

Richtig gute Tests schreiben

Best Practices für bessere (JUnit-) Tests

Java Forum Stuttgart 2023

Roland Krüger und Falk Sippach





Roland Krüger
Freiberufler

 mail@rolandkrueger.info

 @roland_krueger



Falk Sippach



 fs@embarc.de

 @sipp sack

Agenda



Motivation



Testdaten-
Management



Assertions



Struktur



Fazit





Gute Tests



~~Schlechte Tests~~

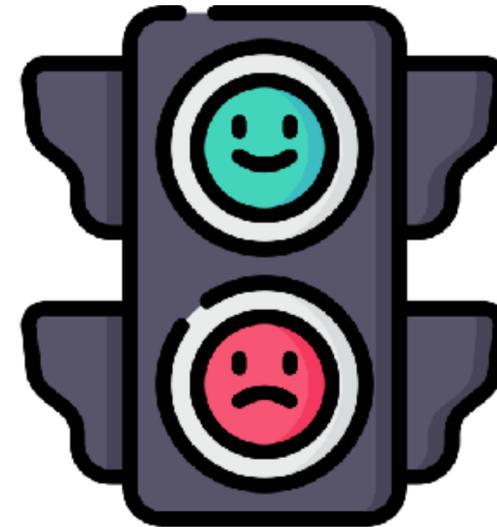
Was wollen wir?

Wir wollen **richtig**
gute Tests schreiben.

Gute Tests = **Schöne** Tests?



Hohe **Test-**
abdeckung?

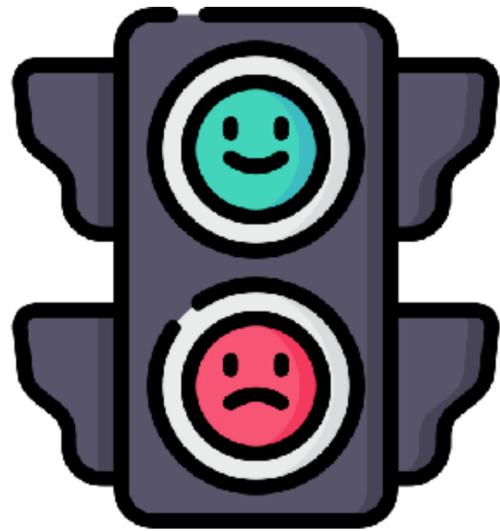


Aber: “Sind doch
nur Tests”

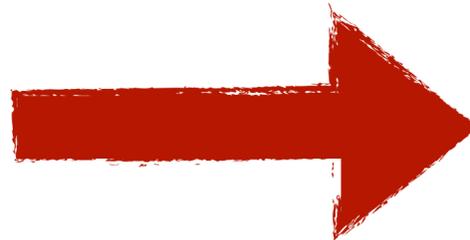
“Ich konzentriere mich
lieber auf den **Produktivcode.**”



Schöne Tests = **Wartbare** Tests?



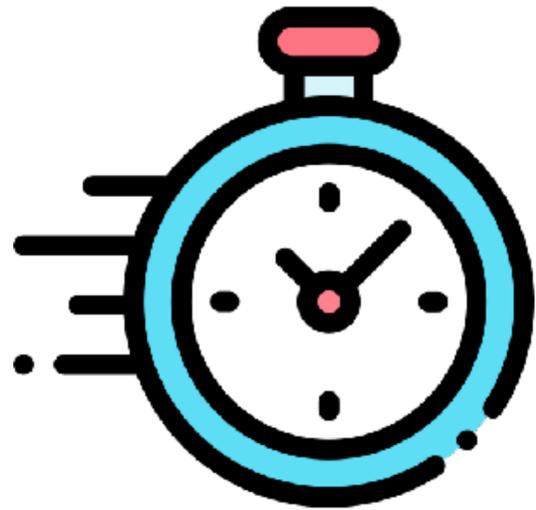
“Sind doch
nur Tests”



Todesspirale von
verrottendem Testcode



Anforderungen an gute Tests (1)



Schnell
schreiben
und ausführen



Gute
Lesbarkeit



Übersichtlich,
nachvollziehbar

Tests dienen implizit
als **Dokumentation**



Anforderungen an gute Tests (2)



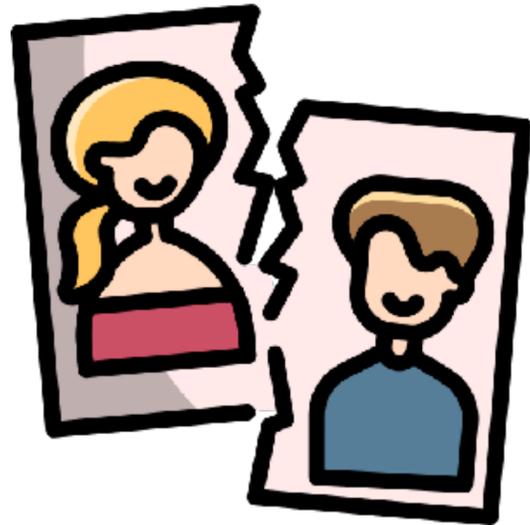
Stabil
gegenüber
Refactorings



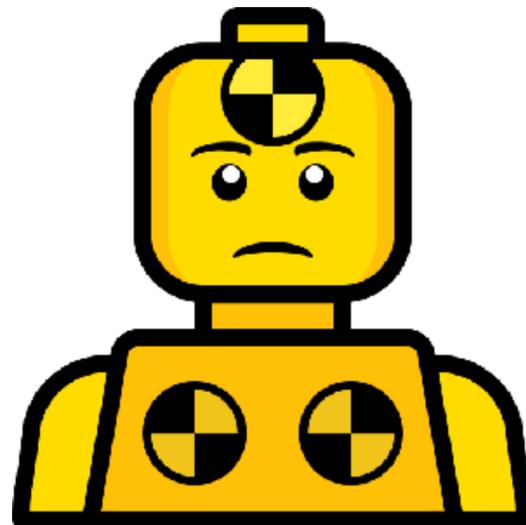
Hohe
Wartbarkeit



Anforderungen an gute Tests (3)



Entkopplung



Mocks zurren
Implementierungs-
details fest



Ziel: wenig Wissen
über interne
Datenstrukturen



von Produktivcode



Wie erreichen wir das?

- Software Craftsmanship **Prinzipien** und **Best Practices** auch **für Tests!**



SOLID



DRY



KISS



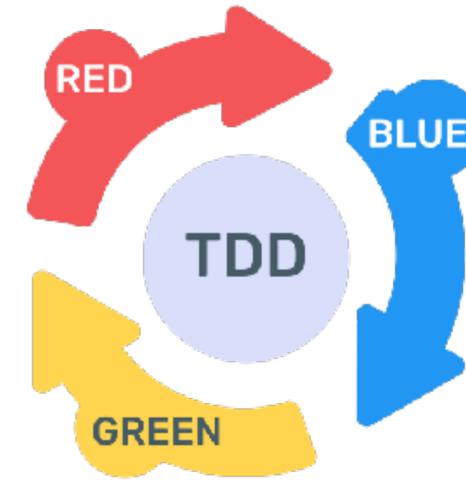
Wiederverwendbarkeit



Principle of Least Surprise



Tests schreiben
ist **unbeliebt**



TDD ist toll,
aber ...



... und eigentlich haben wir
doch **auch keine Zeit.**



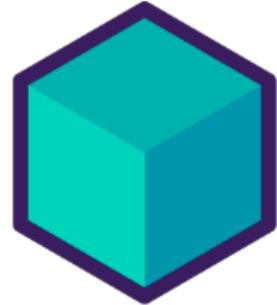
Verrottenden Testcode
zu **pflegen**, ist noch unbeliebter.



Es sollte **leicht** fallen,
Tests zu **schreiben**



JUnit



AssertJ
Fluent assertions for java



Cucumber



ArchUnit

Tools allein reichen nicht



Eigene, projektspezifische **Test-API**

Ziel: frustfreie Test-API

- Intuitiv benutzbar
- keine kognitive Überfrachtung
- Geringer Einarbeitungsaufwand
- Wird gerne vom ganzen Team genutzt
- Erleichtert das Refactoring von Produktiv- und Test-Code



Eine API, die **frustriert**, wird
vermieden.

Dadurch wird das Schreiben
von **Tests vernachlässigt**.

Agenda



Motivation



**Testdaten-
Management**



Assertions



Struktur



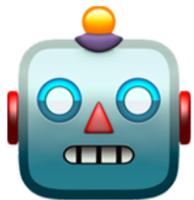
Fazit



Testaufbau



Arrange



Act



Assert

Ich brauch mal schnell ein Testobjekt...

```
@Test
void shouldValidateDateOfBirth() {
    User testUser = new User("john.doe");
    LocalDate today = LocalDate.now();
    LocalDate y2k = LocalDate.of(2000, 1, 1);

    // Rest vom Test ...
```

```
@Test
void shouldConfigureProduct() {
    User anon = new User("anonymous", Role.ANONYMOUS);
    User lockedUser = new User("dieter.develop", Role.CUSTOMER, true);

    Product product1 = new Product("ACME 2000", new PartNumber("prd", 10));
    Product product2 = new Product("ACME 3000", new PartNumber("srv", 20));

    // Rest vom Test ...
```

```
@Test
void shouldProcessShipment() {
    User user = new User("johnnyDoe", Role.CUSTOMER, false);
    Address shippingAddress = new Address("Musterstraße 10", "12345", "Musterstadt");
    Address billingAddress = new Address("Rechnungsweg 9", "67112", "Mutterstadt");

    // Rest vom Test ...
```



... und jetzt was Komplexes

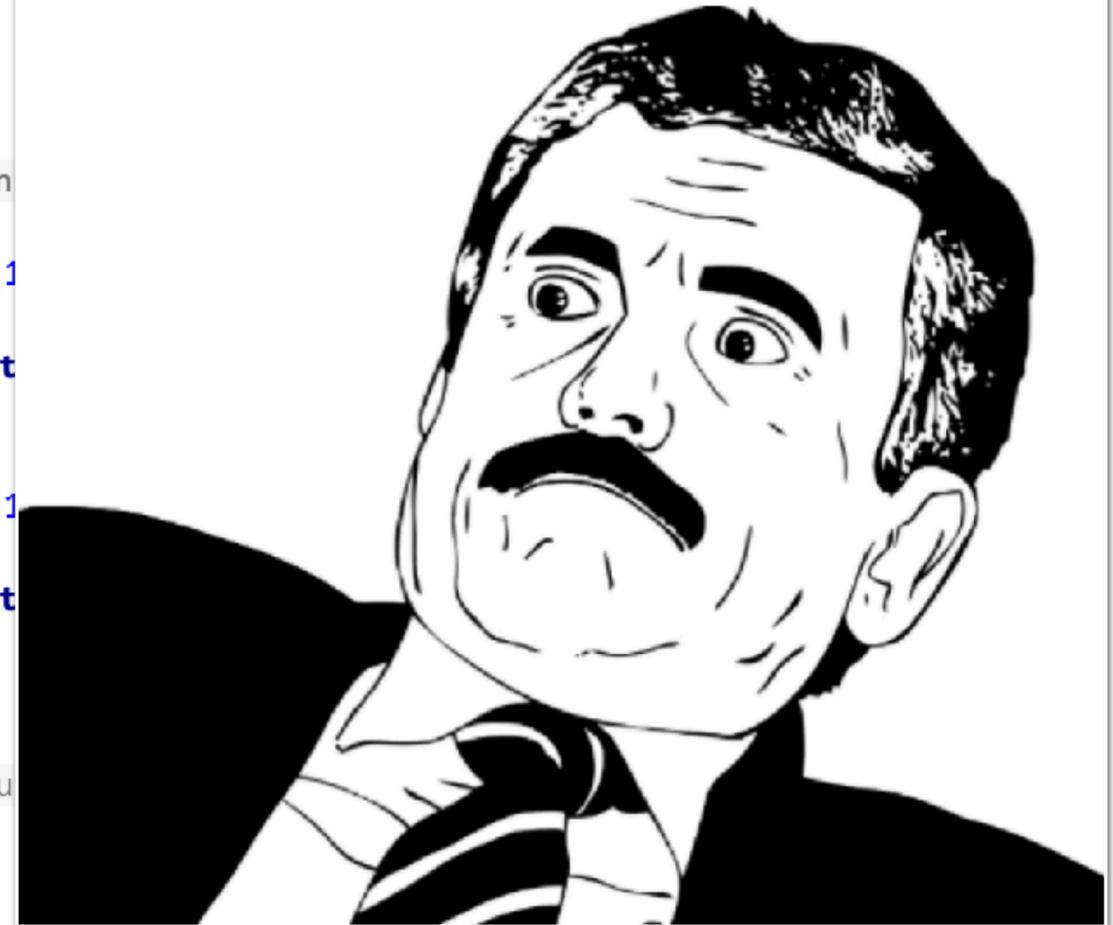
```
@Test
@DisplayName("Test mit umständlichem Testdaten-Setup")
void testWithComplexSetup() {
    User testUser = new User(username: "tanja_test", role: Role.CUSTOMER, isAccountLocked: false);

    CustomerDetailsDto customerDetails = new CustomerDetailsDto(customerReference: "customer_01", isPremium
        hasOrderBlock: false, countryOfOrigin: "de");
    ItemDto item1 = new ItemDto(partNumber: new PartNumber(category: "prd", number: 100), position: 1, amount: 1
        materialNumber: null, productName: null, orderCode: null, productionCountry: null,
        productInformation: "", details: "", productionTimeInDays: 1, isConfigurable: false, isConfiguredCompletely: t
        isSalesRestricted: false, isPriceOnRequest: false, configurationDto: null);

    ItemDto item2 = new ItemDto(partNumber: new PartNumber(category: "prd", number: 200), position: 2, amount: 1
        materialNumber: null, productName: null, orderCode: null, productionCountry: null,
        productInformation: "", details: "", productionTimeInDays: 1, isConfigurable: false, isConfiguredCompletely: t
        isSalesRestricted: false, isPriceOnRequest: false, configurationDto: null);

    OrderDto order = new OrderDto(placedBy: testUser, orderNumber: "123456789", customerDetails,
        items: List.of(e1: item1, e2: item2), paymentDetails: null, shoppingCartNumber: "1", quoteCreatedByCu
        orderDate: LocalDate.now(), messages: Collections.emptyList());

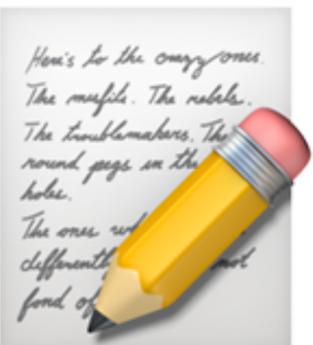
    boolean orderSuccess = service.executeOrder(order);
    assertTrue(condition: orderSuccess, message: "Order was not processed successfully.");
}
```



Lösung: Vordefinierte Testobjekte

Wir brauchen...

- ✓ Sauber ausdefinierte Testdaten
- ✓ Keine leeren Properties
- ✓ Zueinander passende Werte
- ✓ Verwaltet an zentraler Stelle
- ✓ Wiederverwendet von allen Tests



Aber nicht so

```
public final class TestConstants {  
  
    private TestConstants() {  
    }  
  
    public final static String USER_NAME_1 = "jane.doe";  
    public final static String USER_NAME_2 = "britta.musterfrau";  
    public final static String USER_NAME_3 = "dieter.develop";  
  
    public final static String PRODUCT_CODE_ACME2000 = "ACME 2000";  
    public final static String PRODUCT_CODE_ACME3000 = "ACME 3000";  
  
    public final static String CUSTOMER_ID_1 = "c_502949";  
    public final static String CUSTOMER_ID_2 = "c_841984";  
    public final static String CUSTOMER_ID_3 = "c_156435";  
    public final static String CUSTOMER_ID_4 = "c_897111";  
  
    public final static LocalDate TODAY = LocalDate.now();  
    public final static LocalDate TOMORROW = LocalDate.now().plusDays(daysToAdd: 1);  
    public final static LocalDate Y2K = LocalDate.of(year: 2000, month: 1, dayOfMonth: 1);  
}
```



Zentrale Fluent-API für Testobjekte

- Aber wie und wo finde ich die Testobjekte?
- Es gibt genau eine zentrale Klasse als Einstiegspunkt
- Liefert Hierarchie von Testobjekt-Factories
- Beispiel: TestObjects

TestObjects.

```
m users() UserTestObjects
m productData() ProductDataTestObjectGroup
m orderData() OrderDataTestObjectGroup
m customerDetails() CustomerDetailTestObjects
class
```

Press ↵ to insert, → to replace



Code Completion erledigt den Rest

```
User admin = TestObjects.
```

- m users().limitedUser() User
- m users().anonymousUser() User
- m users().administrator() User
- m users().lockedUser() User
- m users().anyCustomer() User

Press ↵ to insert, → to replace



```
ItemDto item = TestObjects  
    .productData()  
    .items().
```

- m **standardItem()** ItemDto
- m **unconfiguredItem()** ItemDto
- m **fullyConfiguredItem()** ItemDto

Press ↵ to insert, → to replace



```
CustomerDetailsDto customerDetails =
```

```
TestObjects.
```

- m customerDetails().standardCustomer() CustomerDetailsDto
- m customerDetails().foreignCustomer() CustomerDetailsDto
- m customerDetails().blockedCustomer() CustomerDetailsDto
- m customerDetails().premiumCustomer() CustomerDetailsDto

Press ↵ to insert, → to replace



Dependency Injection

SOLID

Single Responsibility

Open Closed

Liskov Substitution

Interface Segregation

Dependency Inversion



Dependency Injection in Tests

```
@Test
void shouldDoThings() {
    User anon = TestObjects
        .users()
        .anonymousUser();

    CustomerDetailsDto customer = TestObjects
        .customerDetails()
        .standardCustomer();

    OrderDto order = TestObjects
        .orderData()
        .orders()
        .standardOrder();
}
```

```
@Test
void shouldDoThings(User anon,
    CustomerDetailsDto customer,
    OrderDto order) {
}
```



Dependency Injection in Tests

```
@Test  
void shouldDoThings(@TestUser(ANONYMOUS_USER) User anon,  
                   @TestUser(ADMINISTRATOR) User admin,  
                   @TestUser(LIMITED_USER) User limited) {
```



Dependency Injection womit?

JUnit 5

- Kein Inversion-of-Control-Framework nötig
- JUnit hat alles an Bord, was ihr braucht



JUnit Extensions

I `org.junit.jupiter.api.extension.ParameterResolver`

`AfterAllCallback`

`BeforeAllCallback`

`BeforeTestExecutionCallback`

`InvocationInterceptor`

`TestInstanceFactory`

`TestInstancePostProcessor`

...

@ExtendWith

```
public class TestObjectInjectionExtension implements ParameterResolver
```

```
    @ExtendWith(TestObjectInjectionExtension.class)
```

```
    public class TestWithDependencyInjection
```

```
        @ExtendWith(TestObjectInjectionExtension.class)
```

```
        @Retention(RetentionPolicy.RUNTIME)
```

```
        @Target(ElementType.TYPE)
```

```
        public @interface UsesTestObjects {
```

```
        }
```

```
        @UsesTestObjects
```

```
        public class TestWithDependencyInjection {
```

Wie wird jetzt der ParameterResolver implementiert?



Demo

Das Builder Pattern

Was ist jetzt mit komplexeren Objekten?

... Und wie kriegen wir so etwas in den Griff?

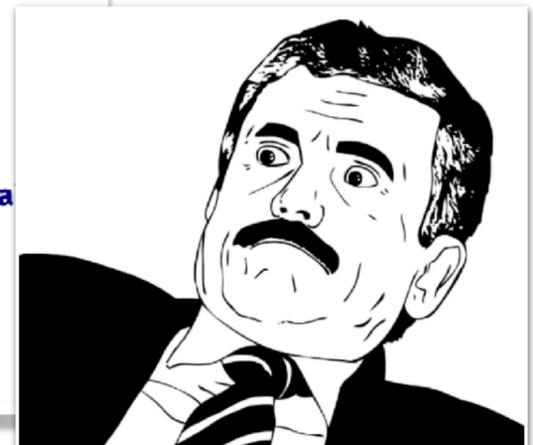
```
@Test
@DisplayName("Test mit umständlichem Testdaten-Setup")
void testWithComplexSetup() {
    User testUser = new User(username: "tanja_test", role: Role.CUSTOMER, isAccountLocked: false);

    CustomerDetailsDto customerDetails = new CustomerDetailsDto(customerReference: "customer_01", isPremium: false,
        hasOrderBlock: false, countryOfOrigin: "de");
    ItemDto item1 = new ItemDto(partNumber: new PartNumber(category: "prd", number: 100), position: 1, amount: 1, unit: "PC",
        materialNumber: null, productName: null, orderCode: null, productionCountry: null,
        productInformation: "", details: "", productionTimeInDays: 1, isConfigurable: false, isConfiguredCompletely: true,
        isSalesRestricted: false, isPriceOnRequest: false, configurationDto: null);

    ItemDto item2 = new ItemDto(partNumber: new PartNumber(category: "prd", number: 200), position: 2, amount: 1, unit: "PC",
        materialNumber: null, productName: null, orderCode: null, productionCountry: null,
        productInformation: "", details: "", productionTimeInDays: 1, isConfigurable: false, isConfiguredCompletely: true,
        isSalesRestricted: false, isPriceOnRequest: false, configurationDto: null);

    OrderDto order = new OrderDto(placedBy: testUser, orderNumber: "123456789", customerDetails,
        items: List.of(e1: item1, e2: item2), paymentDetails: null, shoppingCartNumber: "1", quoteCreatedByCustomer: false,
        orderDate: LocalDate.now(), messages: Collections.emptyList());

    boolean orderSuccess = service.executeOrder(order);
    assertTrue(condition: orderSuccess, message: "Order was not processed successfully.");
}
```



Das Builder Pattern

Kennt jeder, oder?

Das Builder Pattern

- Fragen wir mal ChatGPT:



Beschreibe das Builder Design Pattern kurz und knapp.



Das Builder Design Pattern ist ein Creational Pattern, das eine abstrakte Klasse oder ein Interface definiert, um ein komplexes Objekt schrittweise aufzubauen. Es ermöglicht die Trennung des Konstruktionsprozesses von der Repräsentation des Objekts.



Komplexe Domänen-Objekte



- Lange Konstruktoren
- Viele Properties
- Viele Setter
- Komplexe Unterdatenstrukturen
- Komplexe Produktiv-Builder



- Erzeugung über Konstruktoren und Setter-Methoden zu aufwendig
- müllt uns den Testcode zu



Produktiv-Builder

VS.



Testobjekt-Builder

Builder-API für Testzwecke

- Testobjekt-Builder können
 - sinnvolle Defaults vorgeben
 - Komplexität der Objekterzeugung reduzieren
 - Genau auf die Belange von Tests abgestimmt werden
 - Vorkonfigurierte Testdaten-Objekte wiederverwenden



Verwendung

- Über die Testobjekt-Factory

```
@Test
void shouldDoThings() {
    ItemDtoBuilder itemBuilder = TestObjects.productData()
        .items()
        .standardItemBuilder();
}
```

- Über Dependency Injection

```
@Test
void shouldDoThings(ItemDtoBuilder itemBuilder) {
```



Options-Objekte

Default-Werte

```
public class AddressOptionsObject {  
    private String street = "Musterstraße 1";  
    private String zipCode = "12345";  
    private String city = "Musterstadt";  
  
    public Address build() {  
        return new Address(street, zipCode, city);  
    }  
  
    public AddressOptionsObject withStreet(String street) {  
        this.street = street;  
        return this;  
    }  
  
    public AddressOptionsObject withZipCode(String zipCode) {  
        this.zipCode = zipCode;  
        return this;  
    }  
  
    public AddressOptionsObject withCity(String city) {  
        this.city = city;  
        return this;  
    }  
}
```

Verstecken komplexer
Konstruktoraufrufe

Überschreiben der
Default-Werte

Agenda



Motivation



Testdaten-
Management



Assertions



Struktur



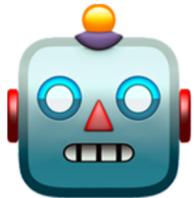
Fazit



Testaufbau



Arrange



Act



Assert

Assertions? Kennen wir alle.

`org.junit.jupiter.api.Assertions`

`assertEquals()`

`assertSame()`

`assertNotNull()`

`assertTrue()`

`assertFalse()`

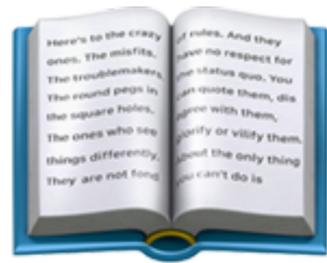
`assertThrows()`

`assertNotEquals()`

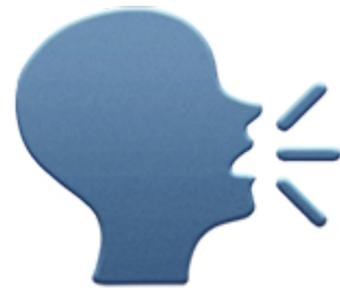
`assertAll()`

Aber reicht das?

Lesbarkeit



Aussagekraft



Refactoring-
Robustheit



Alles gut für einfache Fälle

`x/0`

`assertThrows()`

`“Hello, ” + name = “Hello, World”`

`assertEquals()`



`.canFly() = true`

`assertTrue()`

`[, , , , ].find() = null`

`assertNull()`

Und bei nicht einfachen Fällen?

```
List<String> nameList = List.of(...);  
  
assertEquals(0, nameList.size());  
assertTrue(nameList.isEmpty());  
assertTrue(nameList.contains("Dude"));  
  
LocalDate someDate = ...  
assertEquals(LocalDate.now(), someDate);
```



Assertion Bibliotheken



Hamcrest

Matchers that can be combined to create flexible expressions of intent

STRIKT

AssertJ - fluent assertions java library

Truth - Fluent assertions for Java and Android

Jetzt nochmal

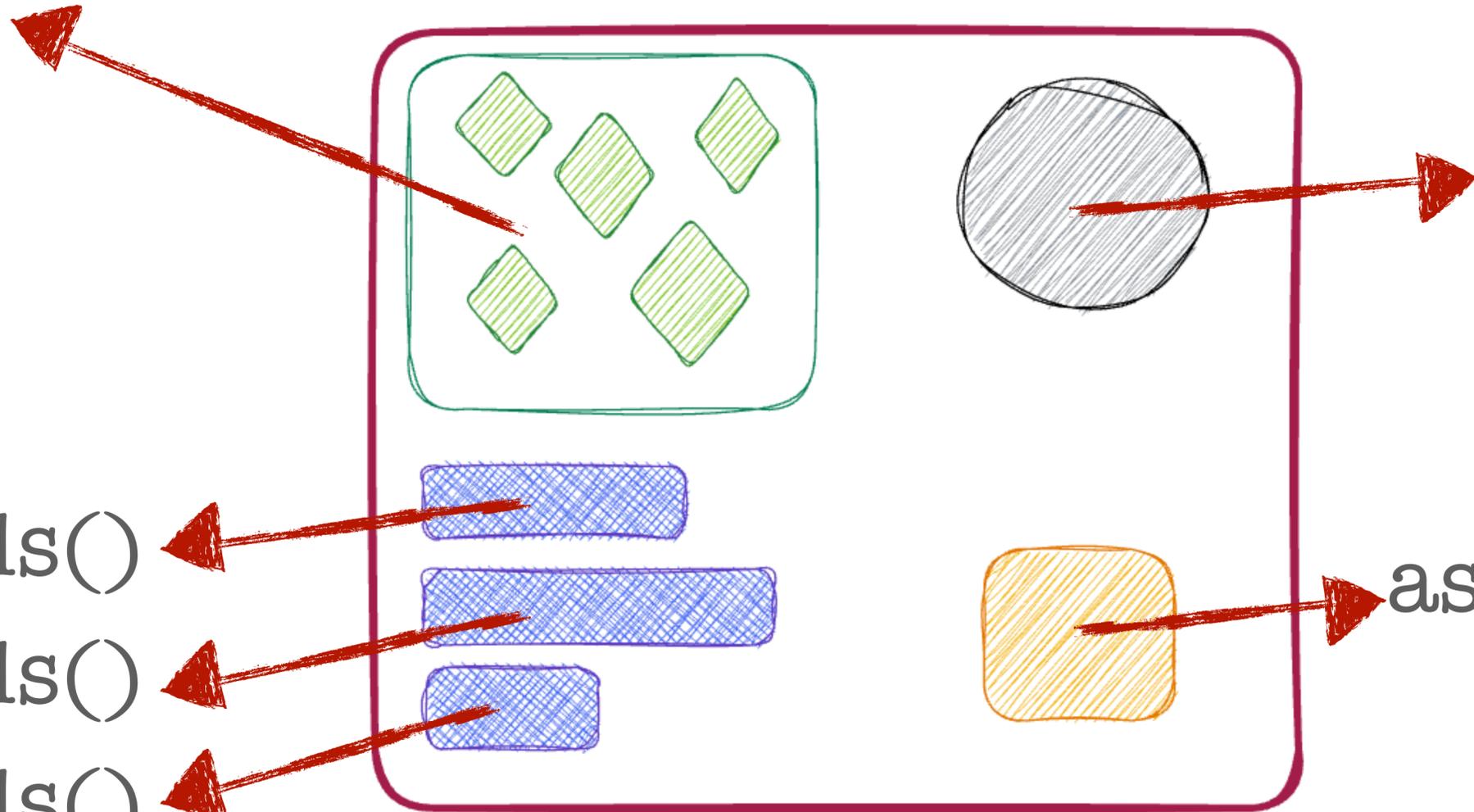
```
List<String> nameList = List.of(...);  
assertThat(nameList).isEmpty();  
assertThat(nameList).contains("dude");  
assertThat(nameList).doesNotContainNull();  
assertThat(nameList).hasSizeBetween(1, 10);
```

```
LocalDate someDate = ...  
assertThat(someDate).isToday();  
assertThat(someDate).isBefore(LocalDate.now());  
assertThat(someDate).hasYear(2023);
```



Was ist mit eigenen Domänenobjekten?

`assertIterableEquals()`



`assertTrue()`

`assertEquals()`

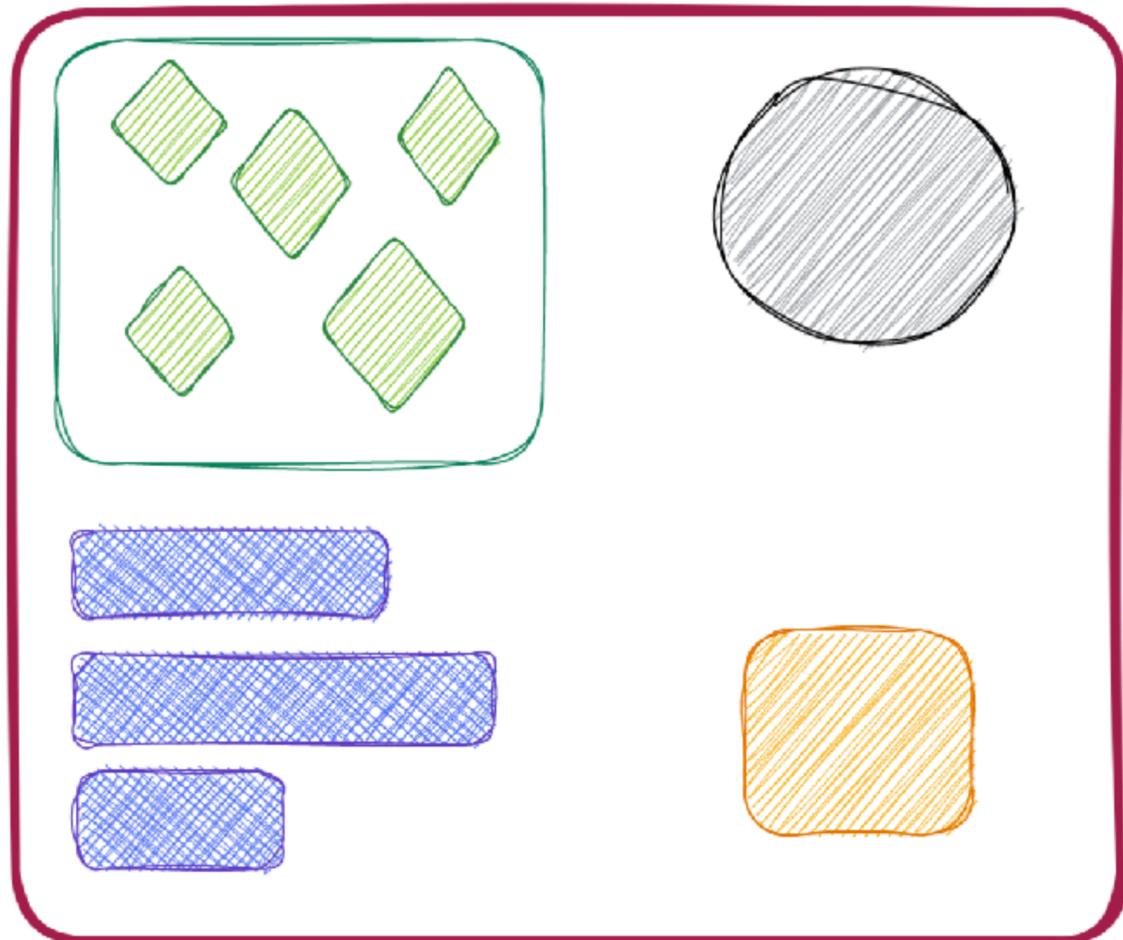
`assertEquals()`

`assertEquals()`

`assertEquals()`



Information Hiding

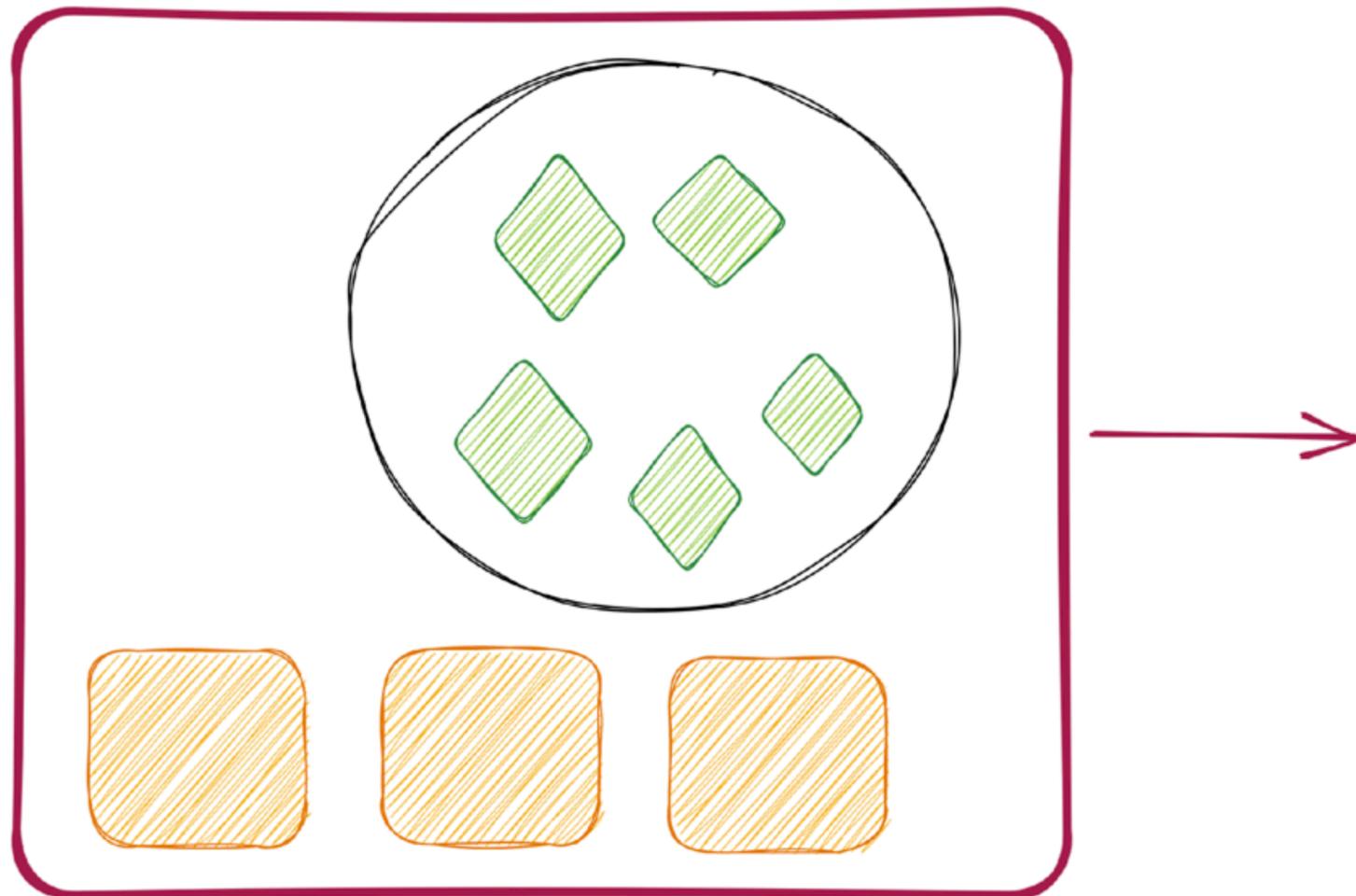


Eigener Matcher kapselt Implementationsdetails

```
assertThat(foo)  
  .isAProperFooBar()
```



Nach Refactoring der Domänenklasse



```
assertThat(foo)  
    .isAProperFooBar()
```



Implementierung eines Matchers

 Demo

Agenda



Motivation



Testdaten-
Management



Assertions



Struktur



Fazit



Sprechen wir über Übersichtlichkeit

- Namen von Testmethoden
- Strukturierung der Tests

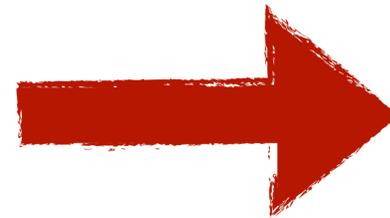


Testnamen mit @DisplayName

- Lesbarkeit von Test-Namen

✓ FlatTest (demo.nestedTests)

- ✓ testPremiumCustomerOrderWithRestrictedProducts()
- ✓ testAnonymousUserOrderFailure()
- ✓ testPremiumCustomerOrderSuccess()
- ✓ testStandardCustomerOrderSuccess()
- ✓ testStandardCustomerOrderFailure()
- ✓ testStandardCustomerOrderWithOrderBlock()
- ✓ testPremiumCustomerOrderWithOrderBlock()
- ✓ testStandardCustomerOrderWithRestrictedProducts()
- ✓ testAnonymousUserOrderSuccess()
- ✓ testAnonymousUserOrderWithRestrictedProducts()



✓ Bestellungen mit verschiedenen Kundentypen (demo.nestedTests)

- ✓ Premium-Kunde bestellt eingeschränkte Produkte
- ✓ Bestellung nicht erfolgreich mit anonymem Kunde
- ✓ Bestellung erfolgreich mit registriertem Kunde
- ✓ Bestellung erfolgreich mit Standard-Kunde
- ✓ Bestellung nicht erfolgreich mit Standard-Kunde
- ✓ Standard-Kunde bestellt mit Bestellsperre
- ✓ Premium-Kunde bestellt mit Bestellsperre
- ✓ Standard-Kunde bestellt eingeschränkte Produkte
- ✓ Bestellung erfolgreich mit anonymem Kunde
- ✓ Anonymer Kunde bestellt eingeschränkte Produkte

Testnamen mit @DisplayName

- @DisplayName-Annotation an Klassen und Methoden

```
@DisplayName("Bestellungen mit verschiedenen Kundentypen")
public class DisplayNameTest {

    @Test
    @DisplayName("Bestellung erfolgreich mit anonymem Kunde")
    void testAnonymousUserOrderSuccess() {
        // test code
    }
}
```

- Alternativ Display Name Generator:
`@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)`

Verschachtelte Tests mit @Nested

- Strukturierung unserer Tests bisher nur auf 2 Ebenen:

- Testklassen
- Testmethoden

```
✓ FlatTest (demo.nestedTests)  
  ✓ testPremiumCustomerOrderWithRestrictedProducts()  
  ✓ testAnonymousUserOrderFailure()  
  ✓ testPremiumCustomerOrderSuccess()  
  ✓ testStandardCustomerOrderSuccess()  
  ✓ testStandardCustomerOrderFailure()  
  ✓ testStandardCustomerOrderWithOrderBlock()  
  ✓ testPremiumCustomerOrderWithOrderBlock()  
  ✓ testStandardCustomerOrderWithRestrictedProducts()  
  ✓ testAnonymousUserOrderSuccess()  
  ✓ testAnonymousUserOrderWithRestrictedProducts()
```

Verschachtelte Tests mit @Nested

- @Nested gibt uns eine weitere Ebene zur Test-Strukturierung

- ✓ Test des Bestellprozesses (demo.nestedTests)
 - ✓ mit anonymem Benutzer
 - ✓ Bestellung schlägt fehl
 - ✓ erfolgreiche Bestellung
 - ✓ Bestellung mit verkaufsbeschränkten Produkten
 - ✓ mit Premiumkunde
 - ✓ Bestellung mit verkaufsbeschränkten Produkten
 - ✓ erfolgreiche Bestellung
 - ✓ Bestellung mit Bestellsperre
 - ✓ mit Standard-Kunde
 - ✓ erfolgreiche Bestellung
 - ✓ Bestellung schlägt fehl
 - ✓ Bestellung mit Bestellsperre
 - ✓ Bestellung mit verkaufsbeschränkten Produkten

Verschachtelte Tests mit @Nested

- Mit @Nested annotierte innere Klassen

```
@DisplayName("Test des Bestellprozesses")
public class NestedTest {

    @Nested
    @DisplayName("mit Standard-Kunde")
    class StandardCustomer {
        @Test
        @DisplayName("erfolgreiche Bestellung")
        void testStandardCustomerOrderSuccess() {
            // test code ...
        }
    }
}
```

Agenda



Motivation



Testdaten-
Management



Assertions



Struktur



Fazit



JUnit Werkzeugkiste - Fazit

- **Extensions** für eigene Erweiterungen
 - ExtensionContext mit eigenem testübergreifendem Storage
- **@Nested** Tests
- **@DisplayName**
- **Klassifizierung** von Tests mit **@Tag**



Best Practices - Fazit

- Schreibt eine **effiziente Test-API**, zugeschnitten auf euer Projekt
- Pflegt einen Vorrat an **sauber ausgearbeiteten Testobjekten**
- Reduziert Setup-Code mit **Dependency Injection**
- Nutzt eine **Assertion-Bibliothek** und deren **Matcher**
- Schreibt **eigene Assertions/Matcher** für eure Domänenobjekte
- Macht Gebrauch von **Design Patterns**



Demo Code

- <https://github.com/rolandkrueger/unit-test-best-practices>



Spicken erlaubt!

architektur SPICKER
Übersichten für die konzeptionelle Seite der Softwareentwicklung

Der Architekturüberblick
Ein Architekturüberblick macht die zentralen Lösungsansätze Ihrer Softwarearchitektur in kompakter Form nachvollziehbar.

Herausforderungen

- Team- oder Projektmitgliedern (z.B. Entwicklern) fehlt ein Überblick über Lösungsansätze auf hoher Ebene, um fokussiert zu arbeiten
- Neue Teammitglieder, die mitentwickeln wollen, finden sich in der Architektur nicht zurecht
- Entscheider und andere Stakeholder haben Unsicherheiten oder geringes Vertrauen in die Lösung
- Teamfremde Kollegen sind an Lösungsansätzen interessiert, finden aber keine oder sehr detaillierte Informationen, die Ihnen einen schnellen Überblick schwierig machen.

Inhalte eines Architekturüberblicks
Arbeiten Sie kleinteilig! Fertigen Sie unabhängige „Zutaten“ an, die sie zu unterschiedlichen Formen rekombinieren, und bei Bedarf iterativ verfeinern.

Formen

- Je nach Zielgruppe und Kommunikationsweg sind für einen Architekturüberblick sehr unterschiedliche Formen denkbar.
- **Architekturwand:** Jedermann zugänglicher, großformatiger, modularer Aushang an einer Wand im Projektraum
- **Architekturflyer oder -poster:** Kleines Handout, z.B. DIN A4 beidseitig bedruckt, 2-3x gefaltet, oder größer produziert (z.B. DIN A1) als Plakat zur weiten Verbreitung
- **Architekturportal im Wiki:** Einstiegsseite(n) im Wiki, die Interessierte durch die Inhalte führen
- **Prägnantes Dokument:** Strukturierter Text, angereichert mit Illustrationen, Umfang maximal 20 Seiten
- **Foliensatz:** 10-15 Folien zur Unterstützung einer Präsentation der Architektur
- **Video:** Aufzeichnung eines Überblicks in Ton und Bild, evtl. kombiniert mit Foliensatz

Zutaten
Was gehört rein? Die Zutaten dieser Abbildung sind auf der nächsten Seite beschrieben. Keine Sorge, Sie brauchen i.d.R. nicht alle.

Problemstellung	Lösungsstrategie	Lösungsdetails
Mission Statement	Lösungsstrategie	Architektur-entscheidungen
Einflussfaktoren Architekturziele Randbedingungen Risiken	Architekturprinzipien	„Sichten“ Struktur Verhalten Verteilung
Kontextabgrenzung	Überblicksbild	Übergreifende Konzepte

Abbildung 1: Überblick über wichtige Zutaten

<http://architektur-spicker.de>



Unsere Architektur-Spicker beleuchten die konzeptionelle Seite der Softwareentwicklung.



Spicker #1:
„Der Architekturüberblick“

- Welche Zutaten gehören in einen Architekturüberblick?
- Welche Formen bewähren sich in welchen Situationen?
- Wie fertigen Sie einen Architekturüberblick an?

➔ embarc.de/architektur-spicker/

Flexible Architekturen (iSAQB® CPSA®-A FLEX)

3-Tages-Training | mit iSAQB®-Zertifikat | 24.-26.07. Darmstadt

10%-Rabatt für Teilnehmer:innen

CONFFS10

(solange der Vorrat reicht, gültig bis 19.07.2023)



Vielen Dank

Wir freuen uns auf Eure Fragen!



mail@rolandkrueger.info



[@roland_krueger](https://twitter.com/roland_krueger)



fs@embarc.de



[@sippsack](https://twitter.com/sippsack)