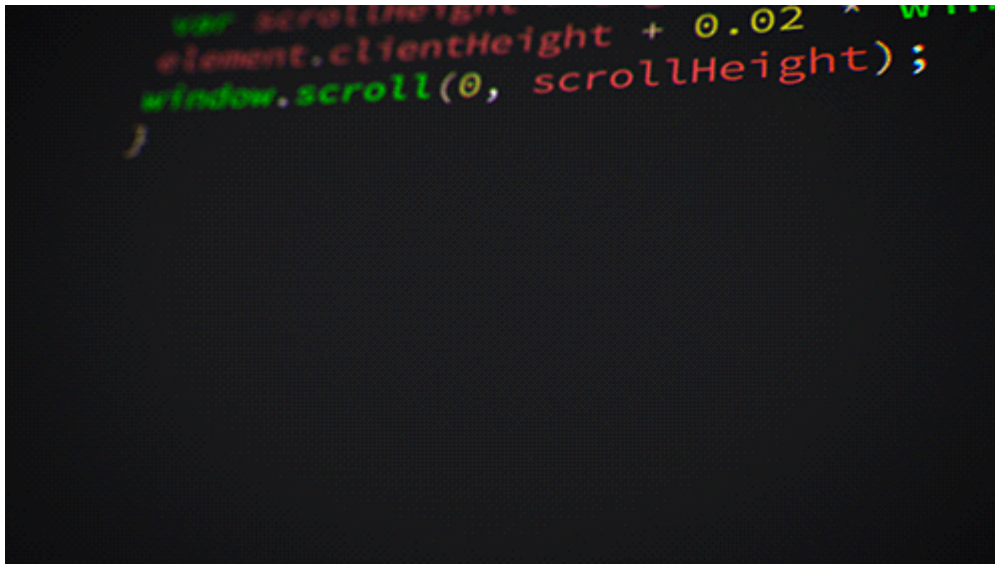


Testen von Software

Parameter	Kursinformationen
Veranstaltung:	Vorlesung Softwareentwicklung
Teil:	19/27
Semester	Sommersemester 2025
Hochschule:	Technische Universität Freiberg
Inhalte:	Softwarefehler, Testen zur Qualitätssicherung, Planung von Tests, Konzepte und Umsetzung in dotnet
Link auf den GitHub:	https://github.com/TUBAF-lfi-LiaScript/VL_Softwareentwicklung/blob/master/19_Testen.md
Autoren	Sebastian Zug, Galina Rudolf & André Dietrich



Softwarefehler

Zu Erinnerung an die bereits diskutierten Softwarefehler ...

1999 verpasste die NASA-Sonde Mars Climate Orbiter den Landeanflug auf den Mars, weil die Programmierer unterschiedliche Maßsysteme verwendeten (ein Team verwendete das metrische und das andere das angloamerikanische) und beim Datenaustausch es so zu falschen Berechnungen kam. Eine Software wurde so programmiert, dass sie sich nicht an die vereinbarte Schnittstelle hielt, in der die metrische Einheit $\text{Newton} \times \text{Sekunde}$ festgelegt war. Die NASA verlor dadurch die Sonde.

[Quelle](#)

Softwarefehler sind sowohl sicherheitstechnisch wie ökonomisch ein erhebliches Risiko. Eine Studie der Zeitschrift iX ermittelte 2013 für Deutschland folgende Werte:

- Ca. 84,4 Mrd. Euro betragen die jährlichen Verluste durch Softwarefehler in Mittelstands- und Großunternehmen
- Ca. 14,4 Mrd. Euro jährlich (35,9 % des IT-Budgets) werden für die Beseitigung von Programmfehlern verwendet;
- Ca. 70 Mrd. Euro betragen die Produktivitätsverluste durch Computerausfälle aufgrund fehlerhafter Software

Was sind Softwarefehler eigentlich?

Ein Programm- oder Softwarefehler ist, angelehnt an die allgemeine Definition für „Fehler“

„Nichterfüllung einer Anforderung“ [EN ISO 9000:2005]

Konkret definiert sich der Fehler danach als

„Abweichung des IST (beobachtete, ermittelte, berechnete Zustände oder Vorgänge) vom SOLL (festgelegte, korrekte Zustände und Vorgänge), wenn sie die vordefinierte Toleranzgrenze [die auch 0 sein kann] überschreitet.“

Im Rahmen dieser Veranstaltung lassen wir Lexikalische Fehler und Syntaxfehler außen vor. Diese sind in der Regel über den Compiler identifizierbar. Darüber hinaus existieren aber :

Fehlertyp	Folgen
Logisch/semantische Fehler	Anweisung ist zwar syntaktisch fehlerfrei, aber inhaltlich trotzdem fehlerhaft (plus statt minus, kleiner statt kleiner gleich, fehlende Synchronisation, usw.)
Designfehler	Strukturelle Mängel auf der Modul oder Systemebene, die das Zusammenspiel der Komponenten, deren Erweiterung, usw. verhindern.
Fehler im Bedienkonzept	Unintuitive Benutzung, das Programm "fühlt sich komisch an"

Darüber hinaus ist es wichtig zwischen Laufzeit- und Designzeitfehlern zu unterscheiden.

Wann entstehen Fehler im Projekt?

Problem- und Systemanalyse:

- Die Anforderungen und Qualitätsmerkmale werden nicht festgelegt.
- Es fehlen eindeutige Begriffsdefinitionen.

Systementwurf:

- Die Systemarchitektur ist gar nicht oder nur mit großem Aufwand erweiterbar.
- Das System ist nicht modular aufgebaut, die Daten sind nicht gekapselt.

Feinentwurf:


- Schnittstellen sind nicht hinreichend spezifiziert
- Interaktionsmodelle weisen Lücken auf

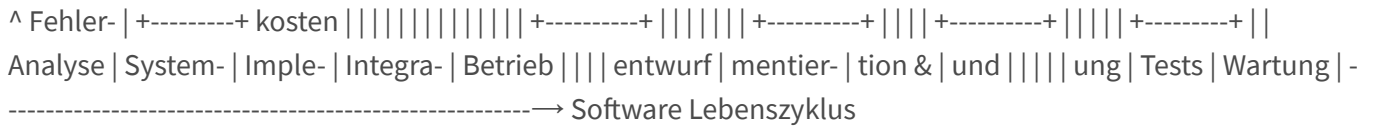
Codierung

- Programmier-Standards bzw. -Richtlinien werden nicht beachtet.
- Die Namensvergabe ist ungünstig.

Betrieb und Wartung:

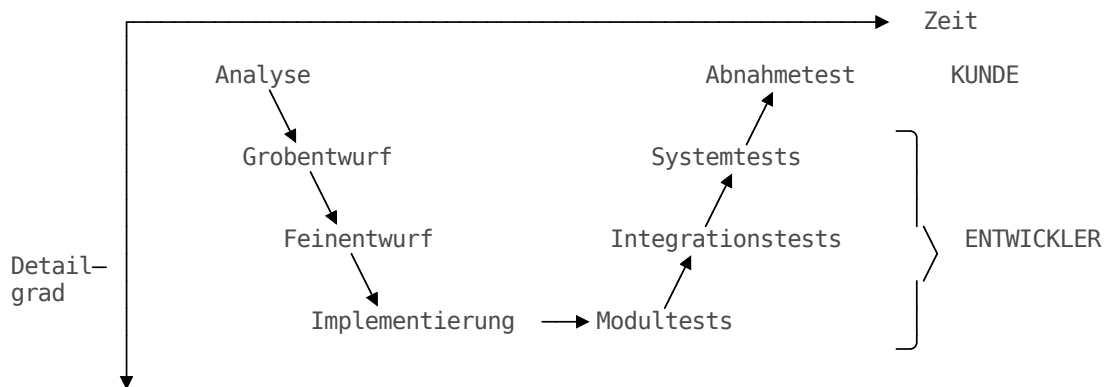
- Die Dokumentation fehlt ganz, ist veraltet oder nicht adäquat.
- Die Schulung der Anwender wird vernachlässigt.
- Das Konfigurationsmanagement ist unzureichend.

<!--https://raw.githubusercontent.com/TUBAF-IfL-LiaScript/VL_Softwareentwicklung/refs/heads/master/19_Testen.md style="width: 100%; max-width: 560px; display: block; margin-left: auto; margin-right: auto;" → 



Testen als Teil der Qualitätssicherung

Welche Tests werden in das Projekt integriert?



| Bezeichnung | Ebene | Durchführender / Ziel |
|--|--|---|
| Modultest,
Komponententest oder
Unittest | Funktionalität innerhalb einzelner abgrenzbarer Teile der Software (Module, Programme oder Unterprogramme, Units oder Klassen) | häufig durch den Softwareentwickler selbst, Nachweis der technischen Lauffähigkeit und korrekter fachlicher (Teil-) Ergebnisse |
| Integrationstest,
Interaktionstest | Zusammenarbeit voneinander abhängiger Komponenten | Testschwerpunkt liegt auf den Schnittstellen der beteiligten Komponenten und soll korrekte Ergebnisse über komplette Abläufe hinweg nachweisen |
| Systemtest | Gesamtes System wird gegen die gesamten Anforderungen (funktionale und nicht-funktionale Anforderungen) getestet | Test in einer Testumgebung statt / wird mit Testdaten durchgeführt - Simulation einer realistischen Umgebung |
| Abnahmetest,
Verfahrenstest,
Akzeptanztest | Testen der gelieferten Software durch den Kunden | Rechtlich bindende Evaluation der Software und deren Bezahlung, unter Umständen bereits auf der Produktionsumgebung mit Kopien aus Echtdaten durchgeführt |

Warum geht es dann trotzdem schief?

- Es ist angeblich keine Zeit für systematische Tests vorhanden (Termindruck).
- Die Notwendigkeit für systematische Tests wird nicht erkannt.
- Die Tests werden manuell realisiert.
- Die Erstellung von Testspezifikationen für systematische Tests wird nicht entwicklungsbegleitend durchgeführt.
- Die Testebenen weisen eine unterschiedliche Realisierung auf (Modultests top, Systemtests flop)

Definition

Es gibt unterschiedliche Definitionen für den Softwaretest:

„the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component.“ [ANSI/IEEE Std. 610.12-1990]

„Test [...] der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen“ ist. [Denert]

"Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen. [Pol]

Welche Unterschiede sehen Sie in den Definitionen?

Unterschied Verifikation vs. Validierung

- **Verifikation** ... ist der Prozess, der sicherstellt, dass ein Softwareprodukt die Spezifikationen erfüllt und korrekt implementiert wurde. (*Bauen wir das Produkt richtig?*)
- **Validierung** ... ist der Prozess, der sicherstellt, dass das Softwareprodukt die Bedürfnisse des Kunden erfüllt und die richtige Software entwickelt wurde. (*Bauen wir das richtige Produkt?*)

[Denert] Ernst Denert: Software-Engineering. Methodische Projektabwicklung. Springer, Berlin u. a. 1991, ISBN 3-540-53404-0.

[Pol] Martin Pol, Tim Koomen, Andreas Spillner: Management und Optimierung des Testprozesses. Ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap. 2., aktualisierte Auflage. dpunkt.Verlag, Heidelberg 2002, ISBN 3-89864-156-2.

Ablauf beim Testen

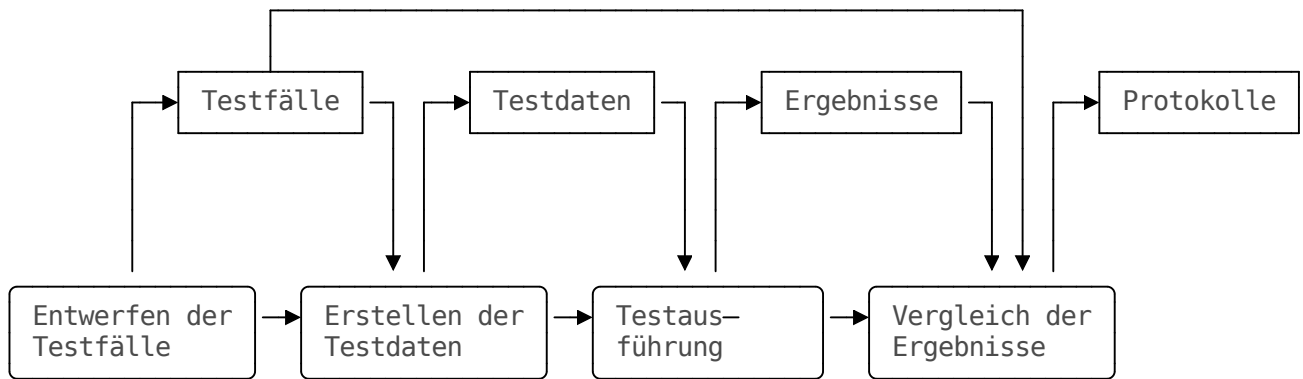


Abbildung motiviert durch [\[Somm01\]](#)

1. Entwerfen der Testfälle

- Analyse der Anforderungen, Dokumentationen um erforderliche Testbedingungen festzulegen
- Nachvollziehbarkeit der Entscheidungen, Weiterentwicklung bei Anpassungen in den Anforderungen bzw. der Spezifikation

2. Spezifizieren der Testfälle

- Ausarbeitung der eigentlichen Beschreibung der Testfälle und Testdaten
- Definition der erwarteten Resultate

3. Testausführung

- (variable) Reihung der Testfälle unter Berücksichtigung von Vor- und Nachbedingungen um Quereffekte abzubilden

4. Evaluation der Ergebnisse

[Somm01] Ian Sommerville: Software Engineering, Pearson Education, 6. Auflage, 2001

Klassifikation Testmethoden

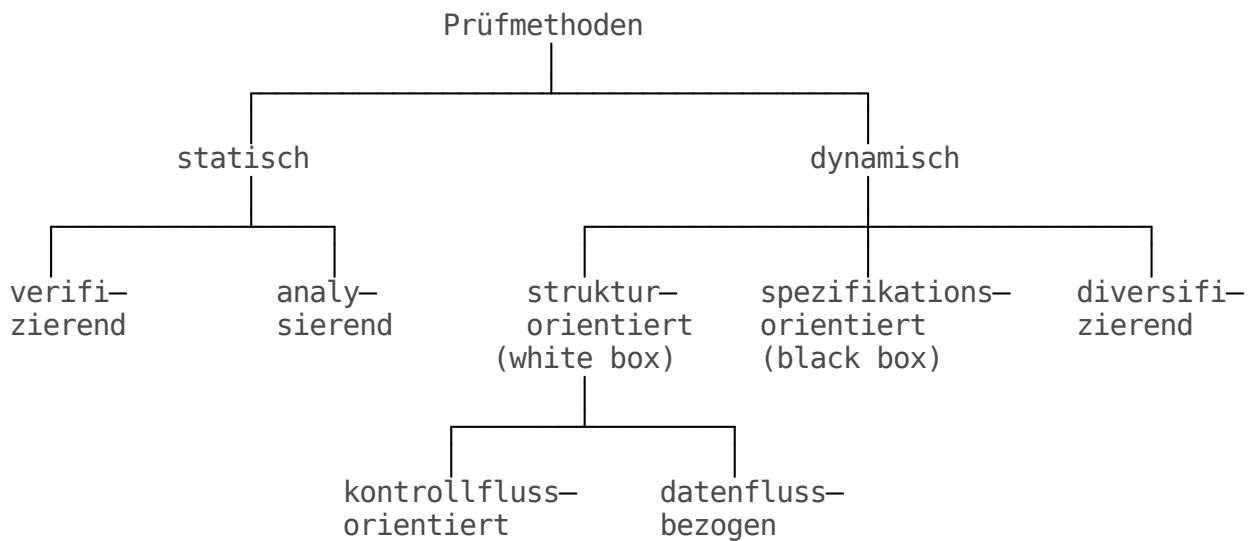


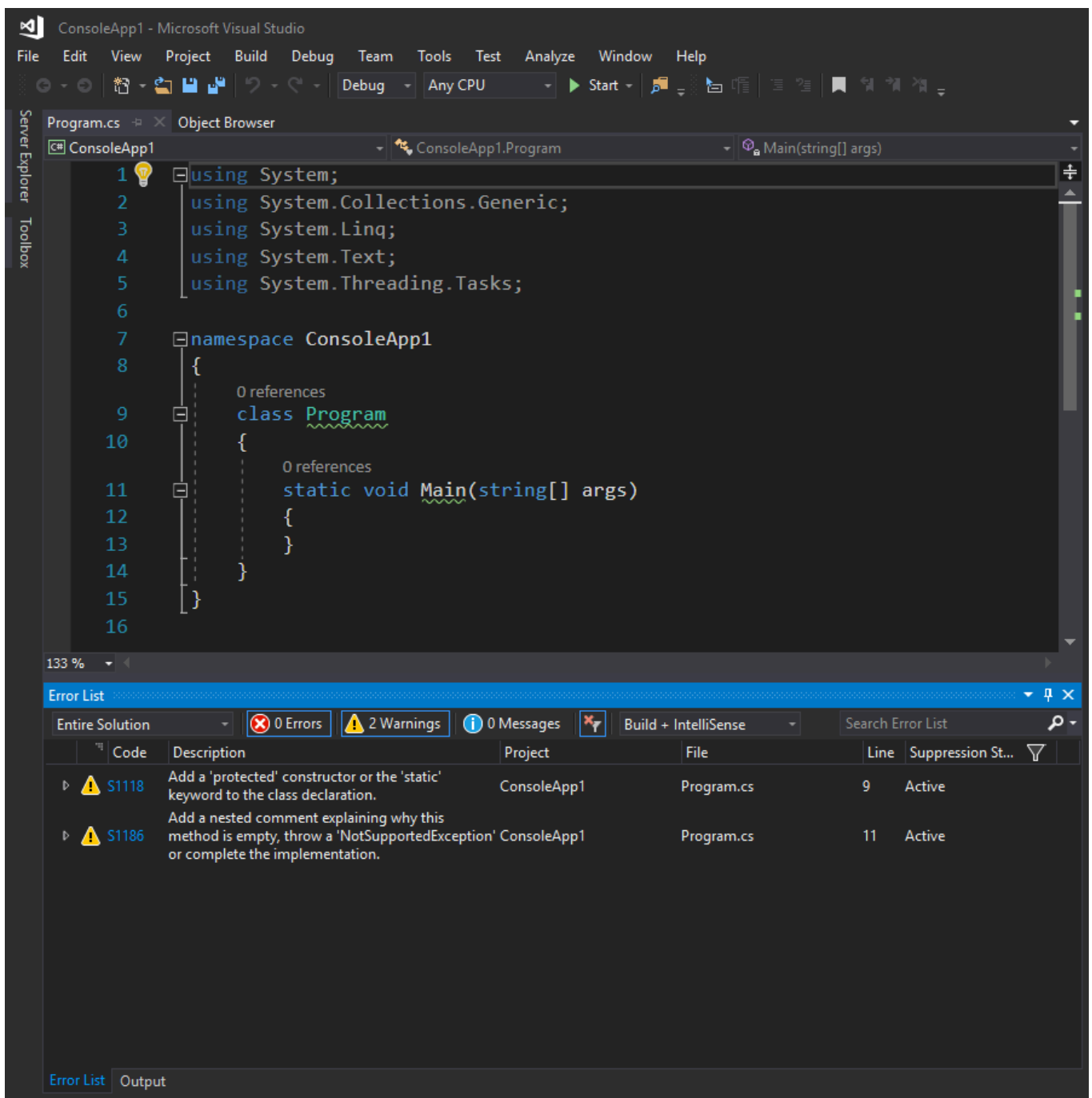
Abbildung motivierte aus [\[Liggesmeyer\]](#)

Statische Code Analysen

... ohne eine Ausführung allein anhand des Codes durchgeführt. Der Quelltext wird hierbei einer Reihe formaler Prüfungen unterzogen, bei denen bestimmte Sorten von Fehlern entdeckt werden können, noch bevor die entsprechende Software (z. B. im Modultest) ausgeführt wird. Die Methodik gehört zu den falsifizierenden Verfahren, d. h., es wird die Anwesenheit von Fehlern bestimmt.

- **Codeanalyse** ... In Anlehnung an das klassische Programm Lint wird der Vorgang der Analyse eines Codefragments auch als linten (englisch linting) bezeichnet.

Das folgende Beispiel zeigt die Ausgabe des Tools SonarLint angewendet auf die initiale Implementierung einer Konsolenanwendung unter Visual Studio 2017. Welche Fehler können Sie ausmachen?



<https://learn.microsoft.com/de-de/dotnet/fundamentals/code-analysis/overview?tabs=net-9>

- **Codereviews** ... Reviews sind manuelle Überprüfungen der Arbeitsergebnisse der Softwareentwicklung. Jedes Arbeitsergebnis kann einer Durchsicht durch eine andere Person unterzogen werden.

Der untersuchte Gegenstand eines Reviews kann verschieden sein. Es wird vor allem zwischen einem Code-Review (Quelltext) und einem Architektur-Review (Softwarearchitektur, insbesondere Design-Dokumente) unterschieden.

<https://www.codereviewchecklist.com/>

- ...

Dynamische Code Analysen

Dynamische Software-Testverfahren sind bestimmte Prüfmethoden, um mit Softwaretests Fehler in der Software aufzudecken. Besonders sollen Programmfehler erkannt werden, die in Abhängigkeit von dynamischen Laufzeitparametern auftreten, wie variierende Eingabeparameter, Laufzeitumgebung oder Nutzer-Interaktion. Wesentliche Aufgabe der einzelnen Verfahren ist die Bestimmung geeigneter Testfälle für den Test der Software.

- **strukturorientiert** ... Strukturorientierte Verfahren bestimmen Testfälle auf Basis des Softwarequellcodes (Whiteboxtest). Dabei steht entweder die enthaltenen Daten oder aber die Kontrollstruktur, die die Verarbeitung der Daten steuert, im Fokus.
- **spezifikationsorientiert** ... die sogenannten Black-Box Verfahren werden zum Abgleich des vorgegebenen, spezifizierten und des realen Verhaltens einer Methode genutzt. Beim Modultest wird z. B. gegen die Modulspezifikation getestet, beim Schnittstellentest gegen die Schnittstellenspezifikation und beim Abnahmetest gegen die fachlichen Anforderungen, wie sie etwa in einem Pflichtenheft niedergelegt sind.
- **diversifizierend** .. Diese Tests analysieren die Ergebnisse verschiedener Versionen einer Software gegeneinander. Es findet entsprechend kein Vergleich zwischen den Testergebnissen und der Spezifikation statt! Zudem kann im Gegensatz zu den funktions- und strukturorientierten Testmethoden kein Vollständigkeitskriterium definiert werden. Die notwendigen Testdaten werden mittels einer der anderen Techniken, per Zufall oder Aufzeichnung einer Benutzer-Session erstellt.

[Liggesmeyer] Peter Liggesmeyer, "Software-Qualität - Testen, Analysieren und Verifizieren von Software", Springer, 2002

Planung von Tests

Nehmen wir an, wir hätten eine Klasse MyMathFunctions mit zwei Methoden implementiert und sollen diese testen ...

```
static class MyMathFunctions{  
    //Fakultät der Zahl i  
    public static int fak(int i) {...}  
    // Grösstergemeinsamer Teiler von i, j und k  
    public static int ggt(int i, int j, int k) {...}  
}
```

Frage: Mit wie vielen Tests könnten wir die Korrektheit der Implementierung nachweisen?

Ein vollständiges Testen aller `int` Werte (2^{31} bis $2^{31} - 1$) bedeutet für die Funktion `fak()` 2^{32} und für `ggt()` $2^{32} \cdot 2^{32} \cdot 2^{32}$ Kombinationen. Testen aller möglichen Eingaben ist damit nicht möglich. Für Variablen mit unbestimmtem Wertebereich (`string`) lässt sich nicht einmal die Menge der möglichen Kombinationen darstellen.

Black-Box-Testing / Spezifikationsorientiert

Black-Box-Testing ... Grundlage der Testfallentwicklung ist die Spezifikation des Moduls. Die Interna des Softwareelements sind nicht bekannt.

Die Güte der Testfälle ist definiert über die Abdeckung möglicher Kombinationen der Eingangsparameter.

Für Black-Box-Testing existieren unterschiedliche Ausprägungen:

- Äquivalenzklassenanalyse
- Grenzwertanalyse [Link](#)
- Zustandsbasierte Testmethoden

Problematisch ist dabei, dass spezifische Lösungen, wie zum Beispiel in folgendem Fall. Der Entwickler hat hier beschlossen die Performance der Berechnung der Fakultät zu steigern, um die Performance des Algorithmus für Werte kleiner 5 zu verbessern (hypothetisches Beispiel!).

```
static class MyMathFunctions{  
    public int fak (int i){  
        if ( i==1 ) return 0;           // Fehler  
        elseif (i == 2) return 1;      // Fehler  
        elseif ... Ergebnisse für 3 und 4 ...  
        elseif (i == 5) return 120;  
        else return i * fak(i-1);  
    }  
}
```

Mit den alleinigen Testfällen `fak(5)==120`, `fak(6)==720` und `fak(10)==3628800` bleiben mögliche Fehler für `fak(1)` und `fak(2)` verborgen.

White-Box-Testing / Strukturorientiert

White-Box-Testing ... beim „quelltextbasierten Testen“ sind die Interna des getesteten Softwareelements bekannt und werden zur Bestimmung der Testfälle verwendet

White-Box-Testing-Verfahren zerlegen das Programm (statisch oder dynamisch) entsprechend dem Kontrollfluss. Die Güte der Testfälle wird danach beurteilt, wie groß der Anteil der abgedeckten Programmpfade ist. Die Bewertung kann dabei anhand differenzierter Metriken erfolgen:

- Zeilenabdeckung
- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- ...

C₀ Anweisungsüberdeckung

Anweisungsüberdeckung (auch C_0 -Test genannt) zerlegt das Programm statisch in seine Anweisungen und bestimmt den Anteil der in den Testfällen berücksichtigten Anweisungen. Üblich ist eine Prüfung von 95%-100% aller Anweisungen durch als C_0 -Kriterium anzustreben:

$$C_0 = \frac{\text{Anzahl überdeckte Anweisungen}}{\text{Gesamtanzahl der Anweisungen}}$$

```
static class MyMathFunctions{
    public int fak (int i){                // Anweisung
        if ( i==1 ) return 0;             // 1
        elseif (i == 2) return 1;         // 2
        elseif ... Ergebnisse für 3 und 4 ... // 3 - 4
        elseif (i == 5) return 120;       // 5
        else return i * fak(i-1);         // 6
    }
}
```

Der oben genannten Black-Box-Test $i = \{5, 6, 10\}$ adressierte lediglich 2 der Anweisungen und generiert damit ein $C_0 = \frac{2}{6} = 0.33$. Mit dem Wissen um die Codestruktur, kann der White-Box-Test sehr schnell den Nachweis erbringen, dass das gezeigte Black-Box-Vorgehen nur unzureichend die Qualität des Codes abprüft.

```
static class MyMathFunctions{
    public int fak (int i){                // Anweisung
        int [] facArray = new int [10];   // 1
        facArray[0] = 1;                  // 2
        facArray[1] = 1;
        ...
        facArray[9] = 1;                  // 9
        // besser:
        // int [] facArray = new int[] { 1, 3, 5, 7, 9 };
        if ( i<10 ) return facArray[i];   // 10 + 11
        else return i * fak(i-1);         // 12
    }
}
```

Mit dem Testfall $i = 1$ lassen sich hingegen vermeintlich $11/12 = 0.91$ der Anweisungen abdecken, die Fehleinschätzung ist aber offensichtlich. Gleichwohl sind die fest hinterlegten Werte aus Erfahrung heraus auch besonders anfällig für Copy-&-Paste-Fehler.

C_1 Zweigüberdeckungstest

Der Zweigüberdeckungstest umfasst den Anweisungsüberdeckungstest vollständig. Für den C1-Test müssen strengere Kriterien erfüllt werden als beim Anweisungsüberdeckungstest. Im Bereich des kontrollflussorientierten Testens wird der Zweigüberdeckungstest als Minimalkriterium angewendet. Mit Hilfe des Zweigüberdeckungstests lassen sich nicht ausführbare Programmzweige aufspüren. Anhand dessen kann man dann Softwareteile, die oft durchlaufen werden, gezielt optimieren.

Die [Zyklomatische Komplexität](#) gibt an, wie viele Testfälle höchstens nötig sind, um eine Zweigüberdeckung zu erreichen.

$$C_1 = \frac{\text{Anzahl überdeckten Zweige}}{\text{Gesamtanzahl der Zweige}}$$

```
static class MyMathFunctions{
    public int fak (int i){
        int [] facArray = new int [10];
        facArray[0] = 1;
        facArray[1] = 1;
        ...
        facArray[9] = 1;
        // besser:
        // int [] facArray = new int[] { 1, 3, 5, 7, 9 };
        if ( i<10 ) return fakArray[i];
        else return i * fak(i-1);
    }
}
```

Mit dem Testfall $i = 1$ ergibt sich eine C_1 -Abdeckung von 0.5.

C_2 Pfadüberdeckung

Das C_1 Kriterium berücksichtigt keine Schleifen im zu untersuchenden Code. Der "Pfad" beschreibt gegenüber dem "Zweig" aber eben auch die mehrfache Ausführung ein und des selben Zweiges. Diese Untersuchung muss entsprechend Schleifen in variabler Durchlaufzahl umsetzen.

C_3 Bedingungsüberdeckungstest

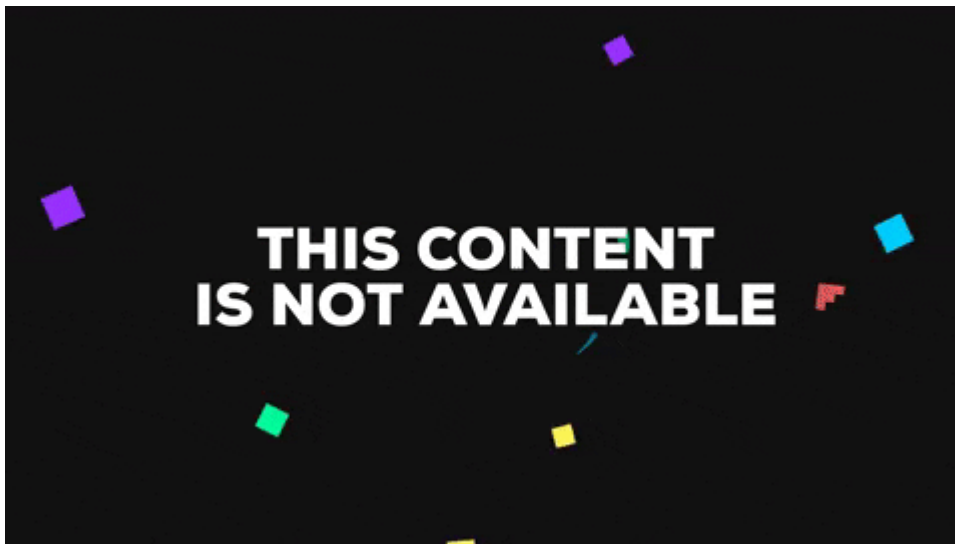
C_3 Tests extrahieren die Bedingungen die zum Eintritt in die Schleifen führen und generieren Testfälle, die alle Kombinationen abdecken.

```
static class MyMathFunctions{
    public int fak (int i){
        boolean a, b;
        if (a || b) { ... }
        else { ... }
    }
}
```

```
}  
}
```

| | Test | Testfälle im Beispiel |
|------|---|--------------------------------------|
| C_3a | Einfachbedingungsüberdeckungstest | 2 (a = b = true sowie a = b = false) |
| C_3b | Mehrfachbedingungsüberdeckungstest | 2^n |
| C_3c | minimaler
Mehrfachbedingungsüberdeckungstest | $\leq 2^n$ |

Und jetzt konkret!



Zu Erinnerung: Testen ist der Vergleich eines Ergebnisses mit einem erwarteten Resultat.

Exkurs: Attribute in C#

Im Folgenden werden wir Attribute als Hilfsmittel verwenden. Entsprechend soll an dieser Stelle ein kurzer Einschub die Möglichkeiten dieser Zuordnung von Metainformationen zum C# Code verdeutlichen.

Attribute erlaube es Zusatzinformationen oder Bedingungen in Code (Assemblies, Typen, Methoden, Eigenschaften usw.) einzubinden. Nach dem Zuordnen eines Attributs zu einer Programmentität kann das Attribut zur Laufzeit mit einer Technik namens Reflektion abgefragt werden.

In C# sind Attribute Klassen, die von der Attribute-Basisklasse erben. Alle Klassen, die von Attribute erben, können als eine Art von „Tag“ für andere Codeelemente verwendet werden. Beispielsweise gibt es das Attribut `ObsoleteAttribute`. Mit diesem Attribut wird gekennzeichnet, dass der Code veraltet ist und nicht mehr verwendet werden sollte.

Beispiele für Standardattribute sind:

| Name | Bedeutung |
|---|--|
| <code>[Obsolete],
[Obsolete("ThisClass is
obsolete. Use ThisClass2
instead.")]</code> | |
| <code>[Conditional("Test")]</code> | Wenn die Zeichenfolge nicht einer #define-Anweisung entspricht, werden alle Aufrufe dieser Methode (aber nicht die Methode selbst) durch den C#-Compiler entfernt. |

Attribute werden in rechteckigen Klammern den jeweiligen Codeelementen vorangestellt. Es können mehrere davon kombiniert werden.



```
1  #define CONDITION1
2  #define CONDITION2
3
4  using System;
5  using System.Diagnostics;
6
7  class Test
8  {
9      static void Main()
10     {
11         Console.WriteLine("Standard Code ");
12         Method0(0);
13         Console.WriteLine("Calling Method1");
14         Method1(3);
15         Console.WriteLine("Calling Method2");
16         Method2();
17     }
18
19     public static void Method0(int x)
20     {
21         Console.WriteLine("Here we run actual algorithm.");
22     }
23
24     [Conditional("CONDITION1")]
25     public static void Method1(int x)
26     {
27         Console.WriteLine("CONDITION1 is defined");
28     }
29
30     [Conditional("CONDITION1"), Conditional("CONDITION2")]
31     public static void Method2()
32     {
33         Console.WriteLine("CONDITION1 or CONDITION2 is defined");
34     }
35 }
```

Standard Code

Here we run actual algorithm.

Calling Method1

CONDITION1 is defined

Calling Method2

CONDITION1 or CONDITION2 is defined

Die Festlegung der Kompilierungsvorgänge anhand von Hinhalten der eigentlichen Code Dateien scheint "unglücklich". Es bietet sich natürlich an, die zugehörigen Konfigurationen in unsere Projektdateien auszulagern.

PreprocessorConsts.cs

```
1 using System;
2 using System.Diagnostics;
3
4 class Test
5 {
6     static void Main()
7     {
8         Console.WriteLine("Standard Code ");
9         Method0(0);
10        Console.WriteLine("Calling Method1");
11        Method1(3);
12        Console.WriteLine("Calling Method2");
13        Method2();
14    }
15
16    public static void Method0(int x)
17    {
18        Console.WriteLine("Here we run actual algorithm.");
19    }
20
21    [Conditional("CONDITION1")]
22    public static void Method1(int x)
23    {
24        Console.WriteLine("CONDITION1 is defined");
25    }
26
27    [Conditional("CONDITION1"), Conditional("CONDITION2")]
28    public static void Method2()
29    {
30        Console.WriteLine("CONDITION1 or CONDITION2 is defined");
31    }
32 }
```

xml PreprocessorConsts.csproj

```
1 <Project Sdk="Microsoft.NET.Sdk">
2   <PropertyGroup>
3       <OutputType>Exe</OutputType>
4       <TargetFramework>net8.0</TargetFramework>
5       <DefineConstants>CONDITION2;</DefineConstants>
6   </PropertyGroup>
7 </Project>
```

```
Standard Code
Here we run actual algorithm.
Calling Method1
Calling Method2
CONDITION1 or CONDITION2 is defined
Standard Code
Here we run actual algorithm.
Calling Method1
Calling Method2
CONDITION1 or CONDITION2 is defined
```

Idee 1: Eigenen Testmethoden



```
1 using System;
2
3 // Zu testende Klasse
4 public class Calculator
5 {
6     public static int DivideTwoValues(double x, double y, ref double re
7     if (y != 0){
8         result = x / y;
9         return 0;
10    }
11    else return -1;
12 }
13 }
14
15 // Testklasse
16 public class TestCalculator{
17     public static void Test_DivideMethod(){
18         double result = 0;
19         int state = Calculator.DivideTwoValues(3,4, ref result);
20         if ((state == 0) & (result == 0.75))
21         {
22             Console.WriteLine("Test bestanden !");
23         }
24         else{
25             Console.WriteLine("Test fehlgeschlagen");
26         }
27     }
28 }
29
30 // Anwendungsprogramm
31 public class Program
32 {
33     public static void Main(string[] args)
34     {
35         //double result = 0;
36         //int state = Calculator.DivideTwoValues(3,4, ref result);
37         //Console.WriteLine($"Das Ergebnis lautet {result}, der State {st
38         );
39         TestCalculator.Test_DivideMethod();
40     }
41 }
```

Test bestanden !

Test bestanden !

Welche Funktionalität fehlt Ihnen in diesem Setup? Welche weitergehenden Features würden Sie für unsere Testmethoden vorschlagen?

Idee 2: Test-Frameworks

```
TestCase MStest

[TestClass] // <-- Framework spezifisch
public class CalculatorTests
{
    [TestMethod] // <-- Framework spezifisch
    public void TestMethod1()
    {
        // Arrange
        double result;
        double x = 3, y = 4;
        int state;
        double expected = 0.75;

        // Act
        int state = Calculator.DivideTwoValues(x, y, ref result);

        // Assert
        Assert.AreEqual(result, expected);
        // ^---- Framework spezifisch
    }
}
```

Vorteile:

- Leistungsfähige API (automatisierte Tests, variable Input-Parameter, Berücksichtigung von Exceptions)
- "Standardisiertes" Nutzungskonzept
- Integration in die Entwicklungsumgebungen

Nachteil:

- verschiedene Interpretationen und Performance der Frameworks

Die wichtigsten Tools unter C# sind [xUnit](#), [nunit](#), [MSTest](#). Einen guten Überblick zum Vergleich der Schlüsselworte liefert [Link](#)

Hierzu nutzen wir das xunit Framework. Eine Folge von Tests für unsere `DivideTwoValues()` Methode könnte dann wie folgt aussehen.

```
using Xunit;

public class Test_DivideTwoValues
{
    ..
```

```
[Fact]
public void Check_StateEqualPositiveInputs()
{
    // Arrange
    double result = 0;
    double dividend = 5;
    double divisor = dividend;
    int expected = 0;
    // Act
    var state = Calculator.DivideTwoValues(dividend, divisor, ref result);
    // Assert
    Assert.Equal(expected, state);
}
```

```
[Fact]
public void Check_StateZeroAsDivended()
{
    // Arrange
    double result = 0;
    double dividend = 5;
    double divisor = 0;
    int expected = -1;
    // Act
    var state = Calculator.DivideTwoValues(dividend, divisor, ref result);
    // Assert
    Assert.True(expected == state);
}
```

```
[Theory] // Übergabe von
    variablen Parametersets
[InlineData(10, 2, 5)]
[InlineData(5, 2, 2.5)]
[InlineData(double.MaxValue, double.MaxValue, 1)] // Edge Cases
[InlineData(double.MaxValue, 1, double.MaxValue)]
public void Check_ResultCalculation(double dividend, double divisor, double
    expected)
{
    // Arrange
    double result = 0;
    // Act
    var state = Calculator.DivideTwoValues(dividend, divisor, ref result);
    // Assert
    Assert.Equal(expected, result);
}
```

Wie setzen wir das Ganze um?

```
dotnet new sln -o unit-testing-example
cd unit-testing-example
```



```
dotnet new classlib -o CalcService          // Code der Divisionsoperation
    einfügen
mv CalcService/Class1.cs CalcService/Division.cs
dotnet sln add ./CalcService/CalcService.csproj
dotnet new xunit -o CalcService.Tests      // obigen Testcode einfügen
dotnet add ./CalcService.Tests/CalcService.Tests.csproj reference ./CalcService/CalcService.csproj
dotnet sln add ./CalcService.Tests/CalcService.Tests.csproj
```

Damit entsteht eine **solution**, die zwei **project** umfasst - die eigentliche Anwendung als **classlib** und die Testfälle.

```
.
├── CalcService
│   ├── CalcService.csproj
│   └── Division.cs
├── CalcService.Tests
│   ├── CalcService.Tests.csproj
│   └── UnitTest1.cs
└── unit-testing-example.sln
```

Das Ausführen der Tests ist nun mit **dotnet test** möglich.

Eine automatische Generierung von Test Merkmalen ist mit Hilfe zusätzlicher Tools, die in dotnet integriert sind möglich.

```
dotnet add package coverlet.collector
dotnet tool install --global dotnet-reportgenerator-globaltool
dotnet tool install --global coverlet.console

dotnet test --collect:"XPlat Code Coverage" --results-directory:"./coverage"
reportgenerator "-reports:coverage/**/*cobertura.xml" "-targetdir:coverage-report/" "-reporttypes:HTML;"
```

Das Argument "XPlat Code Coverage" bezieht sich auf das Zwischenformat der Darstellung. Das **./coverage** dient zur Angabe des Verzeichnisses, in dem die Ergebnisse gespeichert werden sollen. Wenn keines angegeben wird, wird standardmäßig ein TestResults-Verzeichnis innerhalb jedes Projekts verwendet. **reportgenerator** erzeugt dann die entsprechende html-Repräsentation.

< Summary

| | |
|--------------------------|--|
| Class: | CalcService.Calculator |
| Assembly: | CalcService |
| File(s): | /home/zug/Desktop/Vorlesungen/VL_Softwareentwicklung/code/16_Testen/unit-testing-example/CalcService/Division.cs |
| Covered lines: | 6 |
| Uncovered lines: | 0 |
| Coverable lines: | 6 |
| Total lines: | 15 |
| Line coverage: | 100% (6 of 6) |
| Covered branches: | 2 |
| Total branches: | 2 |
| Branch coverage: | 100% (2 of 2) |

Metrics

| Method | Branch coverage i | Cyclomatic complexity i | Line coverage i |
|----------------------|-----------------------------------|---|---------------------------------|
| DivideTwoValues(...) | 100% | 2 | 100% |

File(s)

/home/zug/Desktop/Vorlesungen/VL_Softwareentwicklung/code/16_Testen/unit-testing-example/CalcService/Division.cs

| # | Line | Line coverage |
|----|------|---|
| | 1 | using System; |
| | 2 | |
| | 3 | namespace CalcService |
| | 4 | { |
| | 5 | public class Calculator |
| | 6 | { |
| 6 | 7 | public static int DivideTwoValues(double x, double y, ref double result){ |
| 11 | 8 | if (y != 0){ |
| 5 | 9 | result = x / y; |
| 5 | 10 | return 0; |
| | 11 | } |
| 1 | 12 | else return -1; |
| 6 | 13 | } |
| | 14 | } |

Im Projektordner finden Sie die gesamte Testimplementierung. Diese wurde um eine Python Applikation erweitert, die eine Sprachübergreifende Nutzung einer Csharp Bibliothek illustriert.

Fazit

Testen auf Modulebene - reicht das aus?

Testen auf Modulebene ist ein wichtiger Bestandteil der Softwareentwicklung, aber es ist nicht ausreichend, um die Qualität eines gesamten Systems zu gewährleisten. Es deckt nur die kleinsten Einheiten ab und stellt sicher, dass diese korrekt funktionieren. Allerdings können Fehler in der Interaktion zwischen Modulen oder in der Systemintegration unentdeckt bleiben.

Erweiterung

| Testart | Fokus | Isolation | Beispiel |
|------------------------|------------------|-------------|-------------------------------------|
| Unit-Test | Methode/Funktion | vollständig | <code>Addiere(int a, int b)</code> |
| Modul-/Komponententest | Klasse/Modul | teilweise | <code>Warenkorb.AddArtikel()</code> |
| Integrationstest | mehrere Module | gering | Bestellung → Lager → Versand |
| Systemtest | gesamte App | keine | App starten und Bestellung testen |

Testen auf Modulebene

Modul oder Komponententests sind Tests, die sich auf einzelne Module oder Komponenten einer Software konzentrieren. Sie überprüfen die Funktionalität und das Verhalten dieser Module isoliert von anderen Teilen des Systems.

```
using Xunit;

public class WarenkorbTests {
    [Fact]
    public void Test_Gesamtpreis_fuer_mehrere_Artikel() {
        // Arrange
        var korb = new Warenkorb();
        korb.Hinzufügen(new Artikel { Name = "Buch", Preis = 10.0m });
        korb.Hinzufügen(new Artikel { Name = "Stift", Preis = 2.0m });

        // Act
        var gesamt = korb.Gesamtpreis();

        // Assert
        Assert.Equal(12.0m, gesamt);
    }
}
```

In der realen Software bestehen viele Klassen aus Abhängigkeiten zu anderen Komponenten – z. B. Datenbanken, externe Dienste oder Services.

Mocks sind Test-Doubles, mit denen du diese Abhängigkeiten im Test ersetzen kannst, um:

- das Verhalten der Komponente isoliert zu testen
- kontrollierte Rückgaben zu simulieren
- Seiteneffekte zu vermeiden

Warum ist Mocking wichtig?

Ohne Mocks würdest du in jedem Testfall:

- eine echte Datenbank ansprechen
- eine E-Mail versenden
- einen Webservice kontaktieren

Das macht Tests langsam, fehleranfällig und unzuverlässig.

Best Practices beim Mocking

- Mocke nur explizite Abhängigkeiten (nicht alles!)
- Nutze Interfaces oder abstrakte Klassen als Testanker
- Verwende `.Setup(...)` nur für erwartetes Verhalten
- Nutze `.Verify(...)` zur Kontrolle von Aufrufen (z. B. ob ein E-Mail-Versand ausgelöst wurde)

Produktionscode:

```
public interface IPreisDienst {
    decimal GibPreis(string artikelId);
}

public class Kasse {
    private readonly IPreisDienst _preisDienst;

    public Kasse(IPreisDienst preisDienst) {
        _preisDienst = preisDienst;
    }

    public decimal BerechneGesamtpreis(string artikelId, int menge) {
        var einzelpreis = _preisDienst.GibPreis(artikelId);
        return einzelpreis * menge;
    }
}
```

Testcode mit Mocking:

```
using Moq;
using Xunit;

public class KasseTests {
    [Fact]
    public void BerechneGesamtpreis mit MockDienst() {
```

```

// Arrange
var mockDienst = new Mock<IPreisDienst>();
mockDienst.Setup(d => d.GibPreis("A1")).Returns(10.0m);

var kasse = new Kasse(mockDienst.Object);

// Act
var gesamt = kasse.BerechneGesamtpreis("A1", 3);

// Assert
Assert.Equal(30.0m, gesamt);
}
}

```

Integrationstests

Ein **Integrationstest** prüft das **Zusammenspiel mehrerer Module oder Klassen**, z. B.:

- Controller ↔ Service ↔ Datenbank
- UI ↔ Backend ↔ API
- Repository ↔ Domain-Logik

Ziel ist es, **Schnittstellenfehler** und **Zusammenarbeitsprobleme** zu erkennen – bevor das System als Ganzes getestet wird.

Produktionscode:

```

// Domainmodell
public class Artikel {
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Preis { get; set; }
}


// Repository
public class ArtikelRepository {
    private readonly DbContext _ctx;
    public ArtikelRepository(DbContext ctx) => _ctx = ctx;

    public void Speichern(Artikel a) {
        _ctx.Add(a);
        _ctx.SaveChanges();
    }

    public Artikel? Finde(int id) => _ctx.Set<Artikel>().Find(id);
}

```

Testcode mit Mocking:



```
using Xunit;
using Microsoft.EntityFrameworkCore;

public class ArtikelRepositoryTests {
    [Fact]
    public void Speichern_und_Lesen_von_Artikeln() {
        // Arrange: In-Memory-Kontext
        var options = new DbContextOptionsBuilder<DbContext>()
            .UseInMemoryDatabase("TestDB")
            .Options;

        using var ctx = new DbContext(options);
        var repo = new ArtikelRepository(ctx);
        var artikel = new Artikel { Name = "Test", Preis = 5.0m };

        // Act
        repo.Speichern(artikel);
        var gelesen = repo.Finde(artikel.Id);

        // Assert
        Assert.NotNull(gelesen);
        Assert.Equal("Test", gelesen?.Name);
    }
}
```

Anstatt einen echten Datenbankserver zu verwenden, nutzen wir eine **In-Memory-Datenbank** für die Tests. Diese ermöglicht aber auch wesentlich konkrete Umsetzungen als die Mock-Objekte, da sie die tatsächliche Datenbank-Logik "simuliert".

Testen auf Systemebene

Ein **Systemtest** überprüft das **gesamte Systemverhalten** aus Sicht des Endnutzers. Dabei wird die gesamte Anwendung als Black Box getestet – **alle Komponenten, Module und Schnittstellen** sind integriert.

Ziel: Sicherstellen, dass das System als Ganzes die Anforderungen erfüllt.




Abgrenzung zu anderen Tests

| Testart | Fokus | Perspektive |
|------------------|-------------------------------|-----------------|
| Unittest | Einzelne Methode | Entwickler |
| Komponententest | Klasse/Modul | Entwickler |
| Integrationstest | Zusammenspiel mehrerer Module | Entwickler |
| Systemtest | Gesamtsystem | Nutzer / Tester |

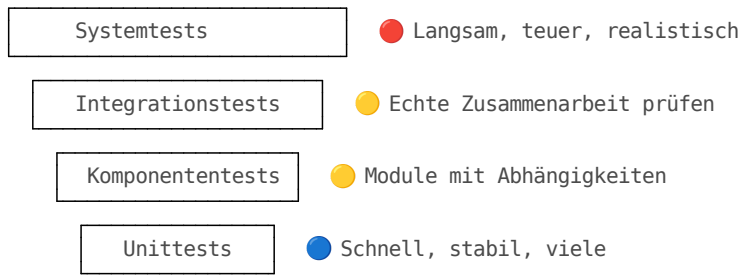
Eigenschaften von Systemtests

- Arbeiten mit **realen oder simulierten Datenbanken, Schnittstellen, UI**
- Testen **End-to-End-Szenarien** (z. B. Anmeldung, Bestellung, Zahlung)
- Häufig **automatisiert** mit Tools wie Selenium, Playwright oder TestServer
- Können auch **manuell** durchgeführt werden (z. B. nach Checklisten)

Vergleich

| Kriterium | Methodentest
(Unit Test) | Komponententest | Integrationstest |
|---------------------|---|--|--|
| Testobjekt | Einzelne Methode oder Funktion | Klasse oder Modul mit internen Abhängigkeiten | Zusammenspiel mehrerer Komponenten/Module |
| Ziel | Korrektheit der kleinsten Einheit | Zusammenarbeit mehrerer Funktionen | Schnittstellen und Zusammenarbeit testen |
| Abhängigkeiten | Werden meist gemockt oder isoliert | Können teilweise eingebunden oder ersetzt sein | Echte Implementierungen (z. B. DB, Services) |
| Beispiel | <code>CalculateSum(int a, int b)</code> | <code>UserService</code> mit <code>EmailService</code> | <code>UserController</code> ↔ <code>UserRepository</code> (mit DB) |
| Tools | xUnit, NUnit | xUnit + Moq/Fakes | xUnit + InMemory DB / Testcontainers |
| Laufzeit | Sehr kurz | Mittel | Mittel bis lang |
| Testgeschwindigkeit |  Schnell |  Mittel |  Mittel |
| Zuverlässigkeit | Hoch (bei guter Isolation) | Mittel (Abhängigkeiten können stören) | Mittel (mehr Fehlerquellen möglich) |
| Fehlersuche | Sehr präzise | Eingrenzbar | Eingrenzbar mit Fokus auf Schnittstellen |
| CI/CD-Einsatz | Immer | Häufig | Häufig / nach jedem Build |

Andere Darstellung



- > ▲ Je weiter oben, desto aufwändiger
- > ▼ Je weiter unten, desto mehr Tests sollten existieren