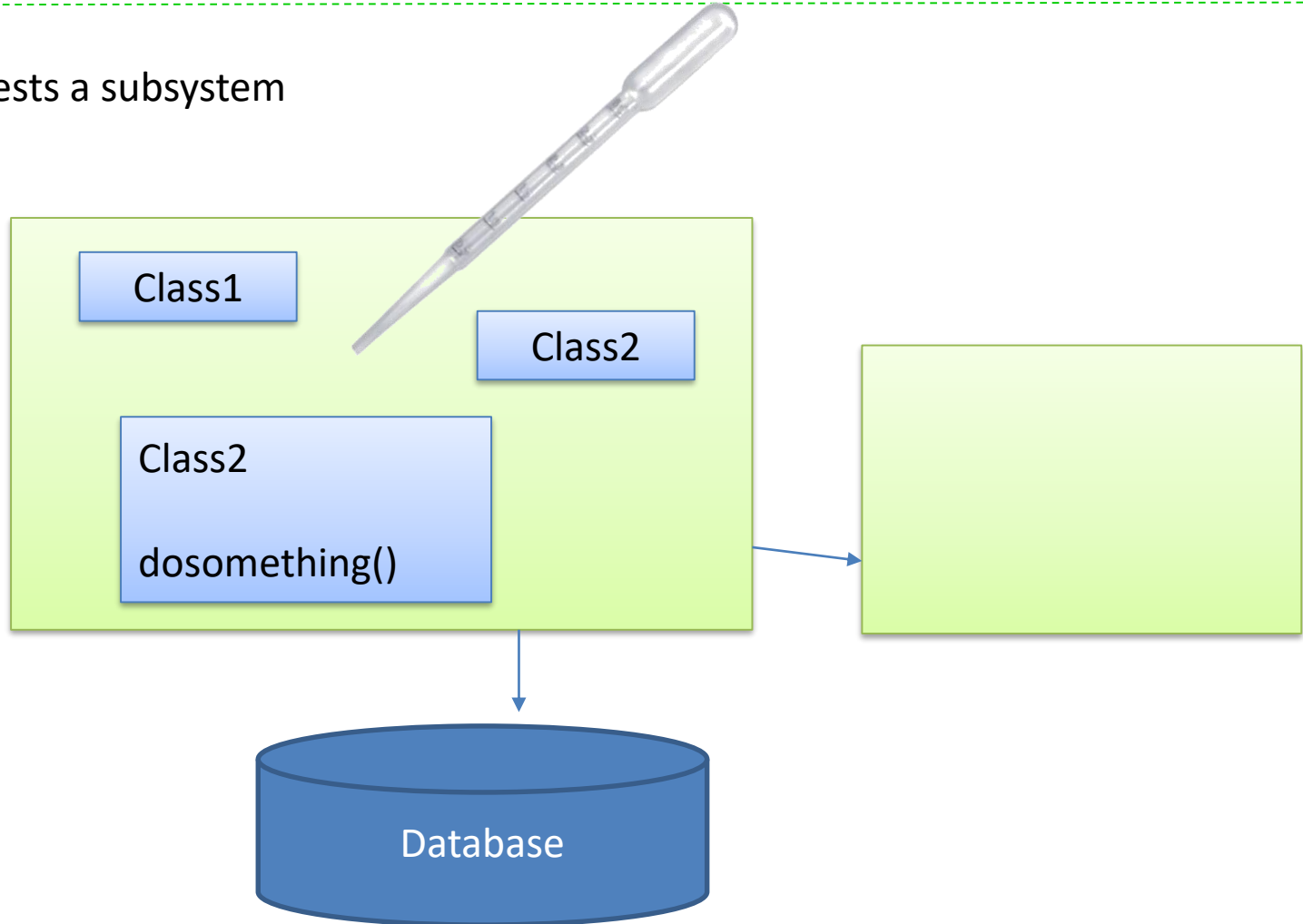
dosomething()

Database

# Testscope: Integrationtest

Tests a subsystem

Class1

Class2

Class2

dosomething()

Database

# Testscope: Integrationtest

Tests a subsystem

Class1

Class2

Class2

dosomething()

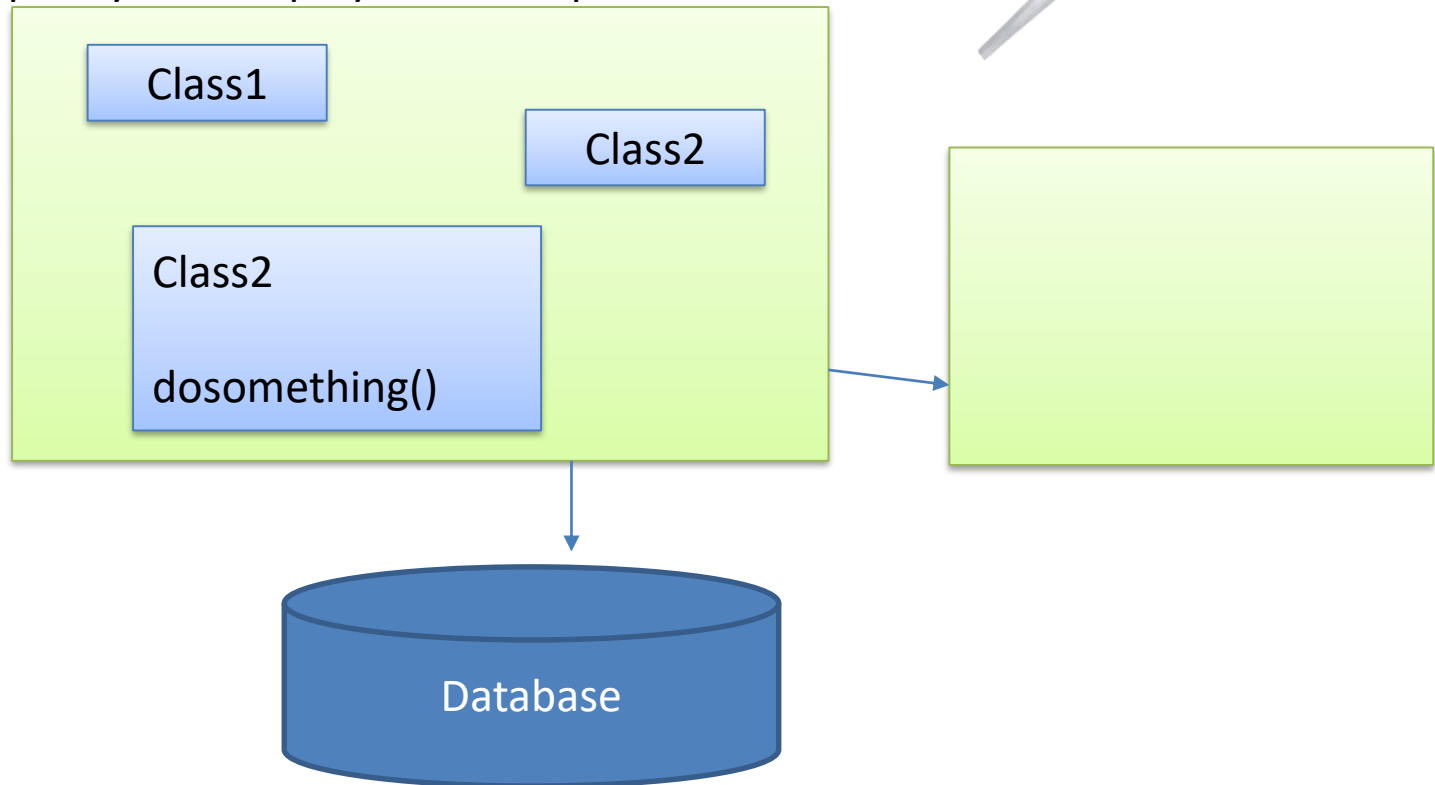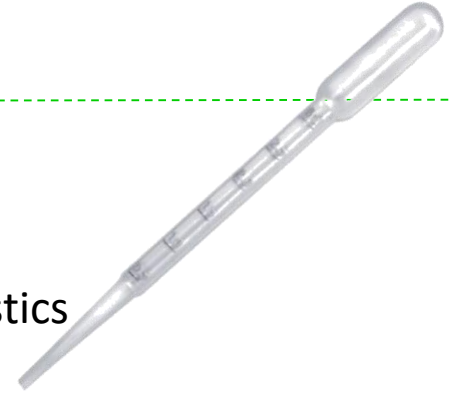Database

# Testscope: Systemtest

Tests an integrated application,
typically parameterized
typically with deployment and platform characteristics

Class1

Class2

Class2

dosomething()

Database

# Impact for test-level

- Start very early with testing
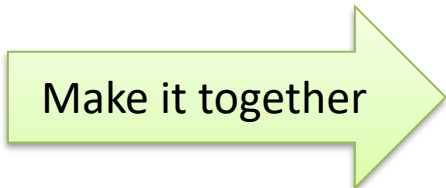- Integrate testing in your development process

# Types of tests

- Start very early with testing

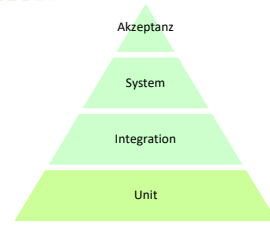- Integrate testing in your development process

Make it allone → Unit-Tests
Integration-Tests

Make it together → Pair-programming
Code-review
Crowd developing
Crowd architecture

# Where can tests help you

- Have you made a testable application
    - Indication for good architecture
    - Indication for good design
    - Indication for no fails
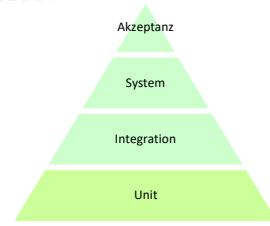
    - Indication for no technical depts

Please note:

An application lives.
You have to tell the same questions time for time.

Be never full satisfied.
Know that current satisfaction is only for the moment.

Do not go ahead too much.
There is no sence for develop requirement you dont know.
Reality will win and overtake.

# Where can tests help you

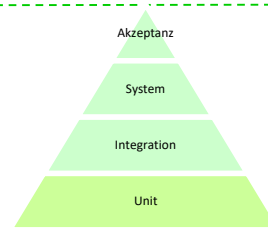Do you know the Beatles?

Do you know John Lennon?

Quote: ➡ Live is what happens while you plan your live.
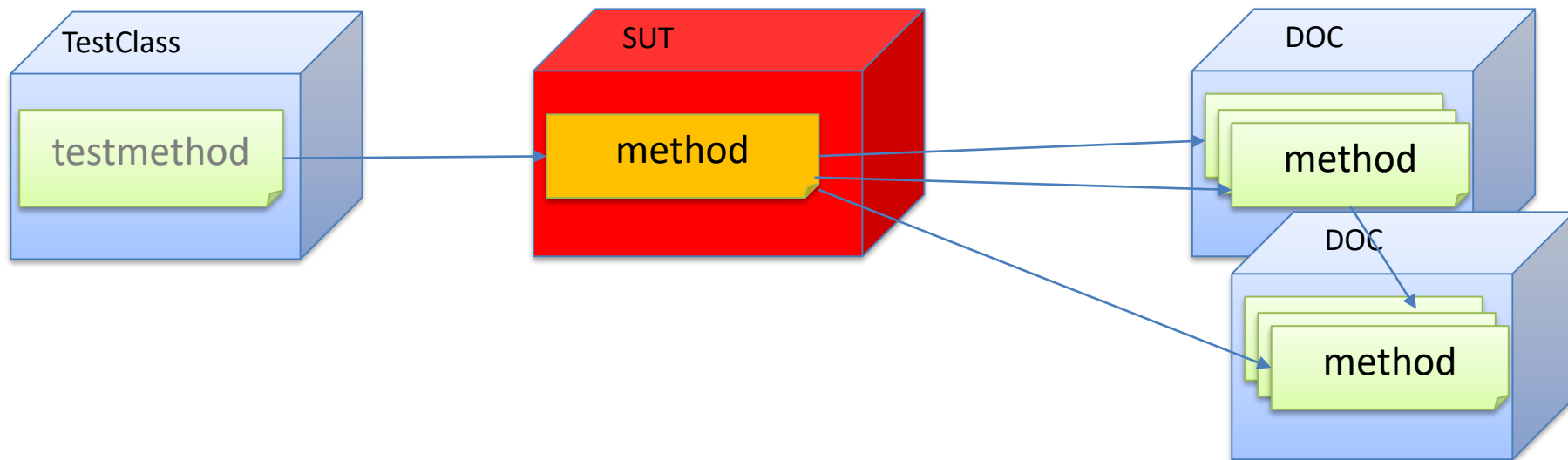
Don't care about the problems will come tomorrow ;)

But be ready to manage the future problems. So you can solve them in a very light way.

This side is free for technical reason

# Test-Level, JUnit

- Unit-tests check a special method/function of a component

- Unit-tests check …
  - … the implementation of a method/function
  - … the logic of a method/function

# Examples wrong created tests

- Test fails and proceeds accidental
- After changing configuration the test fails
- Test fails according events, e.g. time
- One or more tests fail if they proceed in other order

# Hotspots in UnitTests

- Tests check more as one thing
  - E.g. SUT does more than one thing
- Tests pass on concrete order only
- Tests take too much time
- Tests need database
- Tests need complete environment

# How to find good inputs

```java
public class FinancialAccount {

    private Long amount;

    public boolean widthdraw(Long money) {
        long balance = amount - money;
        if (balance >= 0) {
            amount = balance;
            return true;
        }else {
            return false;
        }
    }
}
```
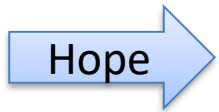
Problem ➡

Testing with all inputs isn't possible.
BUT: We need realistic inputs

# Finding input:
# partition testing, equivalence class

Devide complete possible input into parts (partitions) with equal types.
Take some inputs of every partition
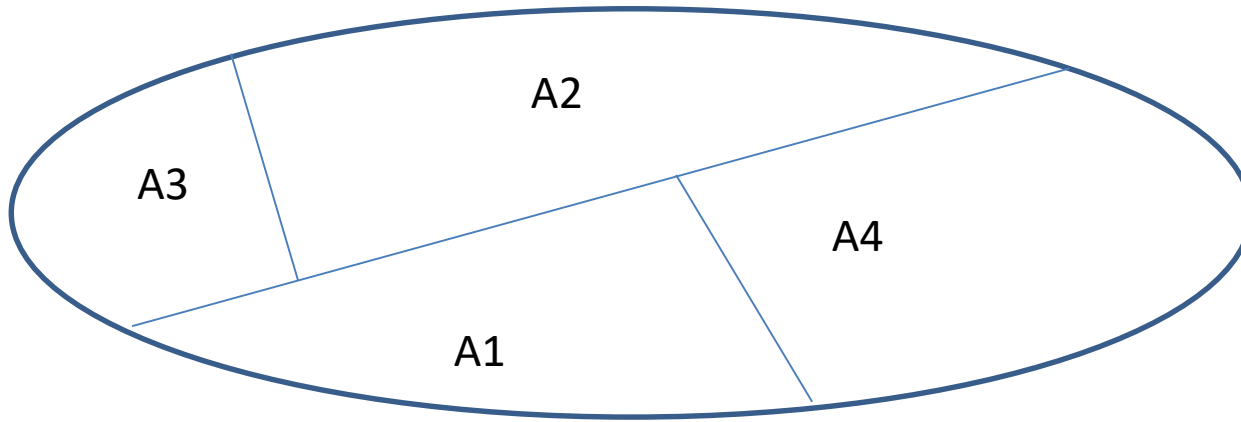


Hope ➡ If the test passes for one input-value of a partition, it will passes
for all other values of that partition

If the test fails for one input-value of a partition, it will fails
for all other values of that partition

# Partition: Strategies

- Boundaries (MAX_LONG, …)
  - Max possible amount
  - Value „above" boundaries (look „without any sence")
- Values at the center of the range
- Well known relevant values (could be zero)
- Values without any sence
- Values which must trigger an error/exception
  - null
  - Zero-length array
  - Not (completly) initialized value

# Partition: Example

```java
public int calculateInterestDays(
        Long startYear,
        Long startMonth,
        Long startDay,
        int finalYear,
        int finalMonth,
        int finalDay) {

    return 0; // that isn't correct ;)
}
```

# Partition: Example

```
public int calculateInterestDays(Long startYear, Long startMonth, Long startDay, int finalYear, int finalMonth, int finalDay) {
    return 0; // that isn't correct ;)
}
```

Month:
 1, 12, 6
 -1, 13, MAX_LONG
 null

# Partition: Example

```java
public int calculateInterestDays(Long startYear, Long startMonth, Long startDay, int finalYear, int finalMonth, int finalDay) {
    return 0; // that isn't correct ;)
}
```

Month:
 1, 12, 6
 -1, 13, MAX_LONG
 null

Year:
 0, 2000, 9999
2100, -9999
2020 (leap year)
2021 (non leap year)

# Partition: Example

```java
public int calculateInterestDays(Long startYear, Long startMonth, Long startDay, int finalYear, int finalMonth, int finalDay) {
    return 0; // that isn't correct ;)
}
```

Month:
1, 12, 6
-1, 13, MAX_LONG
null

Year:
 0, 2000, 9999
2100, -9999
2020 (leap year)
2021 (non leap year)
null

day:
 1, 15, 31
-1, 0, 32, MAX_LONG
null

# Partition: Example

```java
public int calculateInterestDays(Long startYear, Long startMonth, Long startDay, int finalYear, int finalMonth, int finalDay) {
    return 0; // that isn't correct ;)
}
```

Month:
1, 12, 6
-1, 13, MAX_LONG
null

Year:
 0, 2000, 9999
2100, -9999
2020 (leap year)
2021 (non leap year)
null

day:
 1, 15, 31
-1, 0, 32, MAX_LONG
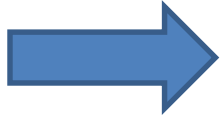null

combinations:
 Month 1, day 0, 15, 31, 32
Month 2, day 0, 14, 28

Leap year, Month 2, day 28, 29
Non Leap year, Month 2, day 28, 29
Same Day >>> Month 1, Day 1, Year 2100 second value equal

# Structure for test: „Triple A"-Pattern

**A**rrange the preconditions
**A**ct on the test object
**A**ssert the results

**Given** a precondition
**When** a thing happens
**Then** a result should be observable

# teststructure: „Triple A"-Pattern

```java
@Test
public void strukturEinesTest() {
    //GIVEN
    String anyOnlyElement = "A";
    Set<String> strings = new HashSet<String>();
    strings.add(anyOnlyElement);

    //WHEN
    strings.add(anyOnlyElement);

    //THEN
    assertTrue("expected size after adding same elements twice times", strings.size() == 1);
}
```
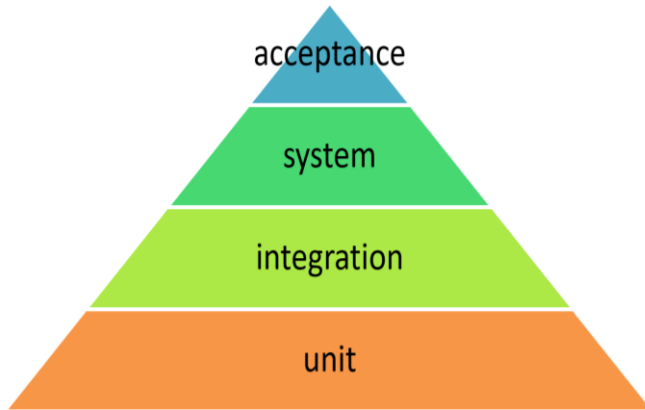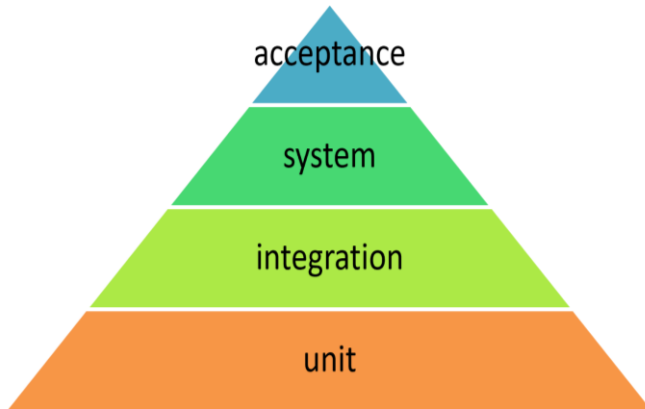
**Given** a precondition

**When** a thing happens

**Then** a result should be observable

# Cutback 1

- Quality
  - Absence of deficits
  - Absence of bugs

- Typical software Quality terms
  - Robustness
  - Efficience
  - Effectife
  - Reliability

# Cutback 2



- Errors in software
  - Requirements
    - Unclear
    - Complex
    - incomplete
  - Design
    - Error
    - Not understand
  - Architecture
    - Error
    - No understand
  - Code
    - Complex
    - Errors
    - Unpredictable behaviour sometime

- QA (quality assurance)
  - Defining quality guards
    - Ensure, that the product meets these guards
    - Improving over the lifecycle
  - Making tests
    - Try to make fails
    - Ensuring the correctness of the software
  - Testing can only show the precence of errors, never their absence.

# Cutback 3



acceptance
system
integration
unit
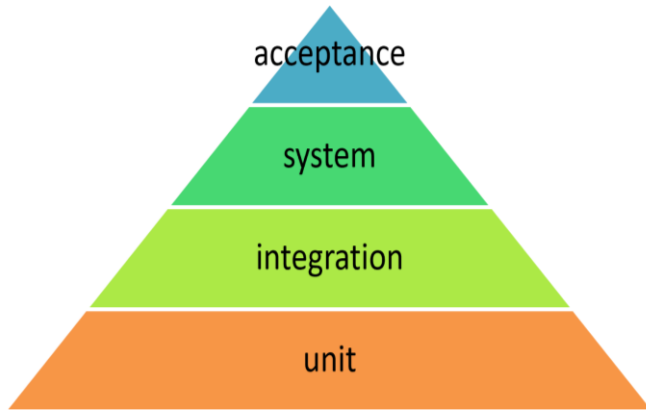
- Focus for unit-tests
  - Subject under test (SUT)
  - Only the class / method / function

- Rules for unit-tests
  - Test only one thing in one test
  - Tests must be undependent
  - Think about good names

# Cutback 4
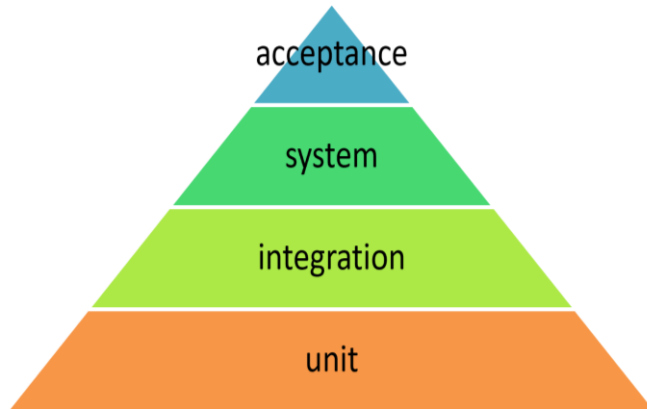
- Structure in tests
  - Triple „A"-pattern
    - (Arrange) GIVEN a
    - (Act) WHEN a
    - (Assert) THEN a
- Input-values for tests
  - Find partitions of equals values
  - Null-value
  - Zero (0)
  - Boundaries
  - Above boundaries
  - More inputs: equal inputs

acceptance

system

integration

unit

# Tests: what are possible tests? How many tests are recommendable?

```java
/**
 * With the AddMashine can be build a sum of added numbers and calculate the average.
 *
 * @author B. Hegmanns
 */
public class AddMashine {

    private Long value;
    private int addedNumbers;

    public AddMashine(Long initialValue) {
        this.value = initialValue;
        this.addedNumbers = 1;
    }

    public void add(Long value) {
        this.value += value;
        this.addedNumbers++;
    }

    public void sub(Long value) {
        this.value -= value;
        this.addedNumbers++;
    }

    public Long getSum() {
        return value;
    }

    public BigDecimal getAverage() {
        return new BigDecimal(addedNumbers).divide(new BigDecimal(value));
    }
}
```

de.hegmanns.tdd.task04.AddMaschine

# Coverage

```java
public void withdraw(BigDecimal money) {
    if (amount.compareTo(BigDecimal.ZERO) >= 0) {
        amount = amount.subtract(money);
        if (amount.compareTo(BigDecimal.ZERO) < 0) {
            throw new RuntimeException("account is now empty ...");
        }
    }else {
        throw new RuntimeException("account is empty");
    }
}
```

# Coverage: statement

```java
@Test
public void testInstructionCoverage1() {
    AccountWithGoodForCustomer account = new AccountWithGoodForCustomer(BigDecimal.ONE);
    RuntimeException throwedException = Assertions.assertThrows(RuntimeException.class,
            () -> account.withdraw(BigDecimal.TEN));
    Assertions.assertEquals("account is now empty ...", throwedException.getMessage());
}

@Test
public void testInstructionCoverage2() {
    AccountWithGoodForCustomer account = new AccountWithGoodForCustomer(BigDecimal.ONE.negate());
    RuntimeException throwedException = Assertions.assertThrows(RuntimeException.class,
            () -> account.withdraw(BigDecimal.ONE));
    Assertions.assertEquals("account is empty", throwedException.getMessage());
}
```
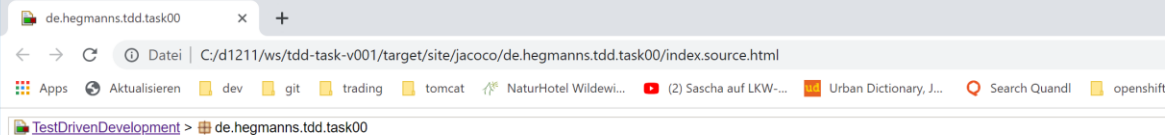
**AccountWithGoodForCustomer.java**

```java
1.  package de.hegmanns.tdd.task00;
2.
3.  import java.math.BigDecimal;
4.
5.  public class AccountWithGoodForCustomer {
6.
7.      private String id;
8.      private BigDecimal amount;
9.
10.     public AccountWithGoodForCustomer(BigDecimal amount) {
11.         this.amount = amount;
12.     }
13.
14.     public void withdraw(BigDecimal money) {
15.         if (amount.compareTo(BigDecimal.ZERO) >= 0) {
16.             amount = amount.subtract(money);
17.             if (amount.compareTo(BigDecimal.ZERO) < 0) {
18.                 throw new RuntimeException("account is now empty ...");
19.             }
20.         }else {
21.             throw new RuntimeException("account is empty");
22.         }
23.
24.
```
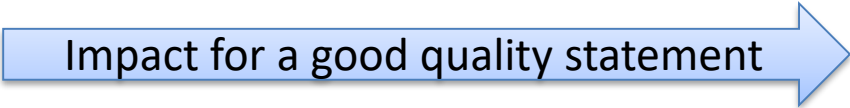
de.hegmanns.tdd.task00    ×    +

← → C    ① Datei   C:/d1211/ws/tdd-task-v001/target/site/jacoco/de.hegmanns.tdd.task00/index.source.html

Apps    Aktualisieren    dev    git    trading    tomcat    NaturHotel Wildewi...    (2) Sascha auf LKW-...    Urban Dictionary, J...    Search Quandl    openshift

TestDrivenDevelopment > de.hegmanns.tdd.task00

## de.hegmanns.tdd.task00

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---------|---------------------|------|-----------------|------|--------|------|--------|-------|--------|---------|--------|---------|
| AccountWithModifiedWithdraw02.java | | 0% | | 0% | 4 | 4 | 9 | 9 | 2 | 2 | 1 | 1 |
| AccountWithModifiedWithdraw01.java | | 0% | | 0% | 3 | 3 | 6 | 6 | 2 | 2 | 1 | 1 |
| Account.java | | 0% | | 0% | 3 | 3 | 6 | 6 | 2 | 2 | 1 | 1 |
| AccountWithGoodForCustomer.java | | 96% | | 75% | 1 | 4 | 1 | 9 | 0 | 2 | 0 | 1 |
| Total | 67 of 99 | 32% | 9 of 12 | 25% | 11 | 14 | 22 | 30 | 6 | 8 | 3 | 4 |

# Coverage: branch and path

```java
@Test
public void testBranchCoverage1() {
    AccountWithGoodForCustomer account = new AccountWithGoodForCustomer(BigDecimal.TEN);
    account.withdraw(BigDecimal.ONE);
}
```

**AccountWithGoodForCustomer.java**

```java
1.  package de.hegmanns.tdd.task00;
2.
3.  import java.math.BigDecimal;
4.
5.  public class AccountWithGoodForCustomer {
6.
7.      private String id;
8.      private BigDecimal amount;
9.
10.     public AccountWithGoodForCustomer(BigDecimal amount) {
11.         this.amount = amount;
12.     }
13.
14.     public void withdraw(BigDecimal money) {
15.         if (amount.compareTo(BigDecimal.ZERO) >= 0) {
16.             amount = amount.subtract(money);
17.             if (amount.compareTo(BigDecimal.ZERO) < 0) {
18.                 throw new RuntimeException("account is now empty ...");
19.             }
20.         }else {
21.             throw new RuntimeException("account is empty");
22.         }
23.     }
24.
```

de.hegmanns.tdd.task00 — C:/d1211/ws/tdd-task-v001/target/site/jacoco/de.hegmanns.tdd.task00/index.source.html

Apps · Aktualisieren · dev · git · trading · tomcat · NaturHotel Wildewi... · (2) Sascha auf LKW-... · Urban Dictionary, J... · Search Quandl · openshift

TestDrivenDevelopment > de.hegmanns.tdd.task00

## de.hegmanns.tdd.task00

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AccountWithModifiedWithdraw02.java | | 0% | | 0% | 4 | 4 | 9 | 9 | 2 | 2 | 1 | 1 |
| AccountWithModifiedWithdraw01.java | | 0% | | 0% | 3 | 3 | 6 | 6 | 2 | 2 | 1 | 1 |
| Account.java | | 0% | | 0% | 3 | 3 | 6 | 6 | 2 | 2 | 1 | 1 |
| AccountWithGoodForCustomer.java | | 100% | | 100% | 0 | 4 | 0 | 9 | 0 | 2 | 0 | 1 |
| Total | 66 of 99 | 33% | 8 of 12 | 33% | 10 | 14 | 21 | 30 | 6 | 8 | 3 | 4 |

# Some words of coverage metric

- 100% coverage says nothing
  - Are all important boundaries tested?
  - Are all related partitions considered?
  - A lazy and good developer can make 100% coverage in an easy way …

Impact for a good quality statement

Combine test with code revision / pair programming
Define quality guard, like „a commit may not decrease coverage"

# Black-box tests

- Oriented on behave without focus on implementation
  - Check output by given input
  - Based on specification

# White-box tests



- Oriented in structure / implementation
  - Needs programmcode / plan / activity-plan
  - Based on source code

# White-box tests

- Oriented in structure / implementation
  - Needs programmcode / plan / activity-plan
  - Based on source code
  - Covers the code-specification
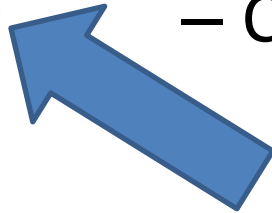
More fitting picture for white box test …

# Black box test: methods

- **Functional test**
  - Smoke test
  - Sanity test
  - Regression test
  - User acceptance test
  - Integration test
- **Non-functional test**
  - Usability test
  - Load test
  - Performance test
  - Stress test
  - Scalability test
  - Compatibility test

# white box test: methods

- **Basis path test**
- **Data flow test**
- **Path test**
- **Loop test**

# JUNIT 4 / 5

# Junit 4

- A unit-test framework for java
- Monolithic
- You can combine junit with other techniques
  - Spring
  - Java/Jacarta EE
- Extendible
  - … with limits

# Junit 5

- A unit-test framework for java

- Modularized

- Extendible

- Compatible with junit4

- Based on JAVA 8

# JUNIT5 dependencies

<junit.jupiter.version>5.3.0</junit.jupiter.version>

```xml
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>${junit.jupiter.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>${junit.jupiter.version}</version>
  <scope>test</scope>
</dependency>
```

# JUNIT5: parameterized tests

<junit.jupiter.version>5.3.0</junit.jupiter.version>

```xml
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>${junit.jupiter.version}</version>
  <scope>test</scope>
</dependency>
```

# JUnit5-Architektur

JUnit4-Tests

JUnit5-Tests

Other Tests

Vintage

Jupiter

3d party

Platform

Tools
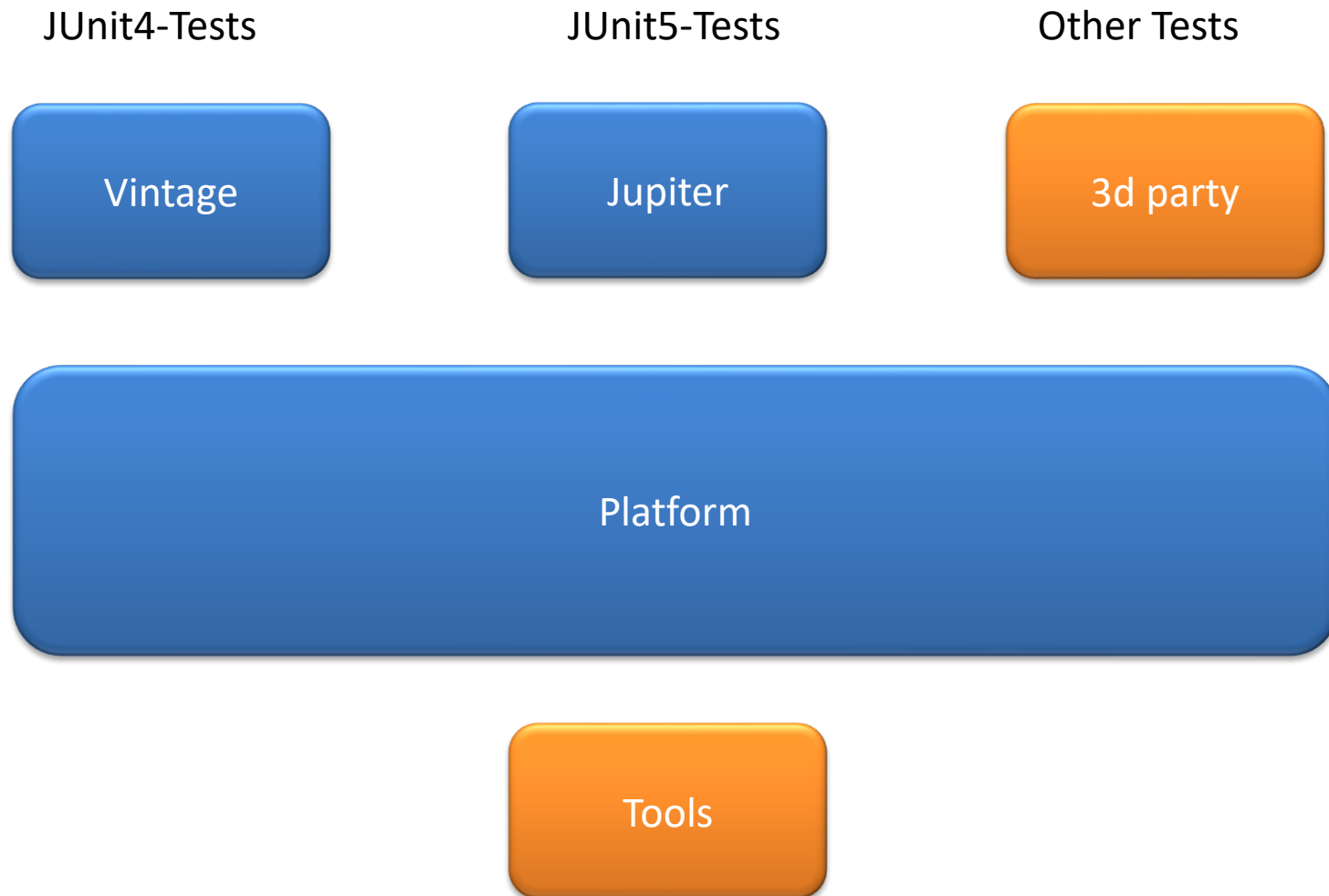
# Auto-Import in Eclipse
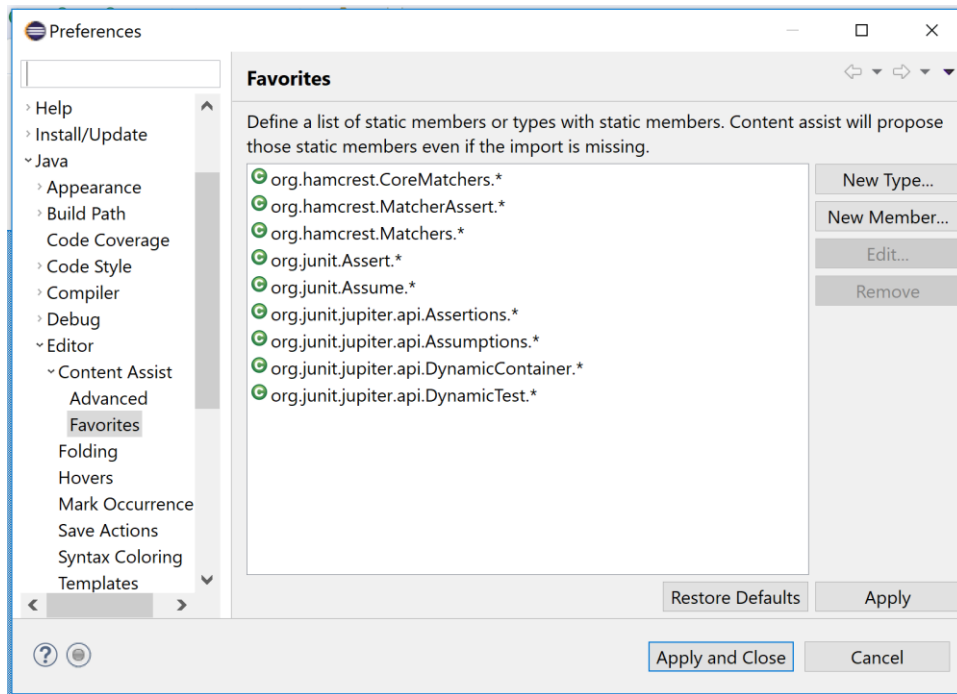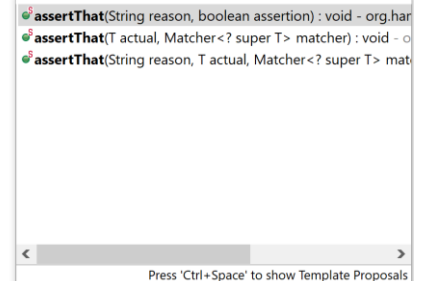
# JUnit5-Launcher
# (in new Eclipse-versions)
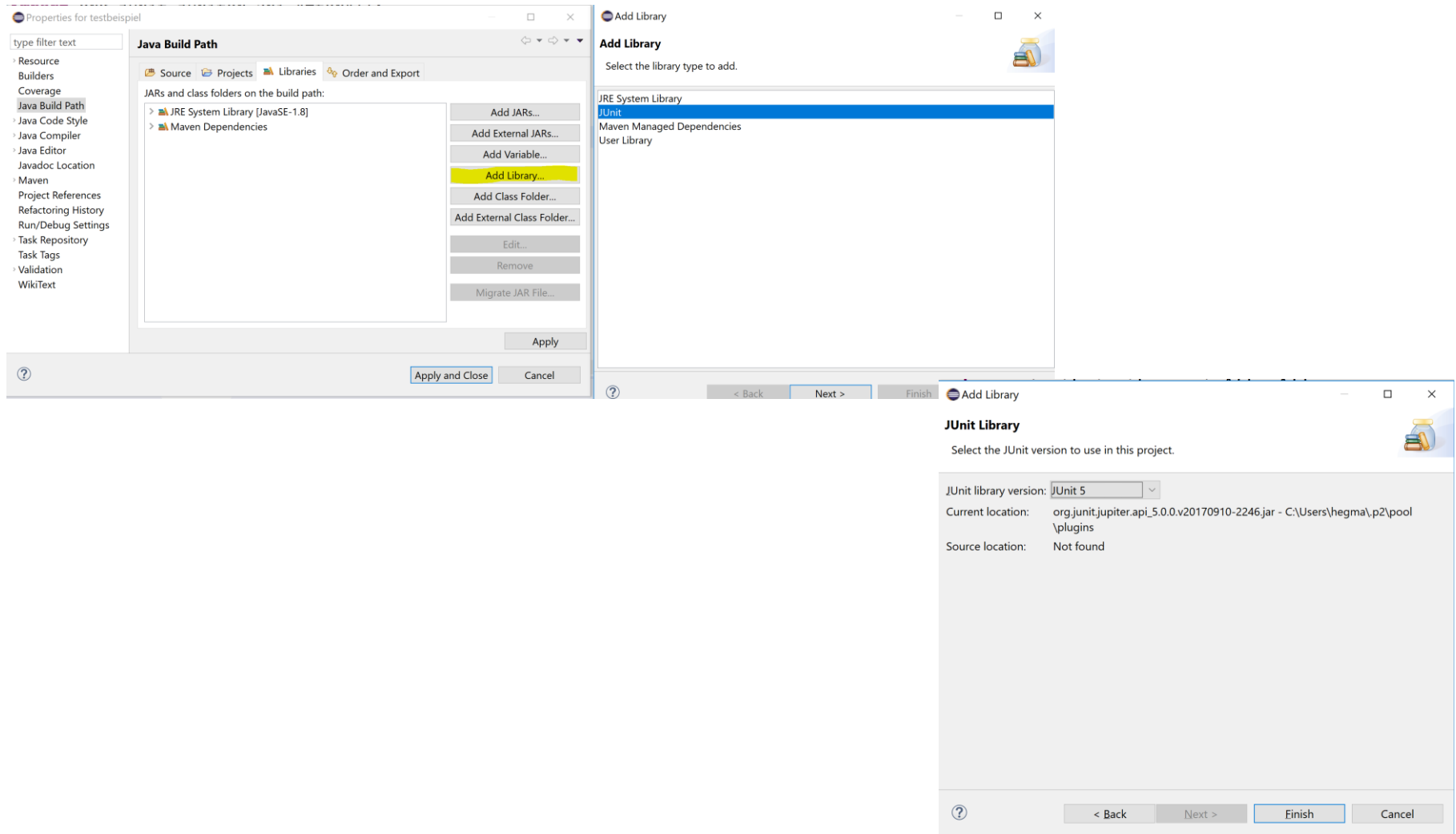
# Using JUnit4-Launcher
# (for older Eclipse-Versionen bzw. Launcher)

```java
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
public class LifecycleAnnotationen {
```

# HELPFUL für TDD: Shortcut

- In focus of TDD:
  - Quick retest
  - Code and test
- Well used shortcut „Ctrl+F11" (Relaunch last Run)

# What make junit4 / 5

- See a class as a test
  - Condition:
    At least one method is annotated with „Test"
  - Junit 4: org.junit.Test
  - Junit 5: org.junit.jupiter.api.Test
- All with @Test annotated methods are proceeded by the test runner
  - Als procceded tests are shown in the test runner windows

# Typical challenge in junit

- Comparing with expected value
  - Result of a method-call with expected result
  - A method-call changed properties of an object
  - A method-call throws an exception
  - A method-cal creates an event

# Assertions / Checks

- By using junit 5:  „org.junit.jupiter.Assertions"
  - Very large

- Third-Party: hamcrest

  - Talking syntax

  - Typesafe

  - Typesafe assertions

  - Extendible

# JUnit5-Assertions

- org.junit.jupiter.api.Assertions
  - May assert-methods

AND-link of asserts:

Assertions.assertAll(…..)

# Assertions.assertAll

```java
@Test
public void assertAllTest() {
    //GIVEN
    String anyOnlyElement = "A";
    Set<String> strings = new HashSet<String>();
    strings.add(anyOnlyElement);

    //WHEN
    strings.add(anyOnlyElement);

    //THEN
    Assertions.assertAll("set",
            () -> assertEquals(1, strings.size()),
            () -> assertEquals("A", strings.iterator().next(), "expected 'A'")
    );
}
```

# Assertions.assertAll: more asserts

```java
@Test
public void assertAllAllFailedTest() {
    //GIVEN
        String anyOnlyElement = "A";
        Set<String> strings = new HashSet<String>();
        strings.add(anyOnlyElement);

        //WHEN
        strings.add(anyOnlyElement);

        //THEN
        Assertions.assertAll("set",
                () -> assertEquals(2, strings.size()),
                () -> assertEquals("B", strings.iterator().next(), "expected 'B'")
        );
}
```

```
org.opentest4j.MultipleFailuresError: set (2 failures)
        expected: <2> but was: <1>
        expected 'B' ==> expected: <B> but was: <A>
        at org.junit.jupiter.api.AssertAll.assertAll(AssertAll.java:66)
```
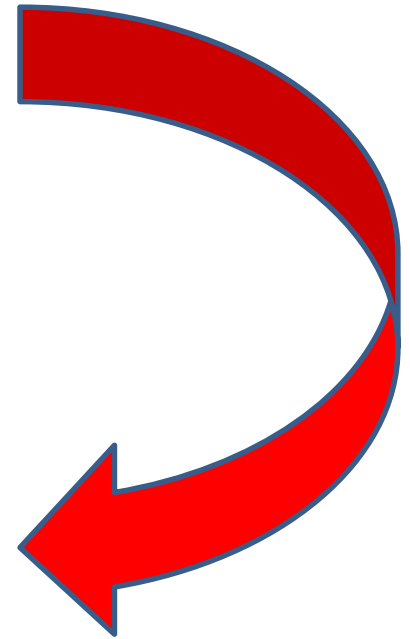
# Assertions.assertAll: more assert



```
@Test
public void assertAllAllFail
    //GIVEN
        String anyOnlyE
        Set<String> str
        strings.add(any

    //WHEN
    strings.add(any

    //THEN
    Assertions.asser
        () -> a
        () -> a
    );
}


org.opentest4j.Multi
    expected: <2
    expected 'B'
    at org.junit
```

**note**

Be carefull mit testing more than one assert in one test(method).

One test(method) should only test ONE aspect.

Sometimes it is a good idea testing more than one aspect.
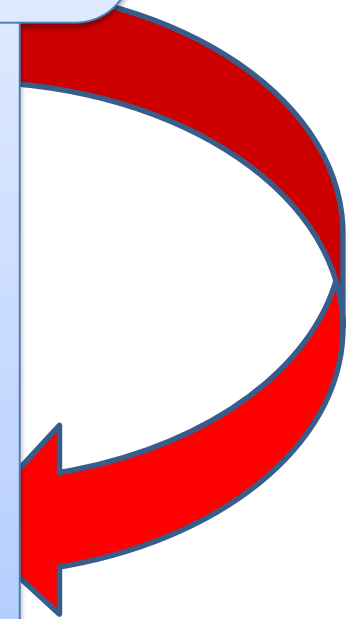
# Assertions.assertThrows

- Expected Exception
  - … and you can use it

```java
@Test
public void exceptionExpected() {
    //GIVEN
    int anyZaehler = 1;
    int zeroNenner = 0;

    //WHEN
    @SuppressWarnings("unused")
    ArithmeticException exception = assertThrows(ArithmeticException.class, () -> {int bruch = anyZaehler / zeroNenner;});

    //THEN
    assertEquals("/ by zero", exception.getMessage(), "expected Message");
}
```

# Assertions.assertThrows

```java
@Test
public void exceptionExpected() {
    //GIVEN
    int anyZaehler = 1;
    int nullNenner = 1;

    //WHEN
    @SuppressWarnings("unused")
    ArithmeticException exception = assertThrows(ArithmeticException.class, () -> {int bruch = anyZaehler / nullNenner;});

    //THEN
    assertEquals("/ by zero", exception.getMessage(), "expected Message");
}
```

```
org.opentest4j.AssertionFailedError: Expected java.lang.ArithmeticException to be thrown, but nothing was thrown.
        at org.junit.jupiter.api.AssertThrows.assertThrows(AssertThrows.java:65)
        at org.junit.jupiter.api.AssertThrows.assertThrows(AssertThrows.java:38)
        at org.junit.jupiter.api.Assertions.assertThrows(Assertions.java:1108)
```
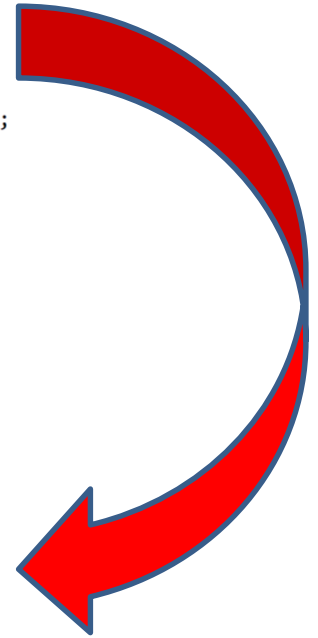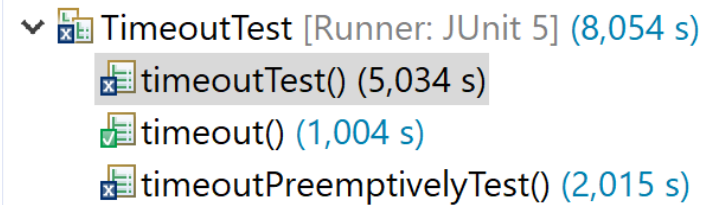
# Assertions.assertTimeout

- Max proceed time
  - With waiting (assertTimeout)
  - Without waiting (assertTimeoutPreemptively)

```java
@Test
public void timeoutTest() {
    assertTimeout(Duration.ofSeconds(2), () -> {Thread.sleep(5000);});
}

@Test
public void timeout() {
    assertTimeout(Duration.ofSeconds(2), () -> {Thread.sleep(1000);});
}

@Test
public void timeoutPreemptivelyTest() {
    assertTimeoutPreemptively(Duration.ofSeconds(2), () -> {Thread.sleep(5000);});
}
```

TimeoutTest [Runner: JUnit 5] (8,054 s)
  timeoutTest() (5,034 s)
  timeout() (1,004 s)
  timeoutPreemptivelyTest() (2,015 s)

# Unit-tests

- Unit-Test tests a concrete object/method
  - Well known as „SUBJECT UNDER TEST" (SUT)

- Rules for unit-test
  - Only test the SUBJECT UNDER TEST
    (not other objects)
  - Only test one thing (output, …) (make more tests)
  - Do not test with environment
  - Do not test with database

# Good name inside unit-test

- Name should describe the use-case
- No continuing numbers
- No fantasy names
- Use possible width
  - Length of names
  - Width of monitor
- Test-classes and methods must not satisfy codestyle rules (PMD)
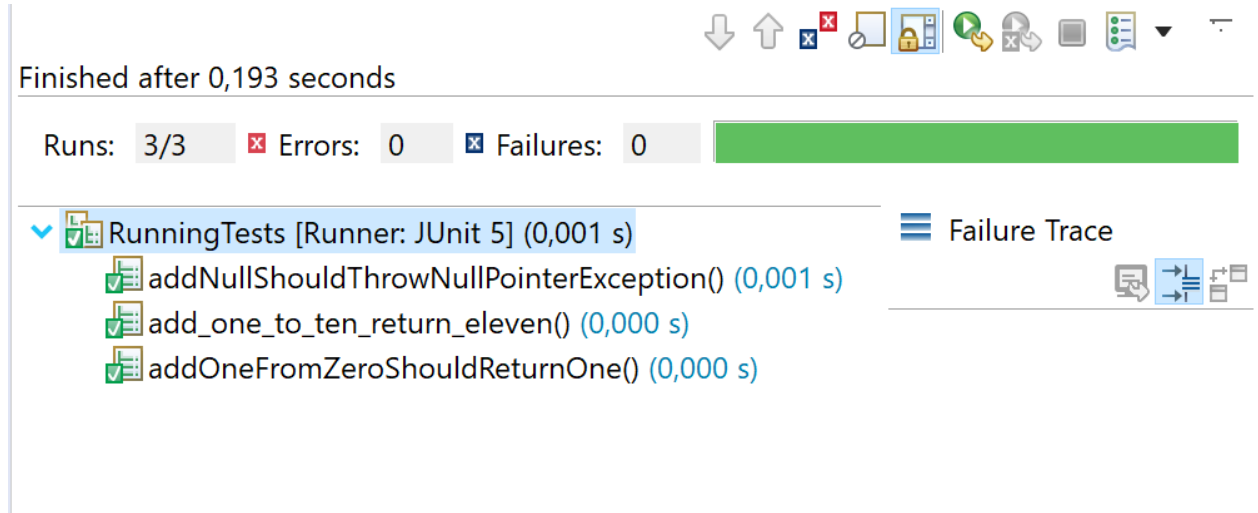
# Good name inside unit-test

public void addOneFromZeroShouldReturnOne()

public void addNullShouldThrowNullPointerException()

public void add_one_to_ten_return_eleven()

Finished after 0,193 seconds

| Runs: | 3/3 | ☒ Errors: | 0 | ☒ Failures: | 0 | |

❯ ▦ RunningTests [Runner: JUnit 5] (0,001 s)
    ✓ addNullShouldThrowNullPointerException() (0,001 s)
    ✓ add_one_to_ten_return_eleven() (0,000 s)
    ✓ addOneFromZeroShouldReturnOne() (0,000 s)

☰ Failure Trace

# Good name inside unit-test

public void addOneFromZeroShouldReturnOne()

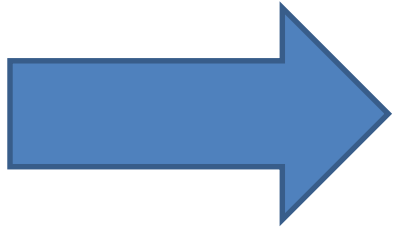public void addNullShouldThrowNullPointerException()

public void add_one_to_ten_return_eleven()

# Unit-test idea:
# look the test-result und you know all

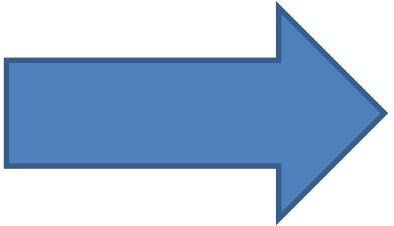Looking into the test or stack-trace wastes time.
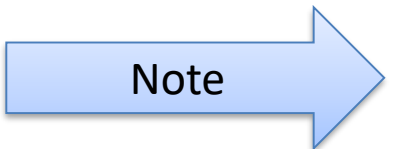
# Typical work in a junit test

Call a method
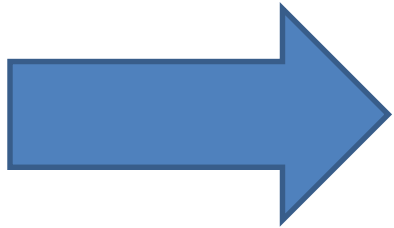
Fetch the result

compare the fetched result
with your expectation

Note → Before your call should be some
preparations necesary
(create instance, …)

# Benefits of unit-tests

- Precondition for refactoring
  - You must have sufficient tests
  - You may not have any red test
- Your classes are testable
  - Its a sign of good design

Martin Fowler:

   If you want to refactor , the essential precondition is having solid tests.

# maintainable software with unittest

- No untestet code
- Clean architecture
- Clean design
- No redudancy
- Precondition for refactoring

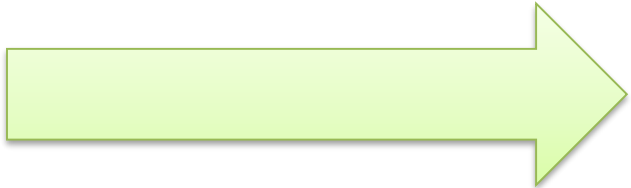# From problem/use-case to solution with unit-test

- No solution on stock
- No solution for future (and unknown) problems
- No code on stock
- You do only just necessary
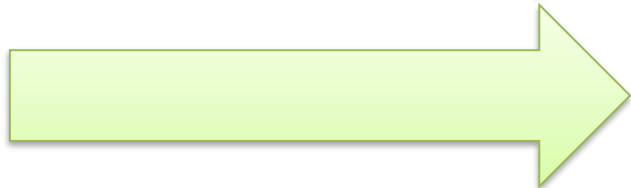
# Rules for developing

- You make a blind flight without unit-test
  - New features
  - bugfix
  - refactoring
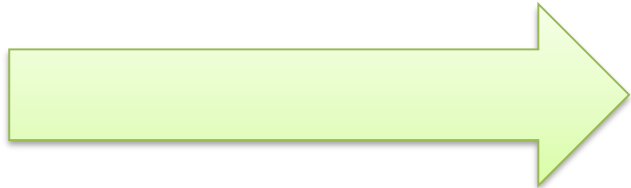
Make a RED test.
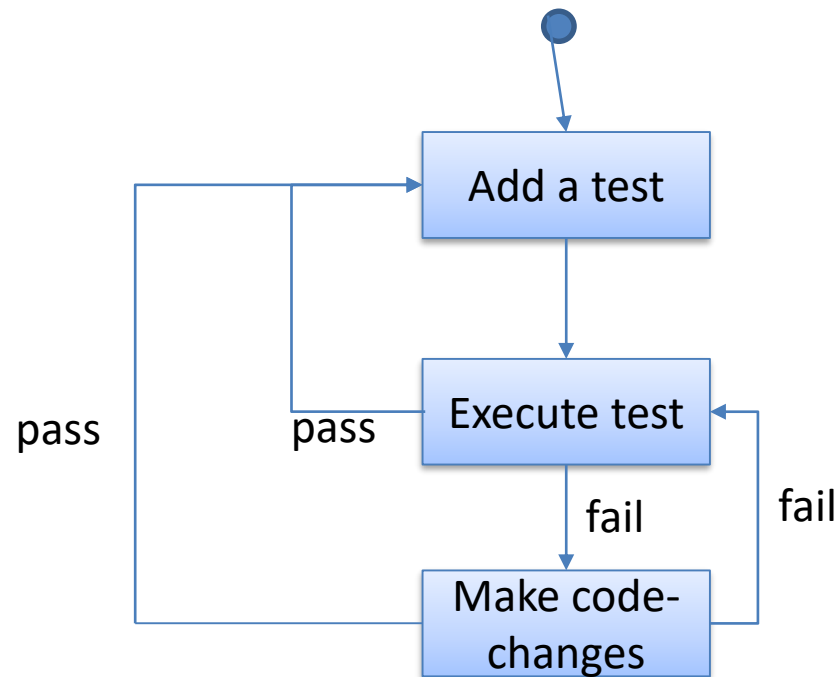Note: Without red test no change

Make changes

Check test for green.

# TDD

- Test Driven Design
- An agile developing method
  - For quick results
  - For good results
    - Means: simple, clear
  - For testet results
    - Means: bug free

# TDD-activities

# TDD process

- Building the complete feature/requirement in very small steps

  - Plan the steps

- Work the TDD-Lifecycle for every small step
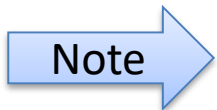
Note → If you are working on a step and your test passes.
Then the step is done.

If you are working on a step and you recognize heavy test-scenario.
Think about refactoring. (but only for green test)

# TDD-step example

The password-checker only accepts passwords between 5 and 10 characters (inclusive)
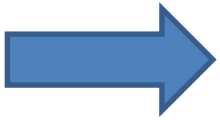
What are good tests?
(Remember equivalence classes)

# Testing more than one class

- Junit-test may only test one method
  - You do not know what fails if fail

- But sometime, a method needs some other objects …

- If you tests all objects, results depend from the result those objects

# More Objects: Example

```java
public class CurrencyConverter {

    public BigDecimal convert(BigDecimal amount, String currencyFrom, String currencyTo) {

        CurrencyDto dto = new CurrencyDto();
        BigDecimal exchangeRate = dto.getExchangeRate(currencyFrom, currencyTo);

        return amount.multiply(exchangeRate);
    }
}
```

- Result depends vom current rate
    - You do not know the current result
      (without looking for current rate)
    - You have to change the expected result in tests

# Testing more than one class

- Junit-test may only test one method
  - You do not know what fails if fail

- But sometime, a method needs some other objects
  - Refactoring
  - Use defined objects with defined results (proxy-objects)

# Proxy-Objects or Mocks

- A Proxy-Object is a deputy for an other object with defined behave.

- You make your tests predictable

A special DTO not connected with database or internet result the same result for every call.

E.g. exchange rate ONE every time.

# creating and using proxy objects

```java
public class ProxyCurrencyDto extends CurrencyDto{

    @Override
    public BigDecimal getExchangeRate(String currencyFrom, String currencyTo) {
        return BigDecimal.TEN;
    }

}
```
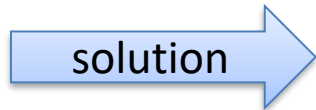
For integration the proxy you need to refactor the CurrencyConverter class

# Benefits for using proxies

- You test only your SUT

- You get a good design
  - You use delegating
  - You use polymorphy

# Mock

- ## Self created proxies
  - – For easy output easy to build
  - – For complex and changing output complex

solution → Mock-Frameworks (mockito, easymock)

Proxy objects with integrated logic
The logic can be called in various situations
Predefining output for the same proxy/mock
Predefining behave for the came proxy (e.g. throwing exception)

# Mock example

```java
@Test
public void absolutMatching(){

    RateProvider provider = Mockito.mock(RateProvider.class);
    when(rateProvider.getFactory(Currency.USD, Currency.EURO)).thenReturn(BigDecimal.TEN);

    BigDecimal amount = converter.convert(Currency.UST, Corrency.EURO, BigDecimal.ONE);
    assertThat(amount, compareEquals(BigDecimal.TEN));
}
```

# Mock: other features

- Mock checks execution
- Mock checks order of execution

```java
public class CoreListSpyTest {

    @Test
    public void exampeSpy() {
        List<String> strings = new ArrayList<>();

        List<String> spyStrings = Mockito.spy(strings);

        spyStrings.addAll(Arrays.asList("a", "b", "c"));
        spyStrings.add("d");
        spyStrings.add("d");

        int countElements = spyStrings.size();

        assertThat(countElements, is(4));
        //better:
        assertThat(spyStrings, hasSize(4));

        Mockito.verify(spyStrings).size();
        Mockito.verify(spyStrings, never()).add("a");
        Mockito.verify(spyStrings, never()).add("b");
        Mockito.verify(spyStrings, times(2)).add("d");
    }
}
```