



TECHNISCHE HOCHSCHULE MITTELHESSEN

**THM**

**CAMPUS  
GIESSEN**

**MNI**

Mathematik, Naturwissenschaften  
und Informatik

# Dokumentieren und Testen

CS1016 Programmierung interaktiver Systeme

von Prof. Dr. Weigel



# Dokumentieren und Testen

## Dokumentieren

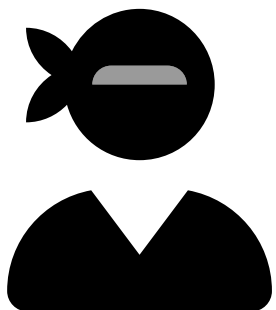
1. README
2. JavaDoc

## Testen

1. JUnit
2. Test-Coverage

# Warum Dokumentieren?

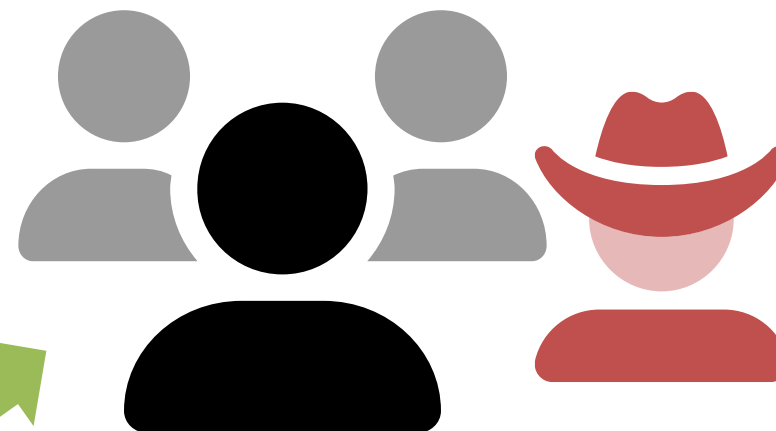
Aktuelles Sie



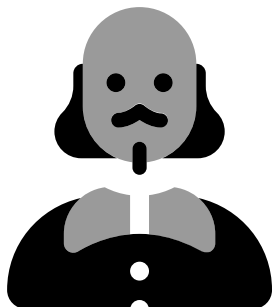
Quellcode



Ihre Kolleg\*innen & Chef\*in



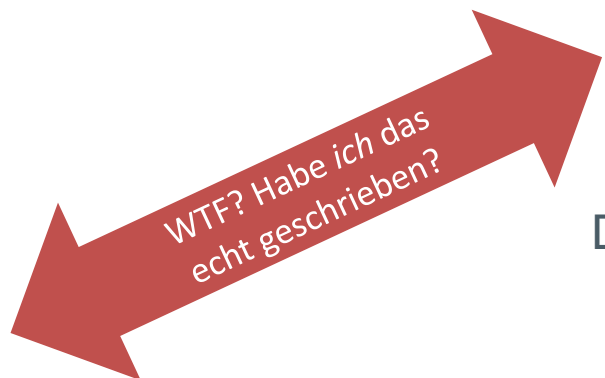
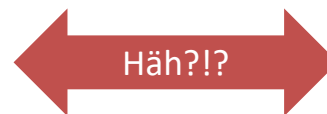
Älteres Sie



Dokumentation



Textuelle Beschreibung  
was der Code macht  
(Abstrakt und im Detail)



Sie schreiben Dokumentationen  
hauptsächlich für andere Menschen  
→ Auf Verständlichkeit achten



# Dokumentieren und Testen

## Dokumentieren

1. **README**
2. JavaDoc

## Testen

1. JUnit
2. Test-Coverage

# README

Enthält die wichtigsten Informationen zur Software

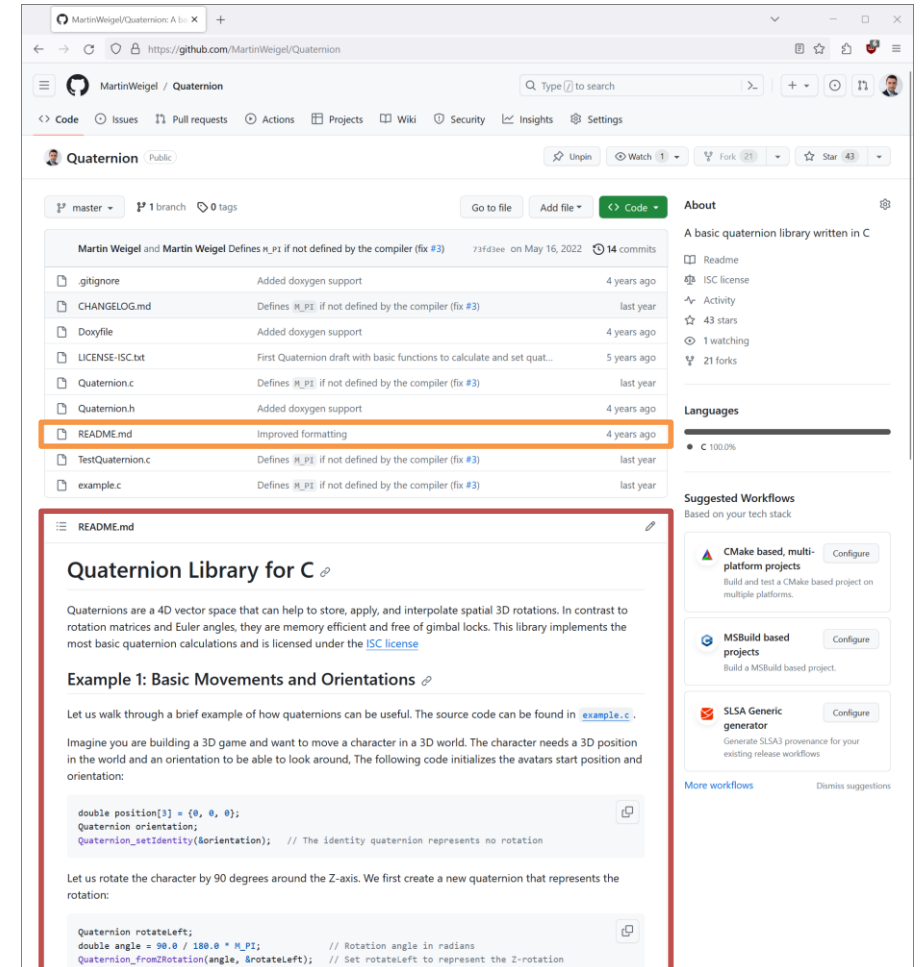
- Beschreibung / Verwendungszweck
- Installationsanleitung
- Einstellungen und Parameter
- Beispiele
- ...

→ Was macht das Programm (nicht der Code)?

Liegt meist im **Hauptverzeichnis** des Projektes

Wird in vielen Entwicklungswerkzeugen direkt **angezeigt** →

In einem von Menschen lesbaren Dateiformat (.txt, .md, .rst)



# README in Markdown (.md)

Markdown ist eine vereinfachte Auszeichnungssprache zum Schreiben von Dokumenten  
Entwickelt von John Gruber im Jahr 2004. Kann einfach in HTML konvertiert werden.

```

1 # Ich bin eine Überschrift
2
3 Normaler Text mit *kursiven* und **fetten** Wörtern.
4
5 - Eine
6 - Kleine
7 - Liste
8
9 ## Eine Unterüberschrift
10
11 Und noch eine:
12
13 1. Liste
14 2. Mit
15 3. Nummern
16
17 ## Und noch eine Unterüberschrift
18
19 So long and thanks for all the ...
20
21 ![Ein Karpador](https://assets.pokemon.com/assets/cms2/img/pokedex/
22 full/129.png)
23 ... HTML-Fan? HTML ist in Markdown auch möglich:
24
25 

```

## Ich bin eine Überschrift

Normaler Text mit *kursiven* und **fetten** Wörtern.

- Eine
- Kleine
- Liste

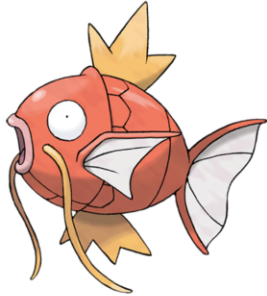
### Eine Unterüberschrift

Und noch eine:


1. Liste
2. Mit
3. Nummern

### Und noch eine Unterüberschrift

So long and thanks for all the...



... HTML-Fan? HTML ist in Markdown auch möglich:



## README im Programmierprojekt (Screenshot WiSe '22/'23)

### Dokumentation (DOK): README

- ☐ **Must-Have:** Die README-Datei enthält eine **Kurzbeschreibung** des Programmes. Alle verwendeten Quellen (z.B. externe Bibliotheken) sind dokumentiert und verlinkt.
- ☐ Die README-Datei enthält einen oder mehrere **Screenshots** des Programmes.
- ☐ Die README-Datei dokumentiert die notwendigen Schritte, um das Programm zu **starten**.
- ☐ Die README-Datei dokumentiert ein **Beispiel**, um das Modell in der **JShell** zu testen.

<input type="checkbox"/> <b>0.5</b>	<input type="checkbox"/> <b>0.375</b>	<input type="checkbox"/> <b>0.25</b>	<input type="checkbox"/> <b>0</b>
Anforderungen komplett erfüllt	Überwiegend zutreffend	Ausreichende Umsetzung	Unzureichende Umsetzung

(Es kann in diesem Semester noch zu Abweichungen des abgebildeten Bewertungsschemas kommen.)



# Dokumentieren und Testen

## Dokumentieren

1. README
2. **JavaDoc**

## Testen

1. JUnit
2. Test-Coverage



## Dokumentieren des Quellcodes

Es sollten dokumentiert werden: *Pakete, Klassen, Interfaces, Records, Enums, Variablen, ...*

### ***Was sollte die Dokumentation enthalten:***

- High-Level Konzepte: “Was macht die Klasse/die Methode/das Paket?”
- Möglicher Wertebereich von Parametern und Rückgabewerte
- Alle Exceptions die geworfen werden können und wann sie geworfen werden
- Typische Nutzung (in welchem Kontext wird das Element häufig verwendet)

### ***Welche Elemente werden für wen dokumentiert?***

- Benutzer des Quellcodes? → public
- Andere Entwickler am gleichen Package? → public/protected/-
- Sich selbst oder nachfolgende Maintainer? → Alles (nicht offensichtliche)

**Aber: Kurz und präzise**

# JavaDoc

Dokumentationswerkzeug für Java. Teil vom JDK. Dokumentation durch spezielle Java-Kommentare.

## Vorteile:

- 😊 Standardisiertes Dokumentationsformat
  - Andere Entwickler kennen das Format
  - In vielen Java-IDEs und Anwendungen nutzbar (viele Exportmöglichkeiten)
- 😊 Dokumentation steht neben dem Quellcode
  - Einfache Anpassung bei Code-Änderung

## Nutzung:

- JavaDoc Kommentare starten mit **/\*\*** und enden mit **\*/**
- Kommentare beziehen sich immer auf das nächste Code-Element
- Tags erlauben Metadaten der Dokumentation (Tags starten mit @)

## Paketdokumentation (in `package-info.java`)

Die Paketdokumentation soll einen high-level Überblick über das Paket vermitteln:

*Welche Aufgaben hat das Paket? Wie soll es verwendet werden? Von welchen anderen Paketen hängt es ab?*

```
/**
```

```
 * The pokemon module contains all available Pokemon and their types.
```

```
 * Pokemon are animal-like creatures that can fight each other.
```

```
 * <p>
```

```
 * New types can be added to <pre>Type.java</pre>.
```

```
 * New Pokemon should be added as a new class.
```

```
 *
```

```
 * @since 1.0
```

```
 * @author Martin Weigel
```

```
 * @version 1.2
```

```
 */
```

```
package pokemon;
```

Paket Beschreibung  
HTML-Formatierung  
ist möglich (z.B. <p>)

} Version seit der das Paket existiert

} Name des verantwortlichen Autors (auch mehrfach setzbar)

} Version der Software

## Klassendokumentationen (auch Interfaces, Records, Enums)

Die Klassendokumentation soll einen high-level Überblick über die Klasse vermitteln:

*Welche Aufgaben hat die Klasse? Wie soll sie verwendet werden? Von welchen anderen Klassen hängt sie ab?*

```
package ninja;
```

```
import main.teams.Fightable;
```

```
/**
```

```
 * A Ninja is a fighter that can fight other Ninjas.
```

```
 *
```

```
 * @see https://en.wikipedia.org/wiki/Ninja
```

```
 * @author Martin Weigel
```

```
 * @since 1.0
```

```
 * @version 1.0
```

```
 */
```

```
public class Ninja implements Fightable<Ninja> {
```

```
    //...
```

```
}
```

## Variablendokumentation

Beschreibung was die Variable macht und wie sie genutzt werden soll/kann:

```
/**  
 * The horizontal position, where the Pokemon is displayed (always positive).  
 */
```

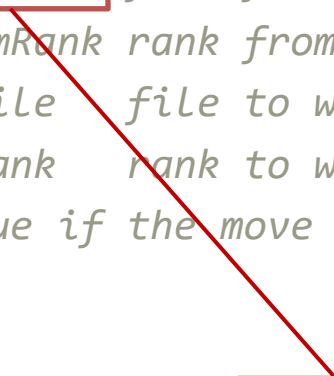
```
public int x;
```

```
/**  
 * The vertical position, where the Pokemon is displayed (always positive).  
 */
```

```
public int y;
```

# Methodendokumentationen

```
/**
 * Validates a chess move.
 *
 * <p>Use {@link #doMove(int fromFile, int fromRank, int toFile, int toRank)} to move a piece.
 *
 * @param fromFile file from which a piece is being moved
 * @param fromRank rank from which a piece is being moved
 * @param toFile file to which a piece is being moved
 * @param toRank rank to which a piece is being moved
 * @return true if the move is valid, otherwise false
 * @since 1.0
 */
boolean isValidMove(int fromFile, int fromRank, int toFile, int toRank) {
    // ...body
}
```



# Übersicht der JavaDoc Tags

Tag & Parameter	Usage	Applies to	Since
<b>@author</b> <i>John Smith</i>	Describes an author.	Class, Interface, Enum	
<b>{@docRoot}</b>	Represents the relative path to the generated document's root directory from any generated page.	Class, Interface, Enum, Field, Method	
<b>@version</b> <i>version</i>	Provides software version entry. Max one per Class or Interface.	Class, Interface, Enum	
<b>@since</b> <i>since-text</i>	Describes when this functionality has first existed.	Class, Interface, Enum, Field, Method	
<b>@see</b> <i>reference</i>	Provides a link to other element of documentation.	Class, Interface, Enum, Field, Method	
<b>@param</b> <i>name description</i>	Describes a method parameter.	Method	
<b>@return</b> <i>description</i>	Describes the return value.	Method	
<b>@exception</b> <i>classname description</i> <b>@throws</b> <i>classname description</i>	Describes an exception that may be thrown from this method.	Method	
<b>@deprecated</b> <i>description</i>	Describes an outdated method.	Class, Interface, Enum, Field, Method	
<b>{@inheritDoc}</b>	Copies the description from the overridden method.	Overriding Method	1.4.0
<b>{@link}</b> <i>reference</i>	Link to other symbol.	Class, Interface, Enum, Field, Method	
<b>{@linkplain}</b> <i>reference</i>	Identical to {@link}, except the link's label is displayed in plain text than code font.	Class, Interface, Enum, Field, Method	
<b>{@value}</b> <i>#STATIC_FIELD</i>	Return the value of a static field.	Static Field	1.4.0
<b>{@code}</b> <i>literal</i>	Formats literal text in the code font. It is equivalent to <code>&lt;code&gt;{@literal}&lt;/code&gt;</code> .	Class, Interface, Enum, Field, Method	1.5.0
<b>{@literal}</b> <i>literal</i>	Denotes literal text. The enclosed text is interpreted as not containing HTML markup or nested javadoc tags.	Class, Interface, Enum, Field, Method	1.5.0
<b>{@serial}</b> <i>literal</i>	Used in the doc comment for a default serializable field.	Field	
<b>{@serialData}</b> <i>literal</i>	Documents the data written by the writeObject( ) or writeExternal( ) methods.	Field, Method	
<b>{@serialField}</b> <i>literal</i>	Documents an ObjectOutputStreamField component.	Field	

[Bildquelle: <https://en.wikipedia.org/wiki/Javadoc>]

# Generieren einer HTML Dokumentation mit javadoc

## Kommandozeile:

```

javadoc -d doc -sourcepath src -subpackages pokemon:ninja
      -cp ./lib/core.jar

```

Ausgabeordner      Quellcode Ordner      Package 1      Package 2

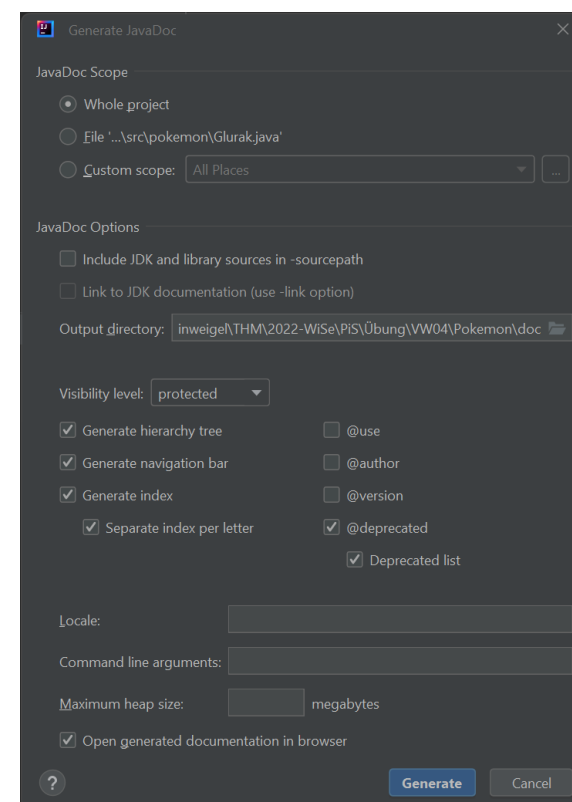
Class Path (z.B. Bibliotheken)

Teil vom JDK unter ~\jdk\openjdk-21\bin

Der Pfad muss eventuell den Umgebungsvariablen hinzugefügt werden

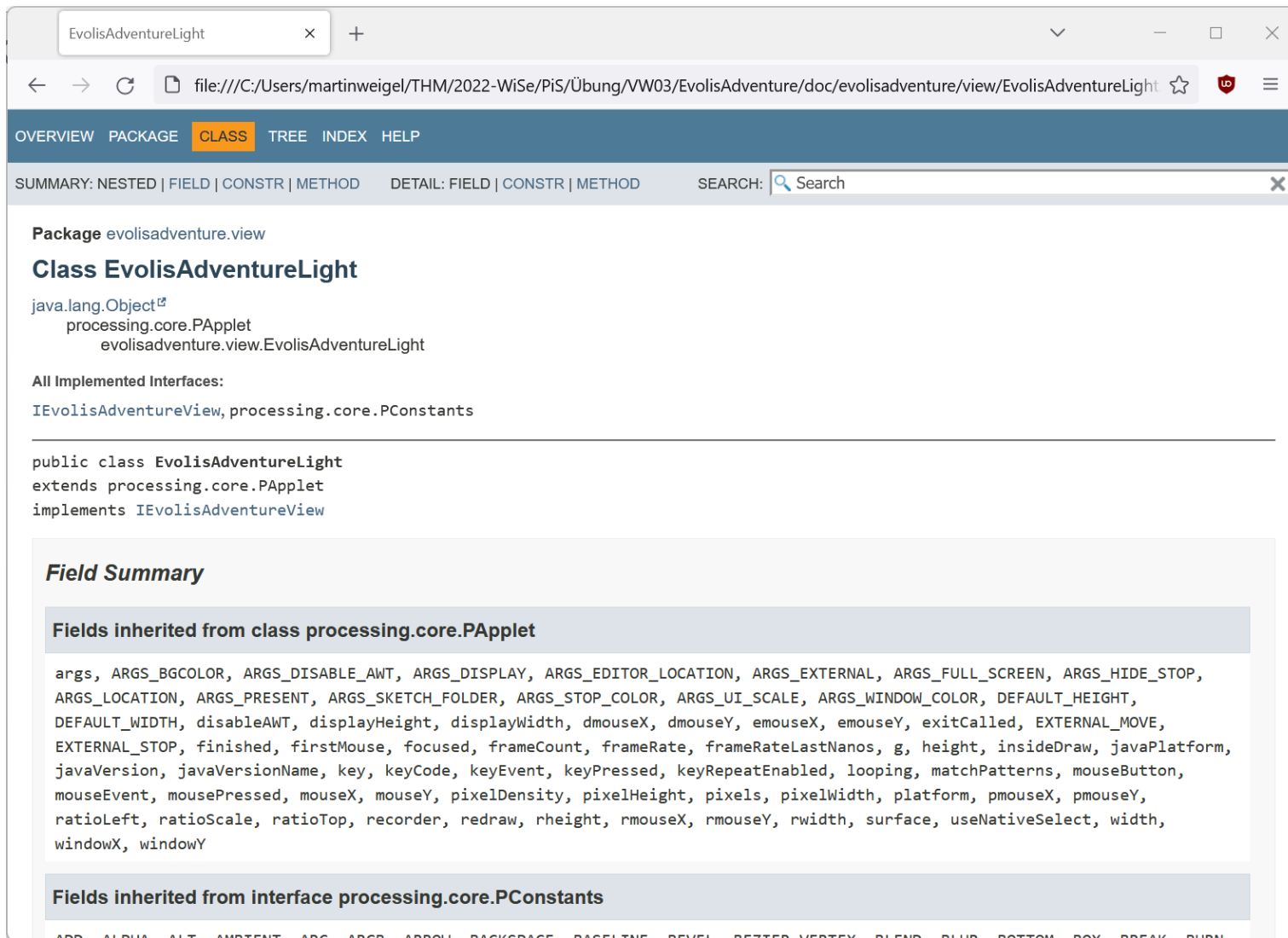
## IntelliJ:

“Tools > Generate JavaDoc”





# Demo



The screenshot shows a web browser window with the address bar displaying a file path. The browser has a single tab titled 'EvolisAdventureLight'. The page content is a Javadoc for the 'EvolisAdventureLight' class, organized into several sections:

- Navigation Bar:** Includes links for OVERVIEW, PACKAGE, CLASS (highlighted), TREE, INDEX, and HELP.
- Summary Bar:** Includes links for SUMMARY: NESTED, FIELD, CONSTR, METHOD, and a search bar.
- Package:** evolisadventure.view
- Class:** EvolisAdventureLight
- Class Hierarchy:**

```

java.lang.Object
├── processing.core.PApplet
└── evolisadventure.view.EvolisAdventureLight
          
```
- All Implemented Interfaces:**

```

IEvolisAdventureView, processing.core.PConstants
          
```
- Class Declaration:**

```

public class EvolisAdventureLight
    extends processing.core.PApplet
    implements IEvolisAdventureView
          
```
- Field Summary:**
  - Fields inherited from class processing.core.PApplet:**

```

args, ARGS_BGCOLOR, ARGS_DISABLE_AWT, ARGS_DISPLAY, ARGS_EDITOR_LOCATION, ARGS_EXTERNAL, ARGS_FULL_SCREEN, ARGS_HIDE_STOP,
ARGS_LOCATION, ARGS_PRESENT, ARGS_SKETCH_FOLDER, ARGS_STOP_COLOR, ARGS_UI_SCALE, ARGS_WINDOW_COLOR, DEFAULT_HEIGHT,
DEFAULT_WIDTH, disableAWT, displayHeight, displayWidth, dmouseX, dmouseY, emouseX, emouseY, exitCalled, EXTERNAL_MOVE,
EXTERNAL_STOP, finished, firstMouse, focused, frameCount, frameRate, frameRateLastNanos, g, height, insideDraw, javaPlatform,
javaVersion, javaVersionName, key, keyCode, keyEvent, keyPressed, keyRepeatEnabled, looping, matchPatterns, mouseButton,
mouseEvent, mousePressed, mouseX, mouseY, pixelDensity, pixelHeight, pixels, pixelWidth, platform, pmouseX, pmouseY,
ratioLeft, ratioScale, ratioTop, recorder, redraw, rheight, rmouseX, rmouseY, rwidth, surface, useNativeSelect, width,
windowX, windowY
          
```
  - Fields inherited from interface processing.core.PConstants:**

```

ADD, ALPHA, ALT, AMBIENT, APC, ARGB, ARROW, BACKSPACE, BASELINE, BEVEL, BEZIER_VERTEX, BLEND, BLUR, BOTTOM, BOX, BREAK, BURN
          
```



# Dokumentieren und Testen

## Dokumentieren

1. README
2. JavaDoc

## Testen

1. JUnit
2. Test-Coverage

# Test Arten

## Manual & Exploratory Testing

Manuelles Ausprobieren des Programmes

→ Aufwändig (Zeit und Arbeitsintensiv, nicht automatisiert)

## E-2-E Tests (aka. Systemtest)

Test des kompletten Systems

Meist durch den Kunden oder Manager

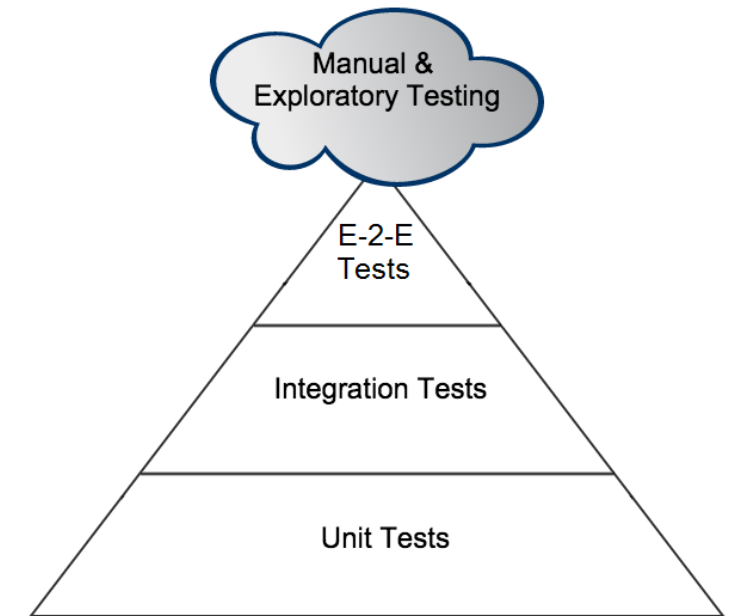
## Integration Tests

Testet das Zusammenspiel mehrerer Programmkomponenten

### Unser Fokus

## Komponenten Tests (aka. Unit Test)

Testen eines einzelnen Verhaltens (=Unit) einer Klasse/Methode



[Bildquelle: <https://www.endoflineblog.com/unit-acceptance-or-functional-demystifying-the-test-types-part2>]

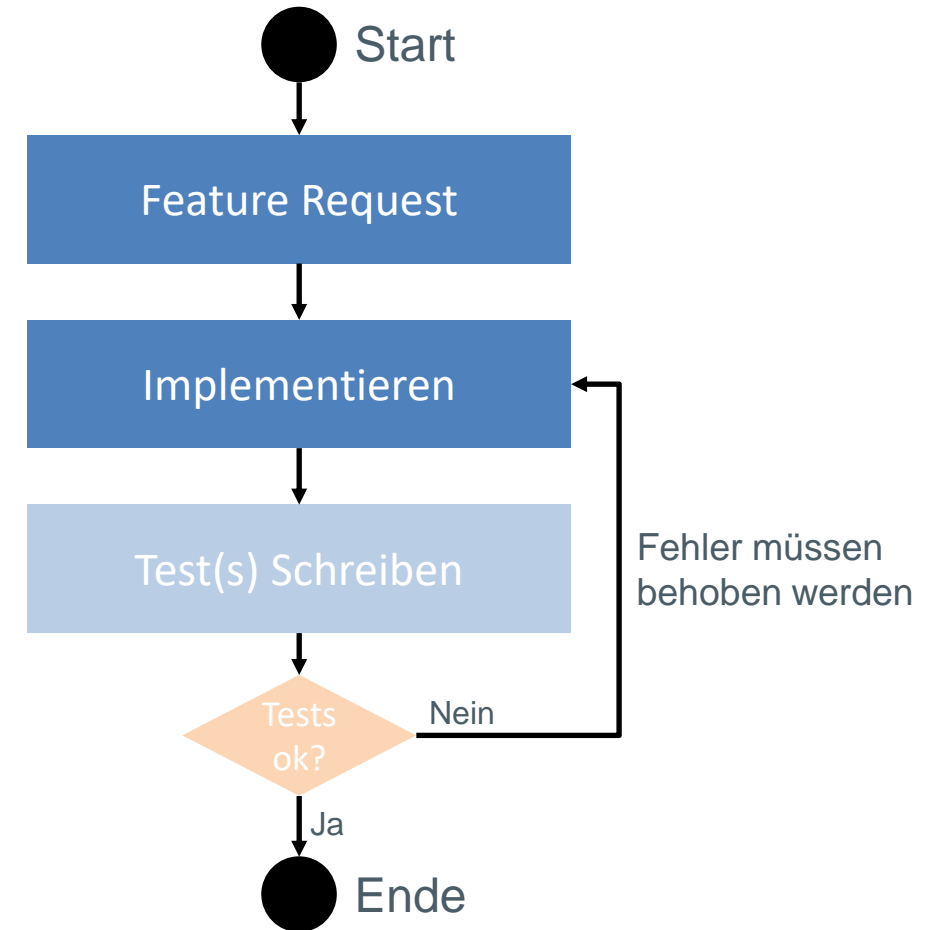
## “Traditionelles” Testen

Erst wird der Code implementiert  
**Danach** werden Tests geschrieben

Vom Entwickler selbst oder einer Test-Abteilung

### ☹ Probleme

1. Test-Phase wird häufig “vergessen”  
(z.B. Zeit “sparen” vor Deadlines)
2. Nicht alle Code-Teile werden getestet  
Der Test deckt nicht das komplette Verhalten ab
3. Tests passen sich der Implementierungs-API an  
Führt eventuell zu einer schlechteren  
Benutzbarkeit und Verständlichkeit der API



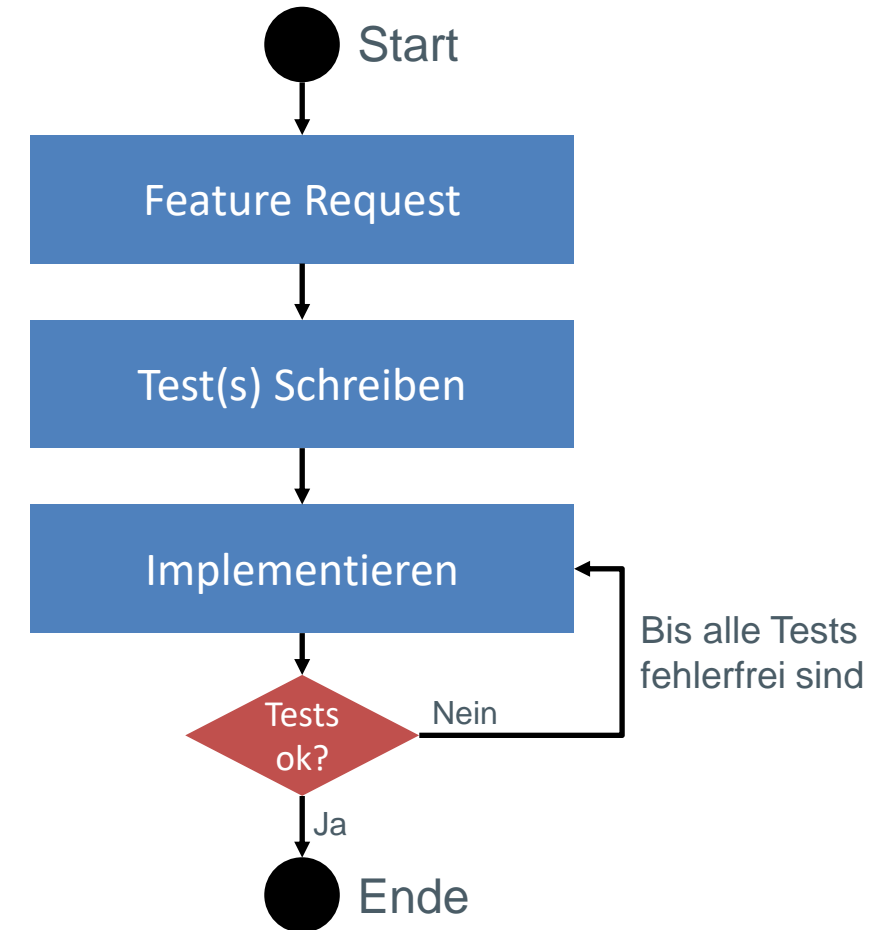
# Test-Driven Development

Tests **vor** der Implementierung schreiben  
→ Spezifikation des Programmverhaltens

Implementierung wird nur geändert, wenn...  
einer oder mehrere Tests fehlschlagen

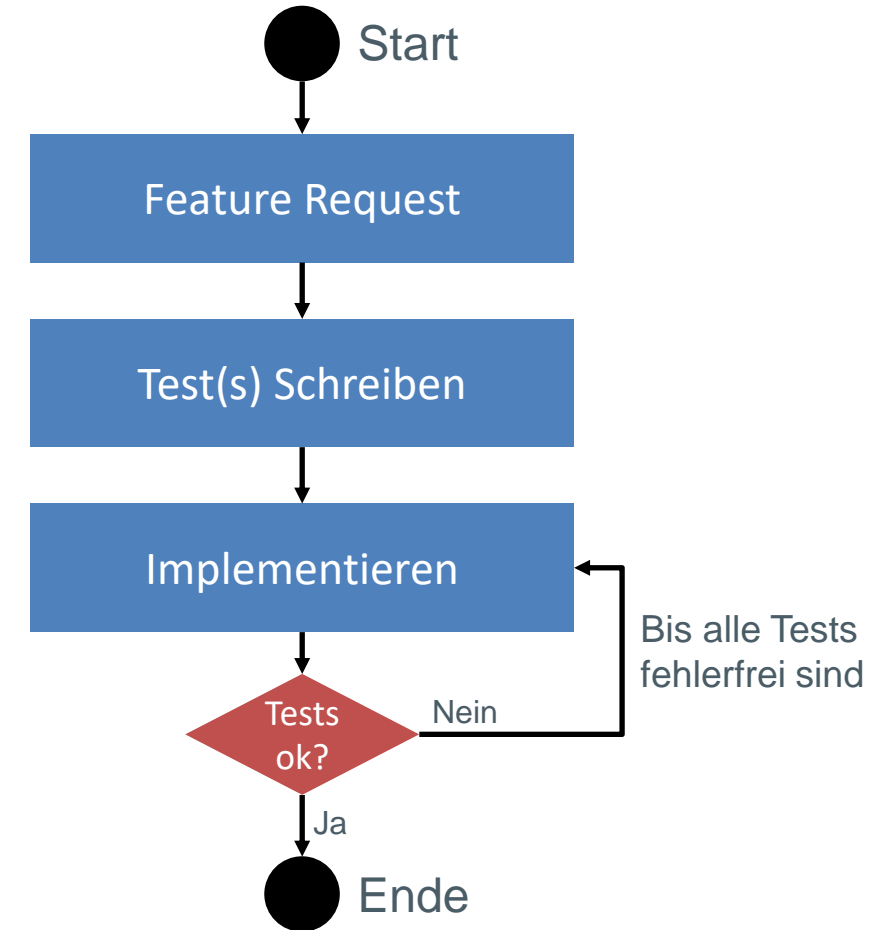
Wichtiges Element von *Extreme Programming* und  
anderen *Agile Development* Konzepten

Die Implementierungsphase versucht mit  
*minimalem* Code die Tests lauffähig zu bekommen  
→ Alle Features werden durch Tests abgebildet  
→ Es wird kein unnötiger Code hinzugefügt



## 😊 Vorteile von Test-Driven Development

1. Hohe Test-Abdeckung:  
*Jede Code-Zeile der Implementierung wurde durch einen Test notwendig (d.h. wird in Tests genutzt)*
2. Veränderte Entwicklungs-Perspektive  
*Tests: Wie sollte meine API aussehen? (= Benutzersicht)*  
*Implementierung: Wie kann ich diese API umsetzen?*
3. Tests dokumentieren die Implementierung
  - a) Welche Features sind vorhanden?
  - b) Tests sind "Beispiele" für andere Entwickler
4. Automatisiertes Evaluieren ist fester Bestandteil der Entwicklung





# Dokumentieren und Testen

## Dokumentieren

1. README
2. JavaDoc

## Testen

1. JUnit
2. Test-Coverage

# JUnit

Ein Framework zum Testen von Java-Programmen

Ähnliche Modultest-Frameworks existieren für viele Sprachen (meist xUnit genannt)

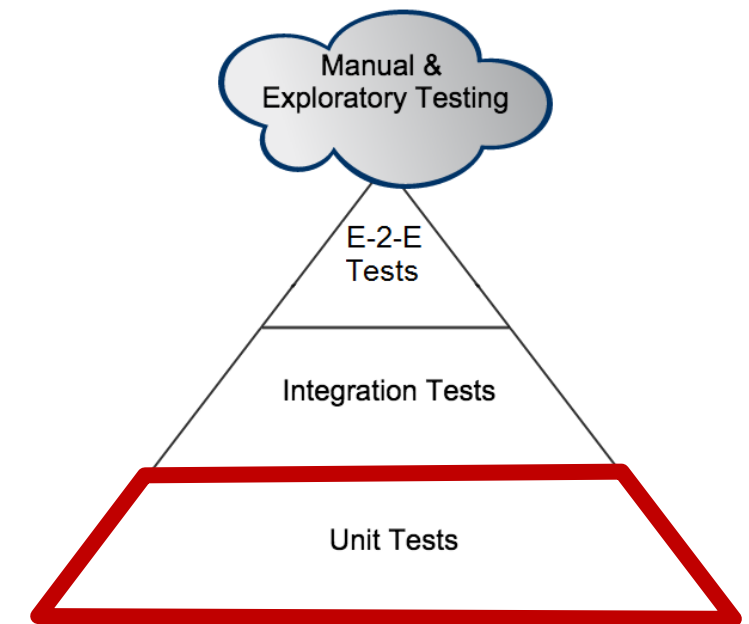
Erlaubt das Testen von Klassen, Methoden und Variablen

## Beinhaltet:

- Vorrichtungen für reproduzierbares Testen von Quellcode
- Methoden zum Vergleichen von Objekten und Werten
- Test-Runners zum Ausführen mehrerer Tests
- Übersichtliche Darstellung der Testergebnisse

Nicht Teil vom JDK: JUnit-Bibliothek muss eingebunden werden

→ IntelliJ schlägt automatisch einen Import vor



[Bildquelle: <https://www.endoflineblog.com/unit-acceptance-or-functional-demystifying-the-test-types-part2>]



## Aufbau eines JUnit Tests

```
package test.model;
```

} Tests in eigenem Packet

```
import evolisadventure.model.Pokemon;
```

```
import org.junit.jupiter.api.Test;
```

} Importiert zu testende Klasse

```
import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
import static org.junit.jupiter.api.Assertions.assertTrue;
```

```
class PokemonTest {
```

} Eine Test-Klasse pro Klasse

```
@Test
```

```
void isAlive_ShouldReturnTrue_WhenHPGreaterZero() {
```

```
    var p = new Pokemon("Bisasam", "", 123, 30);
```

```
    assertTrue(p.isAlive());
```

```
}
```

```
// [...]
```

```
}
```

**Ein Test:** Prüft ob ein neu erstelltes Pokemon mit 123 HP lebendig ist.

Verschiedene Naming-Conventions:

- <zu-testende-methode>()
- <methode>\_Should...\_When...()

## BeforeEach und AfterEach

```
class PokemonTest {
    Pokemon bisasam;

    @BeforeEach
    void setUp() {
        bisasam = new Pokemon("Bisasam", "", 100, 10);
    }

    @AfterEach
    void tearDown() {}

    @Test
    void isAlive_ShouldReturnTrue_WhenHPGreaterZero() {
        assertTrue(bisasam.isAlive());
    }
}
```

### setUp()-Methode:

Wird vor jedem Test aufgerufen.

D.h. jede Testmethode hat ein  
komplett neues Bisasam-Objekt.  
→ Testreihenfolge spielt keine Rolle

### tearDown()-Methode:

Wird nach jedem Test aufgerufen.

### setUp() und tearDown():

- DRY: Vermeidet Code-Duplikationen
- Reduzieren die Abhängigkeiten zwischen Tests  
→ Jeder Test arbeitet mit komplett neuen Objekten

## Die wichtigsten Assert Typen

### **assertTrue(value)**

### **assertFalse(value)**

Der Test schlägt fehl, wenn der Wert, bzw. das Statement, nicht zu **true** ausgewertet

→ Boolean Werte

### **assertNull(value)**

Der Test schlägt fehl, wenn Wert nicht null ist.

→ Objekte

### **assertEquals(expected, actual, [delta])**

Der Test schlägt fehl, wenn der Wert von **actual** nicht dem Wert von **expected** [ $\pm$  **delta**] entspricht

→ Zahlen und String

→ Bei Double/Float delta setzen!

### **assertSame(expected, actual)**

Der Test schlägt fehl, wenn **actual** und **expected** nicht das gleiche Objekt sind.

→ Objekte

### **assertArrayEquals(expected, actual)**

Der Test schlägt fehl, wenn **expected** und **actual** unterschiedlich sind (d.h. nicht die gleichen Werte besitzen).

→ Arrays

### **assertNotEquals/Null/...()**

Jeder Test hat noch eine negierte Assertion. Diese schlägt genau dann fehl, wenn der eigentliche Test nicht fehlschlagen würde.



# Dokumentieren und Testen

## Dokumentieren

1. README
2. JavaDoc

## Testen

1. JUnit
2. Test-Coverage

## Testabdeckung (en. Test Coverage)

Metrik zur Qualitätssicherung: Aussage wie viel vom Verhalten einer Anwendung durch Tests geprüft wurde  
Aussage meist in Prozent angegeben [0, 100]%

### In IntelliJ direkt berechenbar:

- Rechtsklick auf das Run-Icon
- “Run Main.main with Coverage”

### Je nach Branche eigene Vorschriften und Empfehlungen:

- IEEE 1008 „Software Unit Testing“
- ISO 26262 „Road vehicles – Functional safety“
- IEC 61508 „Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems“
- DO-178B „Software Considerations in Airborne Systems and Equipment Certification“

→ Legt eine vertretbare Grenze fest (mindestens X%, wobei X meist 100 ist)

## Function Coverage

Zählt wie viele Funktionen des Programmes von den Tests direkt oder indirekt aufgerufen werden

Indirekte Aufrufe über andere Funktionen werden mitgezählt (wichtig für private-Funktionen)

- 😊 Jede Methode wurde aufgerufen
- 😊 Sehr einfach 100% zu erreichen
- 😞 Unklar wie viel von einer Funktion getestet wurde  
→ Selten verwendet, da geringe Aussagekraft

### Beispiel:

```
static int foo(int x, int y) {  
    int z = 0;  
    if ((x > 0) && (y > 0)) {  
        z = x;  
    }  
    return z;  
}
```

### Beispiel-Aufrufe für 100% White-Box Tests:

1. `foo(0, 0)`

*Funktion zählt als aufgerufen. Aber der Test nutzt nicht den true-Block der if-Bedingung, d.h. die Zeile `z=x` wird nicht im Test verwendet*

## Line Coverage (Statement Coverage)

Zählt wie viele Code-Zeilen, bzw. Statements der Anwendung die Tests durchlaufen

100% Line Coverage → 100% Function Coverage

- 😊 Einfach im Editor visualisierbar
- 😊 Jede Verzweigung mit Anweisung wird mindestens einmal durchlaufen
- 😞 Nicht jedes Programmverhalten wird getestet  
→ Für nicht sicherheitskritische Anwendungen ausreichend (verwendete Metrik im Programmierprojekt)

### Beispiel:

```
static int foo(int x, int y) {  
    int z = 0;  
    if ((x > 0) && (y > 0)) {  
        z = x;  
    }  
    return z;  
}
```

### Beispiel-Aufrufe für 100% White-Box Tests:

1. `foo(1, 1)`

*Geht auch in den true-Block der if-Bedingung, damit die Zeile z=x abgedeckt wird*

## Branch Coverage

Zählt wie viele Verzweigungen der Anwendung von den Tests abgedeckt werden. Auch nicht explizit geschriebene else-Fälle zählen mit!

100% Branch Coverage → 100% Line Coverage

- 😊 Alle Fälle wurden getestet
- 😞 Meist nur durch Whitebox-Tests möglich
- 😞 Es gibt weiterhin nicht getestete Sonderfälle

### Weitere häufige Coverage-Metriken:

- **Loop-Coverage:** Wurden alle Schleifen 0-mal, 1-mal und mehr als einmal durch die Tests durchlaufen?
- **Condition-Coverage:** Alle möglichen Kombinationen der Teilbedingungen (z.B.  $x > 0$ ) müssen getestet sein.
- **Path-Coverage:** Alle möglichen Pfade durch die Anwendung müssen getestet werden.

### Beispiel:

```
static int foo(int x, int y) {
    int z = 0;
    if ((x > 0) && (y > 0)) {
        z = x;
    } else {}
    return z;
}
```

### Beispiel-Aufrufe für 100% White-Box Tests:

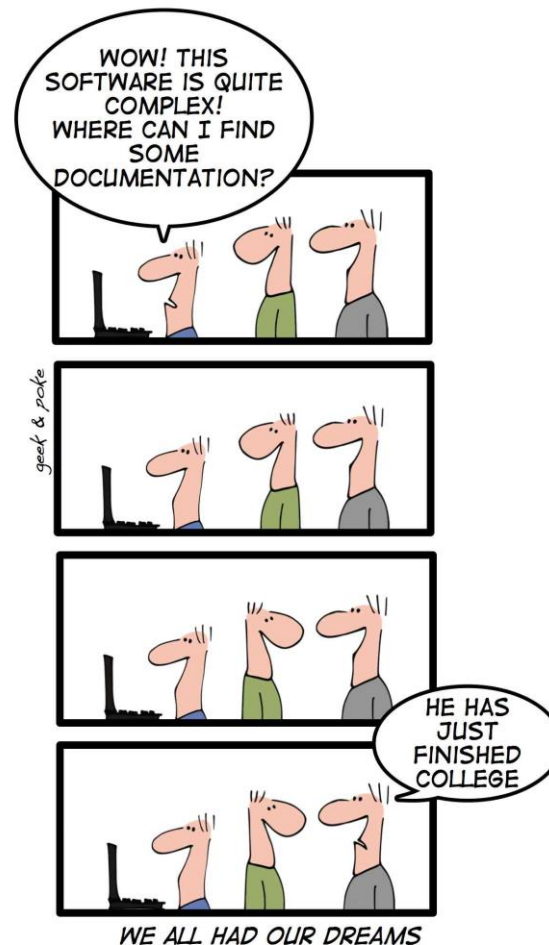
1. `foo(1, 1)`
2. `foo(1, 0)`

Zwei Aufrufe: der Erste für den *true-Block* der *if-Bedingung*, der Zweite für den leeren *false-Block*

(Der Branch-Coverage Bericht muss in den IntelliJ Run-Einstellungen aktiviert werden.)



# Fragen?



[Bildquelle: <http://geekandpoke.typepad.com/.a/6a00d8341d3df553ef0120a7190580970b-pi>]