

Können willkürliche Tests und Zombies unseren Code besser machen?

Patrick Drechsler
🐦 @drechsler
Redheads Ltd.

MATHEMA Campus
📅 2019-05-10

https://pixabay.com/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=156055
https://pixabay.com/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=25637



PATRICK DRECHSLER

- Software Entwickler / Architekt
- Beruflich: C#
- Interessen:
 - Software Crafting
 - Domain-Driven-Design
 - Funktionale Programmierung

🐦 @drechsler

🐙 github.com/draptik

🌐 draptik.github.io

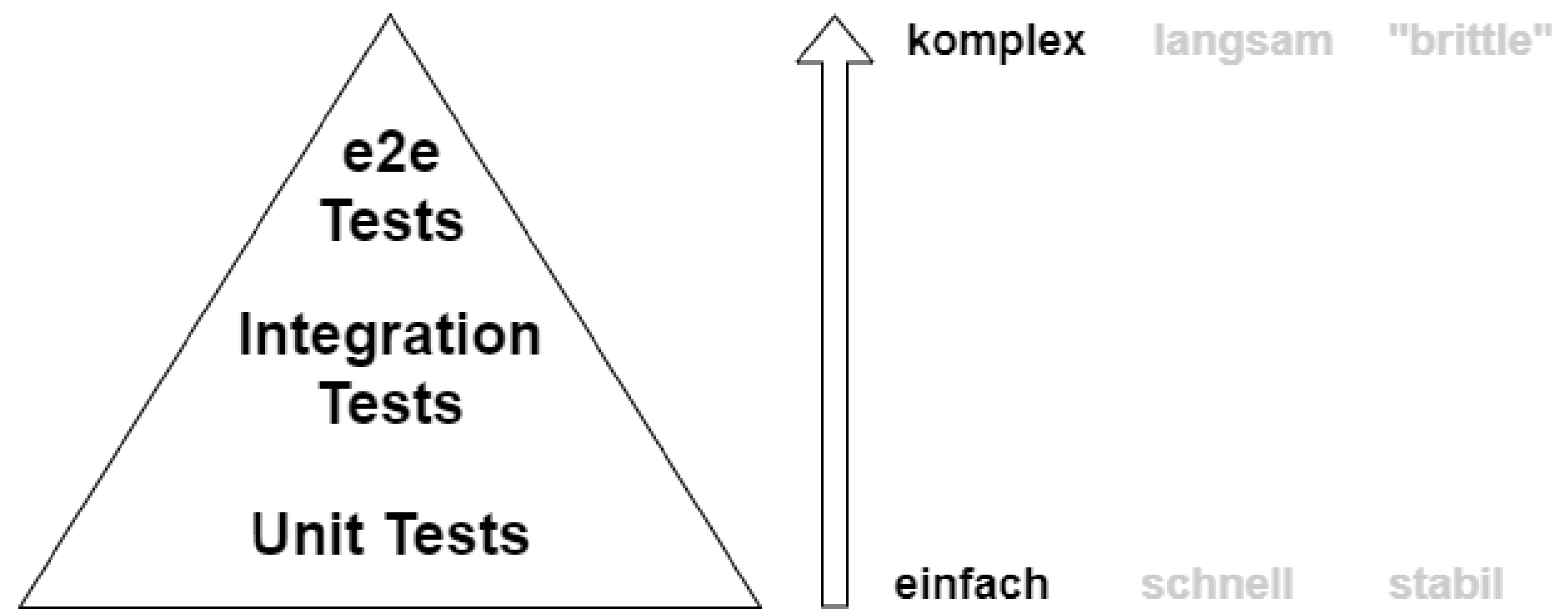
💼 redheads.de

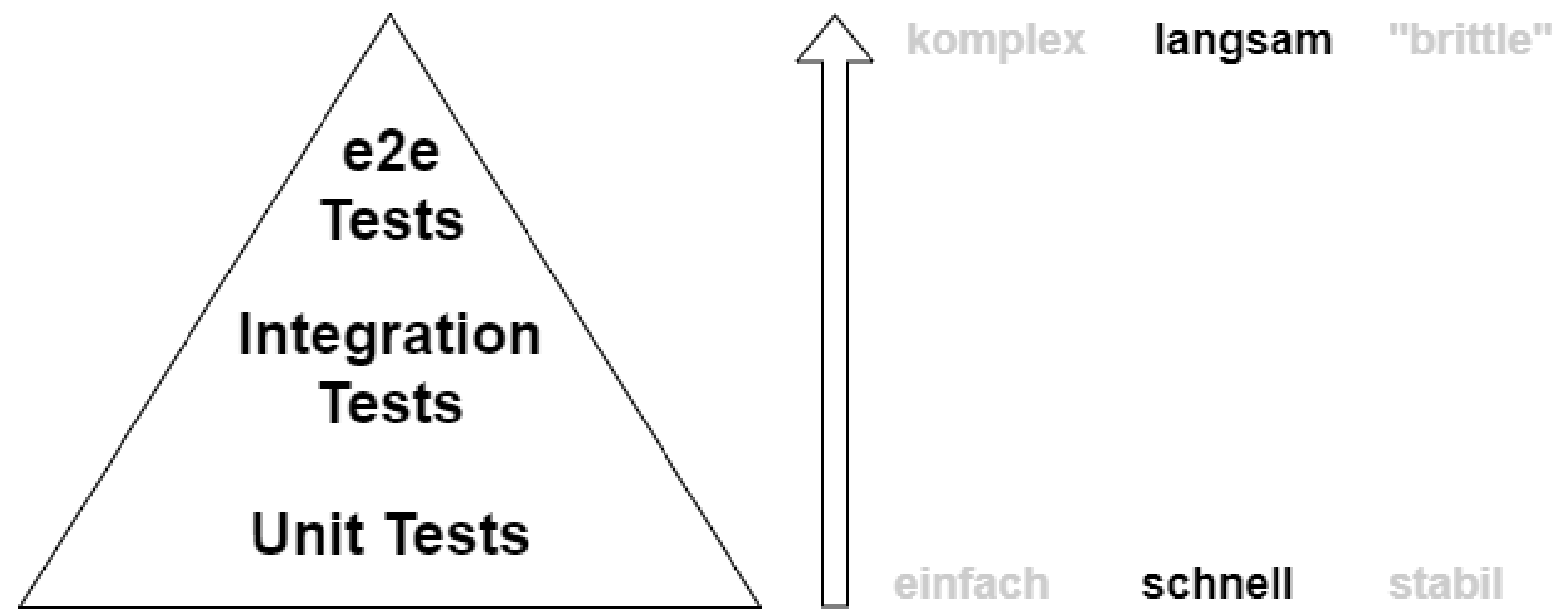


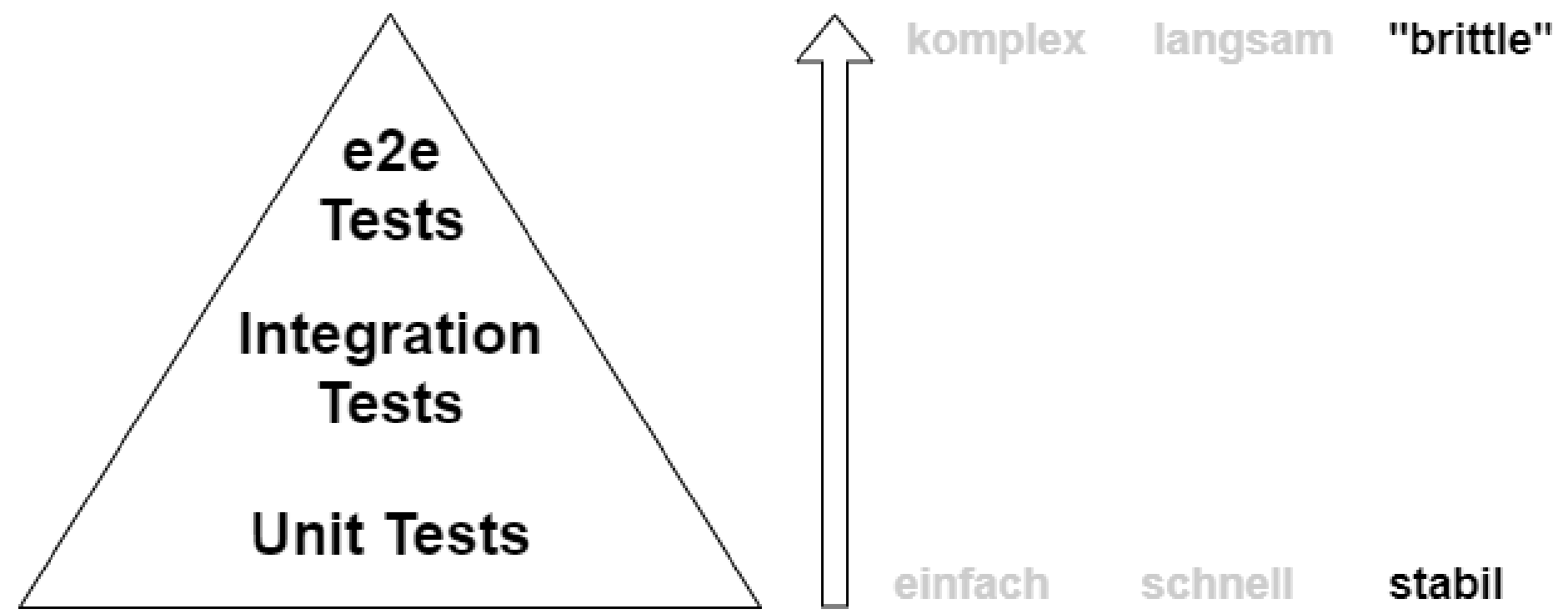
VORTEILE VON TESTS

- Dokumentation
- Modular
- Wartbar
- ...

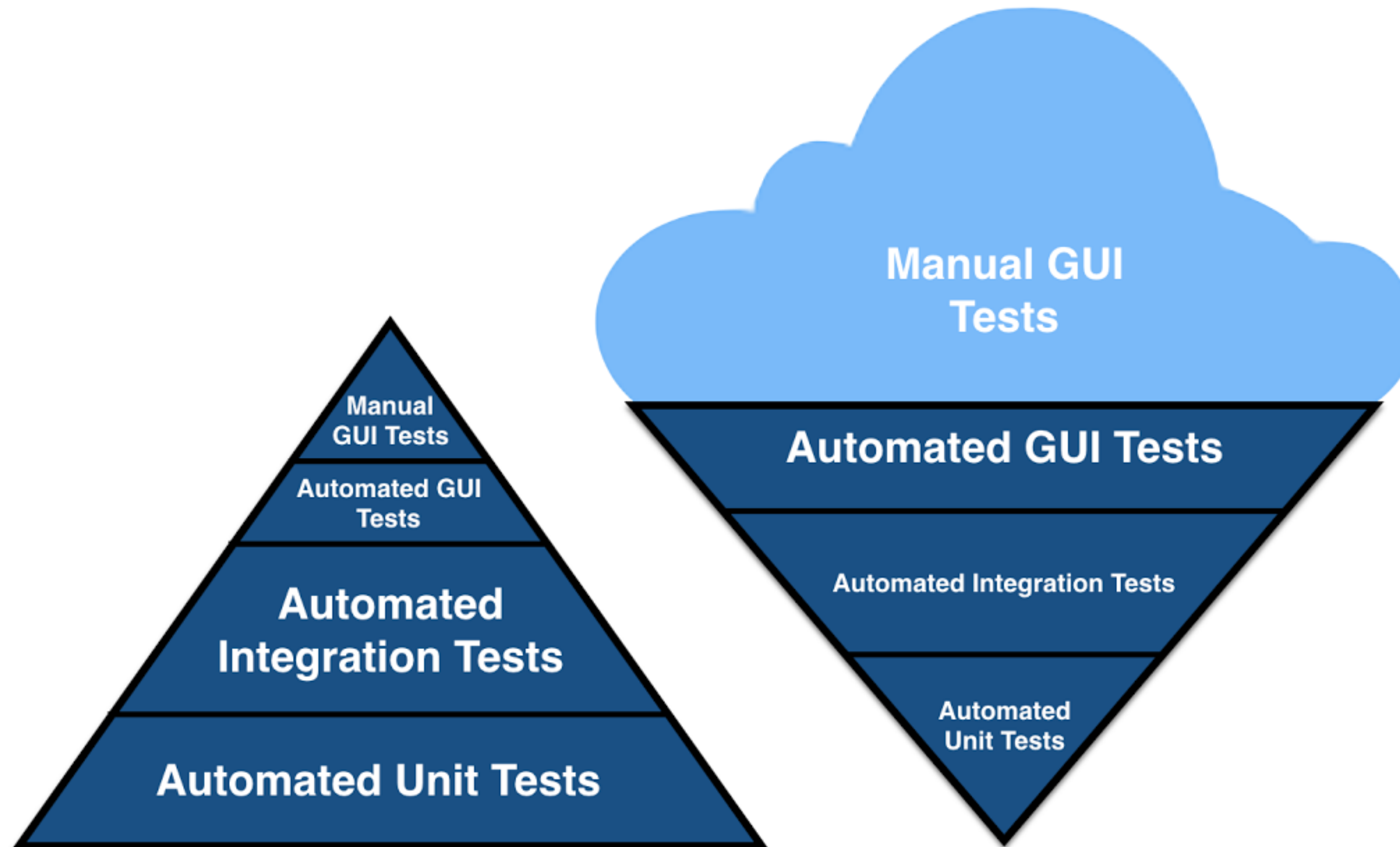
TEST PYRAMIDE



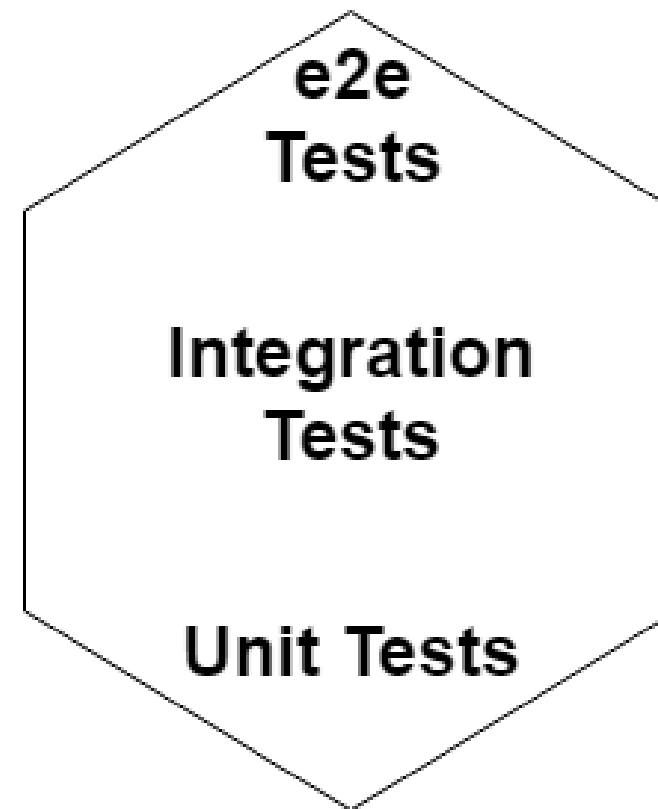




ANTIPATTERN: ICE CREAM CONE



HONEYCOMB



für Microservices

TEST DRIVEN DEVELOPMENT (TDD)



- Test schreiben
- Test sollte fehlschlagen
- minimale (!) Implementierung
- Test grün
- Refactoring (keine Änderung der Funktionalität)
- Repeat

FAZIT TDD

- Design / Architektur wird von Anfang an **modular**
 - vermeiden von zu großen Sprüngen ("Baby Steps")
 - vermeiden von zu vielen Abhängigkeiten
 - vermeiden (globaler) Zustände
- Entstehende Tests sind gleichzeitig:
 - Dokumentation
 - und Beispiel für Konsumenten der "API"

MUTATION TESTING

MUTATION TESTING

Warum?

- Weil Code Coverage nicht die ganze Wahrheit ist
- Qualität der Tests prüfen

GRUNDGEDANKE

- Änderung/Bugs in getesteten Code einbauen ("Mutanten")
- Wenn es keinen Einfluss auf das Test-Ergebnis hat, hat der Mutant überlebt 🤖 ☐ ➖
- Wenn der Mutant einen Test fehlschlagen lässt, ist der Mutant tot 😎

MUTATIONEN

Mutator	Stryker	Stryker.NET	Stryker4s
Arithmetic Operator	✓	✓	✗
Array Declaration	✓	✗	✗
Assignment Expression	✗	✓	n/a
Block Statement	✓	✗	✗
Boolean Literal	✓	✓	✓
Checked Statement	n/a	✓	n/a
Conditional Expression	✓	✓	✓
Equality Operator	✓	✓	✓
Logical Operator	✓	✓	✓
Method Expression	✗	✓	✓
String Literal	✓	✓	✓

DEMO

MUTATION TESTING FRAMEWORKS

- Java: PIT
- Stryker
 - C#: Stryker.NET
 - Javascript
 - Scala

PROPERTY BASED TESTING

Herkömmlicher Unit Test

```
public int Add(int a, int b) ⇒ a + b;
```

```
[Fact]  
public void Adding_1_and_1_returns_2() ⇒  
    Add(1, 1).Should().Be(2);
```

Nachteile:

- es wird nur ein Fall getestet
- unvollständig

Parametrisierter Test

```
[Theory]
[InlineData(1, 1, 2)]
[InlineData(-1, 1, 0)]
[InlineData(0, 0, 0)]
public void Adding_two_number_returns_correct_result(
    int a, int b, int expected)
{
    Add(a, b).Should().Be(expected);
}
```

Nachteil: man vergisst/uebersieht
Randbedingungen

Zufallsdaten verwenden

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

```
[Fact]
public void Adding_2_random_numbers_should_work()
{
    var a = Randomizer<int>.Create();
    var b = Randomizer<int>.Create();

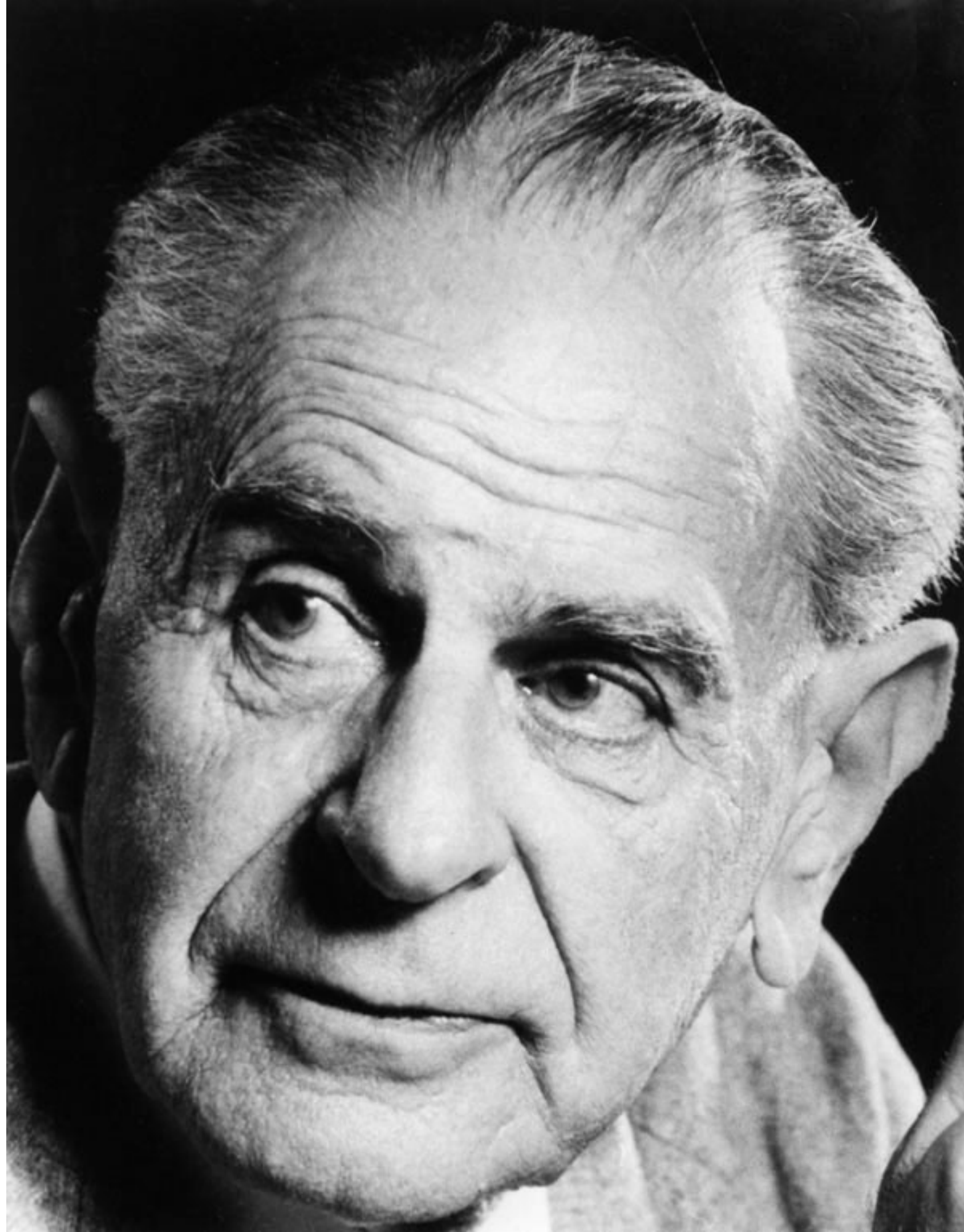
    // Dangerous!
    Add(a, b).Should().BeGreaterOrEqualTo(Int32.MinValue);
}
```

Nachteile:

- Man erwischt nicht alle Randbedingungen
- Was ist der Erwartungswert ("expected")?

PROPERTY BASED TESTING

- Property: "Eigenschaft" (nicht C# Property)
- arbeitet auch mit Zufallsdaten
- Sinnvolle Zufallsdaten
- Genügend Zufallsdaten
- Falsifizierung





WIKIPEDIA
Die freie Enzyklopädie

[Hauptseite](#)
[Themenportale](#)
[Zufälliger Artikel](#)

[Mitmachen](#)

[Artikel verbessern](#)
[Neuen Artikel anlegen](#)
[Autorenportal](#)
[Hilfe](#)

 Nicht angemeldet [Diskussionsseite](#) [Beiträge](#) [Benutzerkonto erstellen](#) [Anmelden](#)

[Artikel](#)

[Diskussion](#)

[Lesen](#)

[Bearbeiten](#)

[Quelltext bearbeiten](#)

[Versionsgeschichte](#)

Wikipedia durchsuchen




Falsifikation

Falsifikation, auch **Falsifizierung** (von lat. *falsificare* „als falsch erkennen“) oder **Widerlegung**, ist der Nachweis der Ungültigkeit einer [Aussage](#), [Methode](#), [These](#), [Hypothese](#) oder [Theorie](#). Aussagen oder experimentelle Ergebnisse, die Ungültigkeit nachweisen können, heißen „Falsifikatoren“.

Eine Falsifikation besteht aus dem Nachweis immanenter Inkonsistenzen bzw. Widersprüche (*Kontradiktion*) oder der Unvereinbarkeit mit als [wahr](#) akzeptierten Instanzen (*Widerspruch zu Axiomen*) oder aus der Aufdeckung eines *Irrtums*. Methodisch konfrontiert man die widersprüchlichen Aussagen, die aus der Ausgangsbehauptung folgen, als Gegenhypothese oder [Antithese](#).



Sind alle Schwäne weiß? Die klassische  Sicht der Wissenschaftstheorie war, dass es Aufgabe der Wissenschaft ist, solche **Hypothesen** zu „beweisen“ oder aus Beobachtungsdaten herzuleiten. Das erscheint jedoch schwer möglich, da dazu von Einzelfällen auf eine allgemeine Regel geschlossen werden müsste, was logisch nicht zulässig ist. Doch ein einziger schwarzer Schwan erlaubt den logischen Schluss, dass die Aussage, alle Schwäne seien weiß, falsch ist. Der Falsifikationismus strebt somit nach einem Hinterfragen, einer **Falsifizierung** von Hypothesen, statt nach dem Versuch eines Beweises.

FALSIFIZIERUNG

- man kann nichts (!) beweisen, sondern nur falsifizieren

Frameworks für viele Sprachen vorhanden

- Haskell: [QuickCheck](#) ("das Original")
- C# / F#: [FsCheck](#)
- Java: [junit-quickCheck](#), [QuickTheories](#), [jqvwik](#)
- Javascript/Typescript: [JsVerify](#), [fast check](#)
- ...

BEISPIEL

```
using FsCheck.Xunit;  
  
[Property]  
public void Generating_stuff_can_fail(int someNumber)  
{  
    someNumber.Should().BeGreaterThan(-11);  
}
```

```
19      [Property(Verbose = true)]  
      0 references | Run Test | Debug Test  
20      public void Generating_stuff_can_fail(int a)  
21      {  
22          a.Should().BeGreaterOrEqualTo(-11);  
23      }  
24  
25  }  
26 }  
27
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
at FsCheck.Testable.evaluate[a,b](FSharpFunc`2 body, a a)
```

Standard Output Messages:

0:

0

1:

0

2:

1

3:

/

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

14:
-7

15:
6

16:
-10

17:
-13

shrink:
-12

Total tests: 7. Passed: 6. Failed: 1. Skipped: 0.

Test Run Failed.

Test execution time: 2.2570 Seconds

watch : Exited with error code 1


```
Starting test execution, please wait...
[xUnit.net 00:00:01.31]      Demo.PropertyBasedTests.Generating_stuff_can_fail [FAIL]
Failed      Demo.PropertyBasedTests.Generating_stuff_can_fail
Error Message:
  FsCheck.Xunit.PropertyFailedException :
  Falsifiable, after 17 tests (2 shrinks) (StdGen (822947264,296551605)):
Original:
-14
Shrunk:
-12

---- Expected a to be greater or equal to -11, but found -12.
Stack Trace:
```

```
Falsifiable, after 17 tests (2 shrinks)  
  (StdGen (822947264,296551605)):
```

```
Original:
```

```
-14
```

```
Shrunk:
```

```
-12
```

```
----- Expected a to be greater or equal to -11, but found -12.
```

- nach 17 Zufallstests: Test schlägt fehl
- 2 Shrinks: einfachster Datensatz, der Fehler auslöst
- StdGen: Reproduzierbar

- smarter Testdaten-Generator
- Wie kann uns das helfen die "Add" Methode zu verbessern?
- Die Eigenschaft (Property) von "Add" muss überdacht werden

```
Add(a, b) = Add(b, a)           // Commutativity
Add(a, Add(b, c)) = Add(Add(a, b), c) // Associativity
Add(a, IDENTITY) = a             // Identity
```

WAS IST IDENTITY?

Aufruf der Funktion mit einem Wert und der Identity gibt den Wert zurück

- Bsp.:
 - Addition: Identity ist Zahl 0
 - Multiplikation: Identity ist Zahl 1
- Identity Property unterscheidet Addition von Multiplikation
- Kommutativ und Assoziativ Properties sind gleich

Zu mathematisch?

BEISPIEL: STRING.REVERSE

Eigenschaften von `string.Reverse()`?

- zweimaliger Aufruf liefert wieder das Original
- Ergebnis hat die selbe Laenge wie Original
- Erster Buchstabe des Originals entspricht letztem Buchstaben der Ausgabe
- Zeichen vorne hinzufuegen mit anschliessendem reverse sollte gleich reverse von hinten anhaengen sein.
- Reihenfolge ist an der Mitte gespiegelt
- usw...

BEISPIEL: MD5

- The hash is 32 characters long.
- The hash only contains hexadecimal characters.
- Equal inputs have the same hash.
- The hash is different from its input.
- Similar inputs have significantly different hashes.

Source:

<http://www.erikschieboom.com/2016/02/22/property-based-testing/>

Mit **Arbitraries** Zufallsdaten beeinflussen

```
[Property(Arbitrary = new[] { typeof(NonNullStringArbitrary) })]  
public void Dummy(string input) => input.Should().NotBeNull();
```

```
[MyProperty]  
public void Dummy2(string input) => input.Should().NotBeNull();
```

```
public static class NonNullStringArbitrary  
{  
    public static Arbitrary<string> Strings() =>  
        Arb.Default.String().Filter(x => x != null);  
}
```

```
public class MyPropertyAttribute : PropertyAttribute  
{  
    public MyPropertyAttribute() =>  
        Arbitrary = new[] { typeof(NonNullStringArbitrary) };  
}
```

Conditional Properties: func.When(...)

```
[Property]
public Property Anything_divisible_by_three_but_not_five_returns_fizz(int
{
    Func<bool> property = () => Fizz.Buzz(input) == "Fizz";

    return property.When(input % 3 == 0 && input % 5 != 0);
}
```

LIVE CODING

PROPERTY BASED TESTING




ZUSAMMENFASSUNG

- gut für Algorithmen und alles, was keinen Zustand hat
- Testdatengenerierung
- wie TDD: mehr Design-Tool als Test-Tool
- Gegensatz zu TDD: nicht beispielbasiert!
 - man muss mehr denken □

FAZIT

- **TDD**
 - führt oft zu besserem Design
- **Mutation testing**
 - Qualität von Tests untersuchen
- **Property based testing**
 - neue Eigenschaften vom Code entdecken
 - automatische Testdaten-Generierung

DANKE!

- Slides & Code <https://github.com/redheads/2019-05-10-mathemacampus-testing>
- Kontakt
 -  patrick.drechsler@redheads.de
 -  @drechsler
 -  draptik