Automatisierte Tests und TDD mit Visual Studio

.NET User Group Regensburg

26.01.2015

Agenda

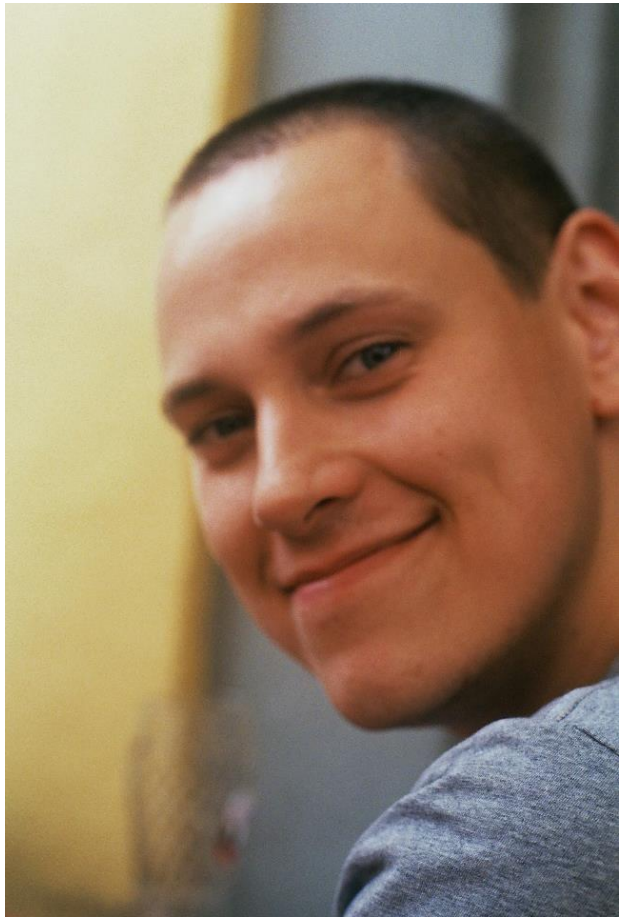
- Automatisierte Tests am Beispiel von Vier Gewinnt
  - Was und Warum?
  - Test-Frameworks für .NET?
  - Testarten?
  - Wie Tests sinnvoll aufbauen?
- Test-Driven-Development
  - Der Red-Green-Refactor-Commit Zyklus
  - Design Patterns für Tests

Zu meiner Person

Kenny Pflug

- Promovend Medieninformatik
- Wissenschaftlicher Mitarbeiter
- Lehrbeauftragter objektorientiertes Programmieren

@feO2x



Automatisierte Tests am Beispiel Vier Gewinnt

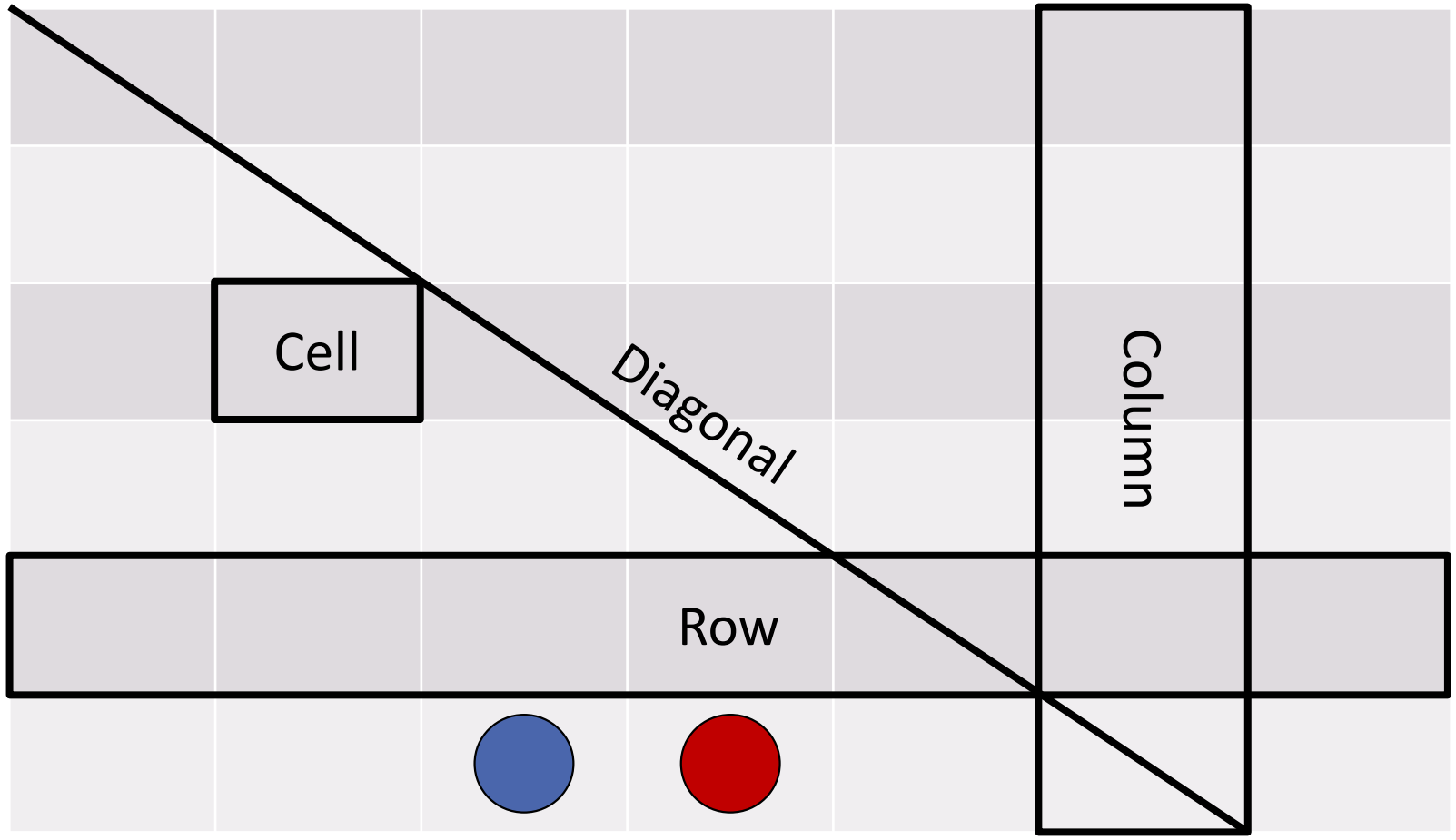
Live Coding

Was war nochmal Vier Gewinnt?

Das Spiel wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in das die Spieler abwechselnd ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten (senkrecht) und sechs Reihen (waagerecht). Jeder Spieler besitzt 21 gleichfarbige Spielsteine. Wenn ein Spieler einen Spielstein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte. Gewinner ist der Spieler, der es als erster schafft, vier oder mehr seiner Spielsteine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Spielbrett komplett gefüllt ist, ohne dass ein Spieler eine Viererlinie gebildet hat.

-Wikipedia

Objektorientierte Analyse



Source Code für dieses Beispiel

Der Source Code zu Vier Gewinnt ist hier zu finden:
https://github.com/feO2x/NetUserGroupRegensburg.ConnectFour

Test Frameworks für .NET

Es existieren sehr viele Frameworks, die drei gängigsten sind aber:

- xunit.net
- Nunit
- MSTest

Wir sehen uns heute MSTest und xunit.net an.

Anatomie eines automatisierten Tests

```
[TestMethod]
public void ChipCanBeAddedToColumnWhenItIsNotFull()
{
    // Arrange
    var columnCells = new List<Cell>
                      {
                            new Cell(0, 0), …
                            new Cell(0, 5)
                      };
    var testTarget = new Column(columnCells);
    var chip = new Chip("Foo", new Color(128, 0, 0));

    // Act
    testTarget.SetChip(chip);

    // Assert
    Assert.AreEqual(chip, columnCells[0].Chip);
}
```

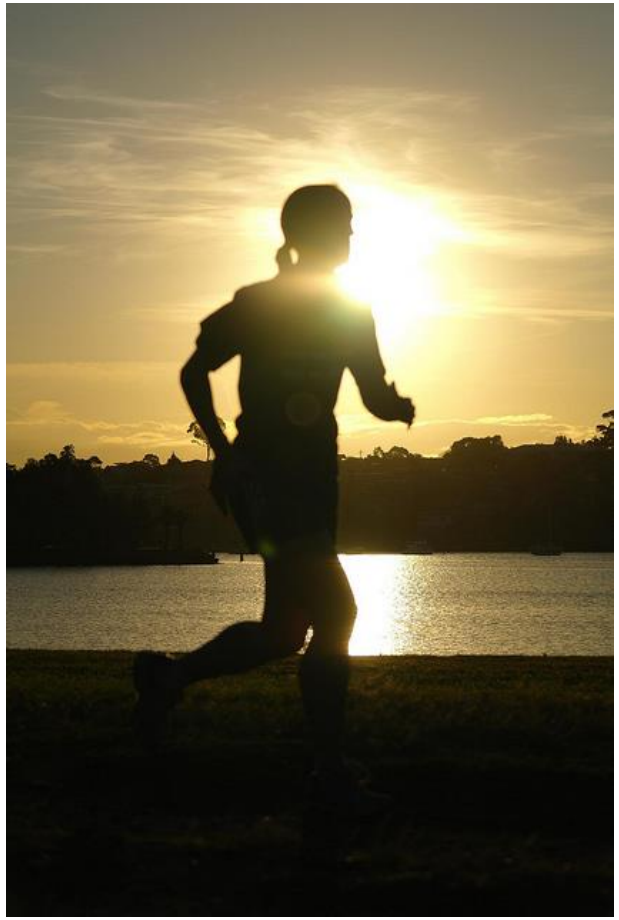
Folgende Phasen sollten durchlaufen werden:

- **Arrange**: hier werden alle Objekte erstellt, die für diesen Test benötigt werden, insbesondere das **System Under Test (SUT)**.
- **Act**: hier wird die Funktionalität, die getestet werden soll, aufgerufen.
- **Assert**: Hier wird überprüft, ob die Act-Funktionalität das gewünschte Ergebnis gebracht hat.

Dieses Muster ist eine Abwandlung des Four-Phase-Test.

Der Test Runner

- Ein Test Runner findet alle Testmethoden in einer Solution und führt diese aus.
- Tritt bei einem Test eine Exception auf, gilt dieser als fehlgeschlagen und wird mit rot markiert.
- Ansonsten gilt der Test als erfolgreich und wird grün markiert.



Wann sollte ich Tests laufen lassen?

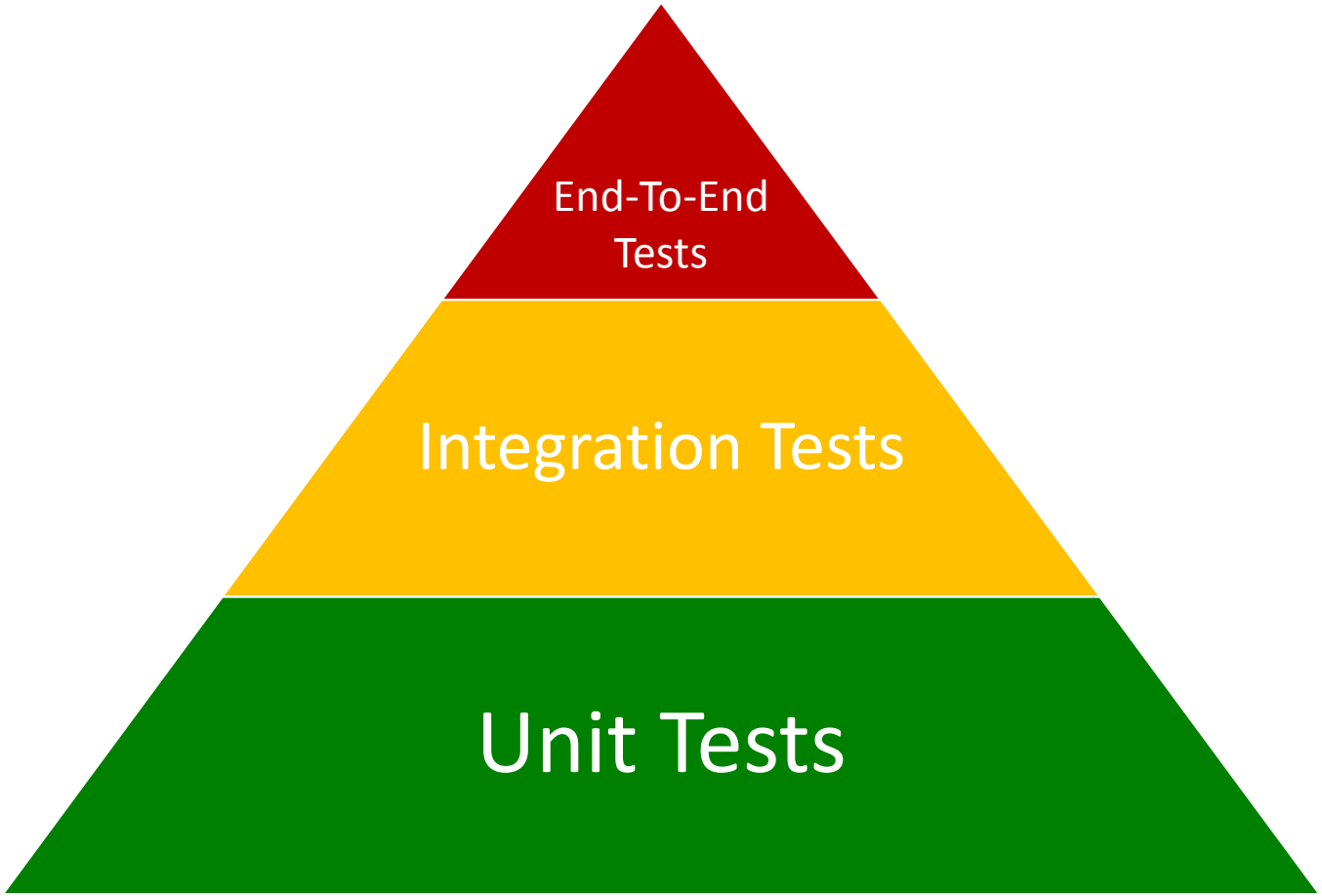
- Test Runs können jederzeit vom Programmierer gestartet werden
  → Schnelles Feedback
- Test Runs werden üblicherweise auch bei einem Build & Deploymentvorgang auf dem Server gefahren, um sicherzustellen, dass die Software den Anforderungen entspricht
- Insbesondere nach Merge-Operationen ist das Abspielen automatisierter Tests sinnvoll (Go To Fail Bug)

Was teste ich hier genau?

Automatisierte Tests werden bezüglich ihres Wirkungsbereichs in drei Kategorien eingeteilt:

- **Unit Tests**: hier wird ein Objekt in kompletter Isolation getestet. Abhängigkeiten zu anderen Objekten werden durch Test Doubles ersetzt.
- **Integrationstests**: hier werden mehrere Objekte im Verbund getestet. Bestimmte Objekte können dennoch durch Test Doubles ersetzt werden, externe Systeme müssen nicht unbedingt direkt angesprochen werden.
- **End-To-End-Tests**: hier wird das komplette System mit allen Produktionskomponenten getestet (keine Test Doubles).

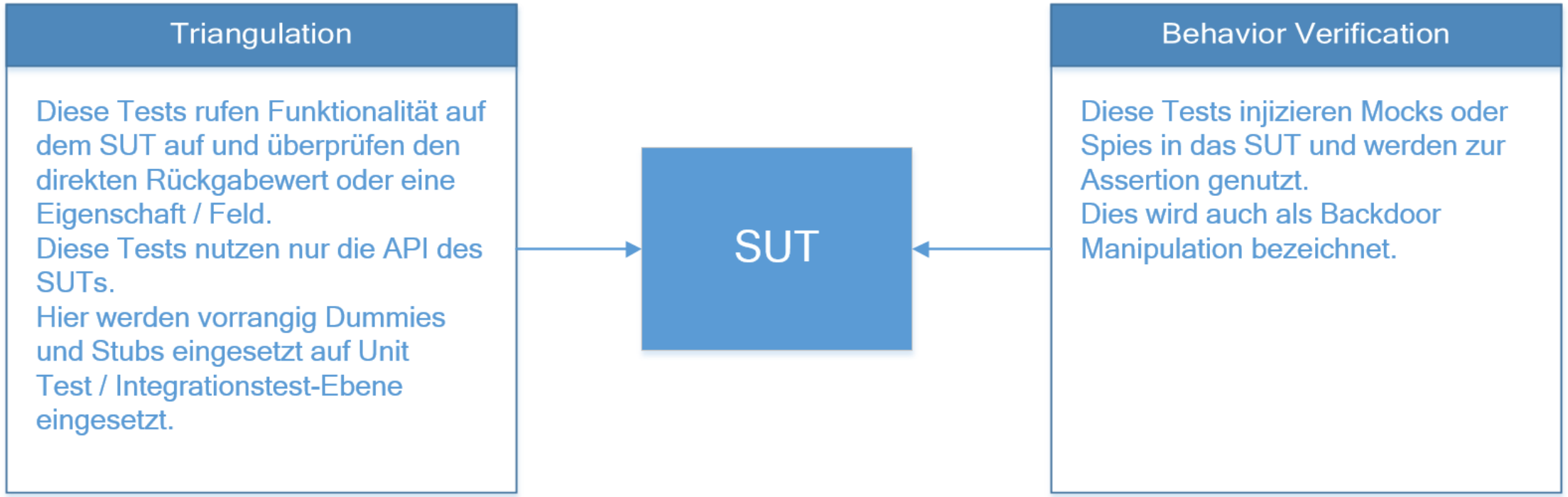
Die Testpyramide



Test Doubles

- **Dummy**: ein Objekt, dass als Platzhalter in einem Test genutzt wird, aber eigentlich keine eigene Funktionalität enthält.
- **Stub**: ein Objekt, dass eine echte Komponente, die das SUT referenziert, ersetzt und Eingaben für das SUT liefert, die es zum Ablauf eines Tests braucht.
- **Spy**: Zeichnet auf, ob das SUT eine bestimmte Methode aufgerufen hat (und wie häufig, mit welchen Parametern).
- **Mock**: Kombination aus Stub und Spy – ein Mock liefert Eingaben zurück und zeichnet auf. Zusätzlich löst er eine Exception aus, wenn bestimmte Punkte vom SUT nicht erfüllt worden sind.
- **Fake**: ein Objekt, dass eine reale Komponente nachbildet, aber eigentlich nicht zur Test-Assertion herangezogen wird (bspw. eine In-Memory-Datenbank).

Auf welche Arten kann ich SUTs testen?



Anders ausgedrückt

Triangulation

Überprüfe, was über die API des SUTs sichtbar ist

Behavior Verification

Überprüfe, ob das SUT mit anderen Objekten korrekt kommuniziert

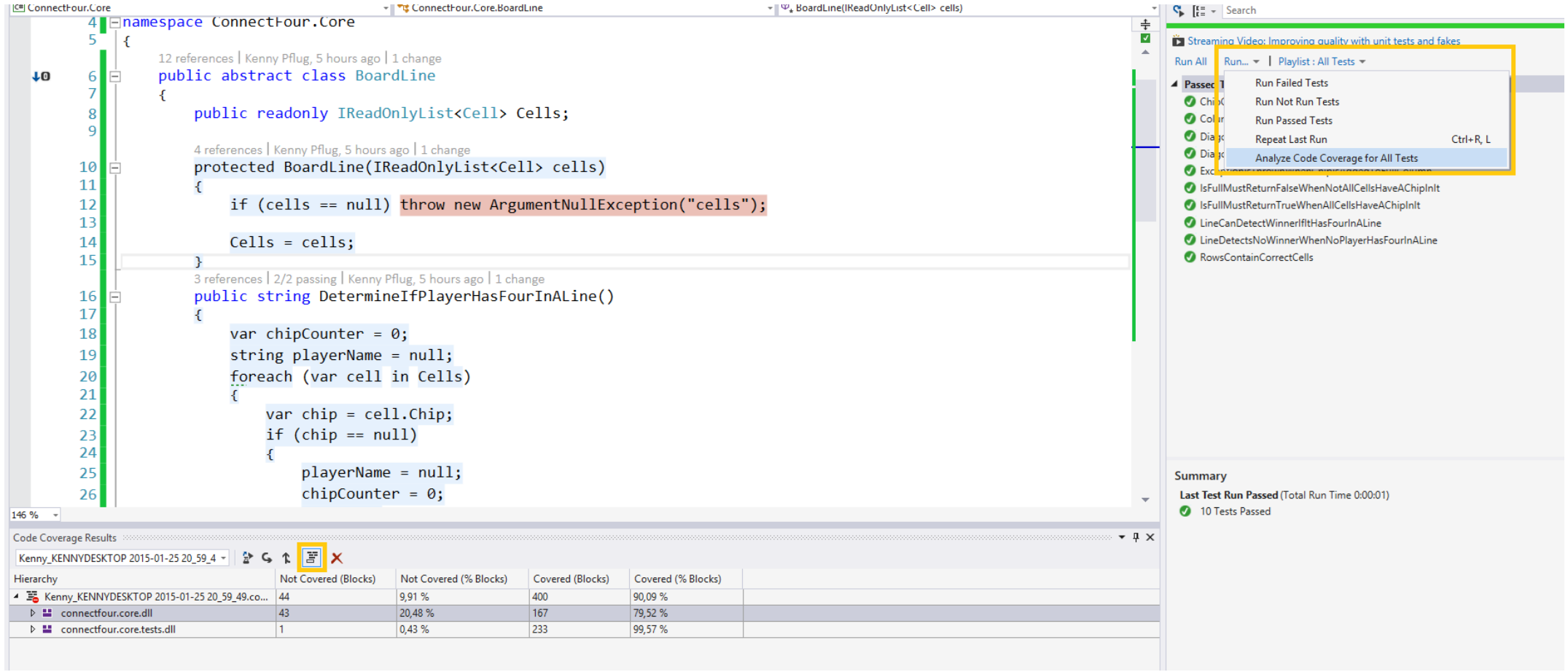
Wie viele Asserts pro Test?

- Idealerweise genau **ein Assert pro Unit Test** (denn diese lösen eine Exception aus, wenn eine Annahme nicht zutrifft).
- Manchmal lassen sich zwei oder drei Asserts nicht vermeiden.
- Refaktorieren Sie ihren Code, sobald sie mehr als vier Assertions in ihrem Test haben. Üblicherweise wird die Überprüfung, die man in mehreren Asserts durchführt, in eine Methode gebündelt, diese unabhängig getestet und dann für eine einzelnes Assert-Statement aufgerufen.
- Bei End-To-End Tests (oder umfangreichen Integrationstests) können durchaus mehrere Asserts in einer Methode vorkommen.

Test Initialize und Teardown

- Jedes Test-Framework bietet an, bestimmte Methoden vor und nach einem Test auszuführen.
- Diese Methoden (insbesondere die Teardown-Methode) wird in jedem Fall ausgeführt (egal of die Methode abbricht oder nicht).
- In Managed Languages ist es oft nicht notwendig, eine Teardown-Methode anzugeben (übernimmt der Garbage Collector).

Code Coverage (1)



Code Coverage (2)

- Ein hoher Code-Coverage-Wert sagt nichts über die Qualität der Tests aus und ob Sie alle wichtigen Fälle abdecken.
- Ein niedriger Code-Coverage-Wert zeigt auf, dass zu wenig Tests existieren.
- Code-Coverage-Werte von ca. 70% – 80% sind üblicherweise normal, können aber je nach Kontext abweichen.
- Die Qualität Ihres Produktions- und Testscodes sollten Sie mit anderen Metriken bestimmen.

FIRST

Tests sollten laut dem FIRST-Paradigma wie folgt aussehen:

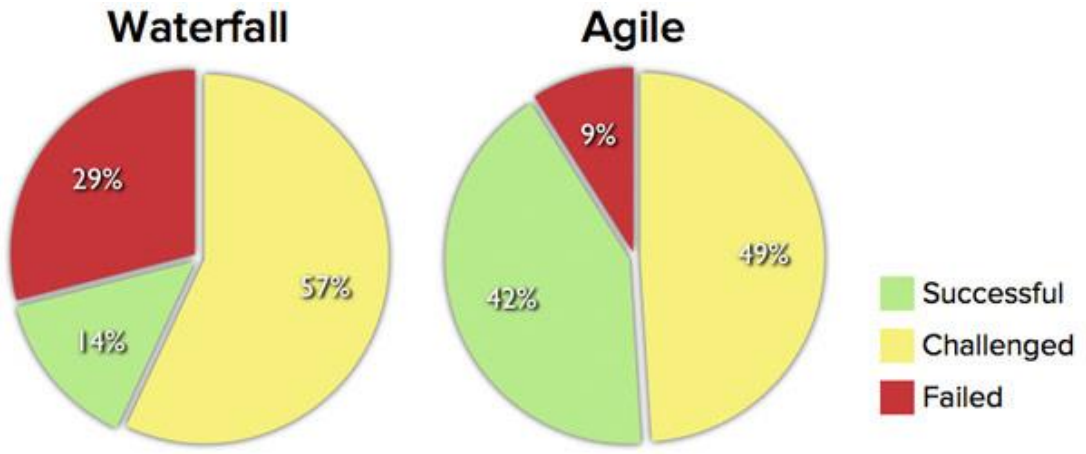
- **F**ast – sie sollen schnell durchlaufen
- **I**solated – jeder Test soll unabhängig von anderen Tests laufen (keine Ablaufreihenfolge bei Tests notwendig)
- **R**epeatable – Tests können beliebig oft ausgeführt werden
- **S**elf-validating – ein Tests soll selbst bestimmen können, ob er erfolgreich war oder fehlgeschlagen ist
- **T**imely – Tests sollen zur korrekten Zeit geschrieben werden (d.h. vor dem Produktionscode)

Ziel von Automatisierten Tests

- Schnelles Feedback
- Vermeidung von regressiven Bugs
- Fehlerlokation
- Andere Herangehensweise an die API des SUTs (Tests sind der erste Client der Produktionscodes)
- Weniger Bugfixing, weniger Zeit im Debugger

Nachteile von Automatisierten Tests

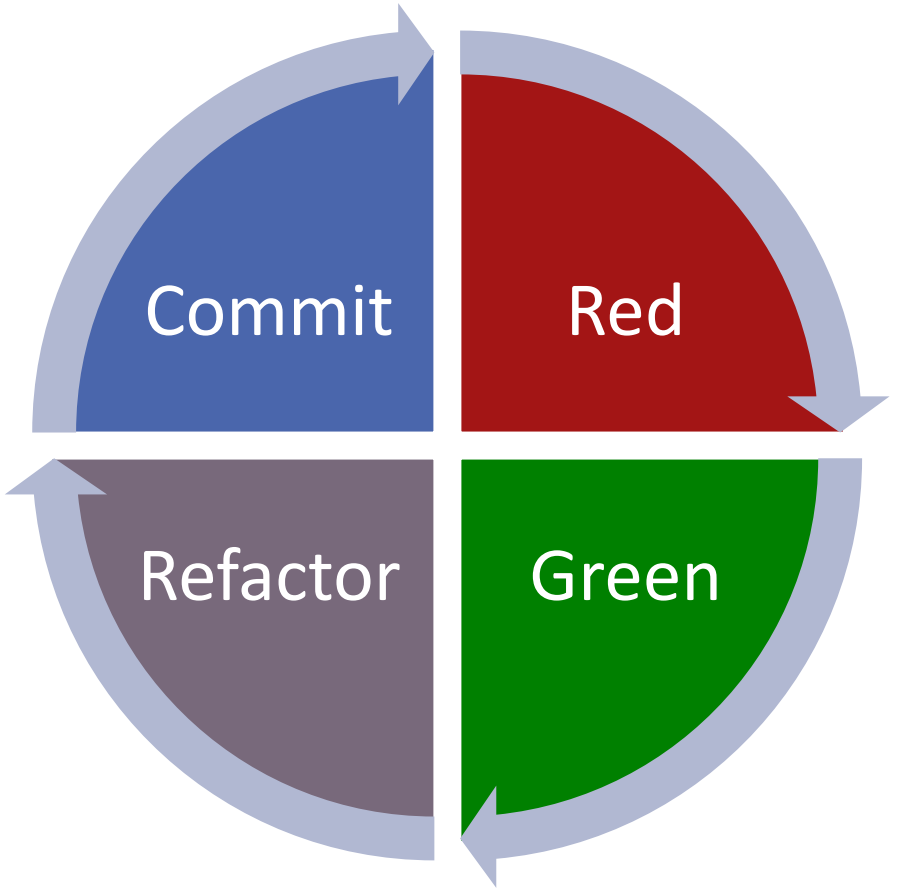
- Die Erstellung von Code dauert 50% - 100% länger.
- Diese Dauer wiegt sich auf, da deutlich weniger Zeit mit Bugfixing und im Debugger verbracht wird.



Source: The CHAOS Manifesto, The Standish Group, 2012.

Test-Driven Development

Der TDD-Zyklus

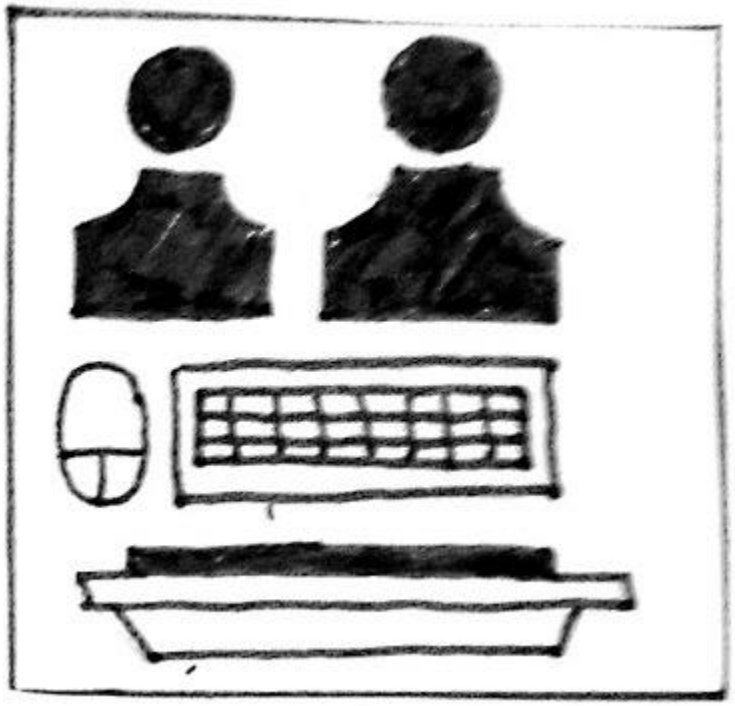


Wie starte ich mit TDD?

- Grundsätzlich sollte man einen Test zu der Funktionalität schreiben, die sich am leichtesten hinzufügen lässt.
- Anschließend erstellt man anhand der Exceptionmeldungen, die der Test hervorbringt, schrittweise die Implementierung.
- Ist die Funktionalität zu diesem Test fertiggestellt, wird überprüft, ob der Code noch weiter refactored werden kann (sowohl Test- als auch Produktionscode).
- Anschließend kann die neue Funktionalität ins Source Control System eingecheckt werden.
- Der Zyklus kann jetzt wiederholt werden, indem ein neuer Test für neue Funktionalität geschrieben werden kann.

TDD und Pair Programming

- TDD kann sehr gut mit Pair Programming umgesetzt werden.
- Ein Entwickler schreibt die Tests, der andere implementiert den Produktionscode (nach 5 – 10 kann man die Seiten wechseln).



Devil's Advocate und Gollum Style

- Bei TDD Pair Programming kann man eine Technik namens **Devil's Advocate** befolgen: **Der zweite Entwickler schreibt nur so viel Code, um die Tests erfolgreich abschließen zu können.**
- Dies entspricht nicht unbedingt der gewünschten Implementierung – der Entwickler, der die Tests verfasst, muss diese also genauer spezifizieren.
- Wenn man diese Technik alleine umsetzt, wird dies **Gollum Style** genannt.

Warum dem Test Code vertrauen?

- Test Code soll eine **Cyclomatic Complexity von 1** haben.
- Der Test soll **mindestens einmal fehlschlagen** und nach der Implementierung erfolgreich durchlaufen.

Ganz allgemein sollte der Test Code leicht lesbar sein. Pflegen Sie Ihren Test Code genauso wie Produktionscode.

Test Code als Documentation / Specification

- Wenn man innerhalb von Testmethoden Abstraktion einsetzt und diese ordentliche benennt, kann man die Lesbarkeit deutlich erhöhen.
- Dies kann zur Bildung von Domain Specific Languages (DSLs) führen – d.h. die Namen für Bezeichner decken sich mit dem Fachjargon der Domäne.
- Diese Code ist auch zu Dokumentations- oder Spezifikationszwecken geeignet.

```
[Fact]
public void A_User_Can_Join_The_Club()
{
     UsingCredentials("appName", "appKey");
     var userId = GetUserId(„mail@foo.com");
     EnsureUserIsNotAMember(userId);

     JoinClub(userId);

     Assert.IsTrue(IsAMember(userId));
}
```

Aus dem Nähkästchen

Probleme bei TDD und Automatisierten Tests

The Test Suite sucks because…

- Tests schreiben dauert zu lange.
- Der Kunde möchte nicht die Zeit für das Schreiben der Tests bezahlen.
- Wenn ich eine Sache am Test- oder Produktionscode ändere, brechen viele meiner anderen Tests und es dauert lange, bis alles wieder funktioniert.
- Meine Tests sind lang, ich kann Sie nicht ordentlich lesen und verstehen.

Problem: Stubs returning Stubs / Mocks

- Im Beispiel rechts greift man auf die `SendEmail` Funktion zu, indem man `Contact` und `Email` dereferenziert.
- In einem Test für diese Funktionalität müssen `Contact` und `Email` gestubt / gemockt werden.
- Das führt zu komplexen Testcode – man braucht mehr Zeit, um ihn zu schrieben.
- Das ist ein Signal für nicht gut designten Produktionscode.

```
public class Notifier
{
    private readonly User _user;

    public Notifier(User user)
    {
        if (user == null)
            throw new ArgumentNullException("user");
        _user = user;
    }

    public void Notify()
    {
        _user.Contact
             .Email
             .SendMail("Something happened");
    }
}
```

Problem: Mocking the SUT

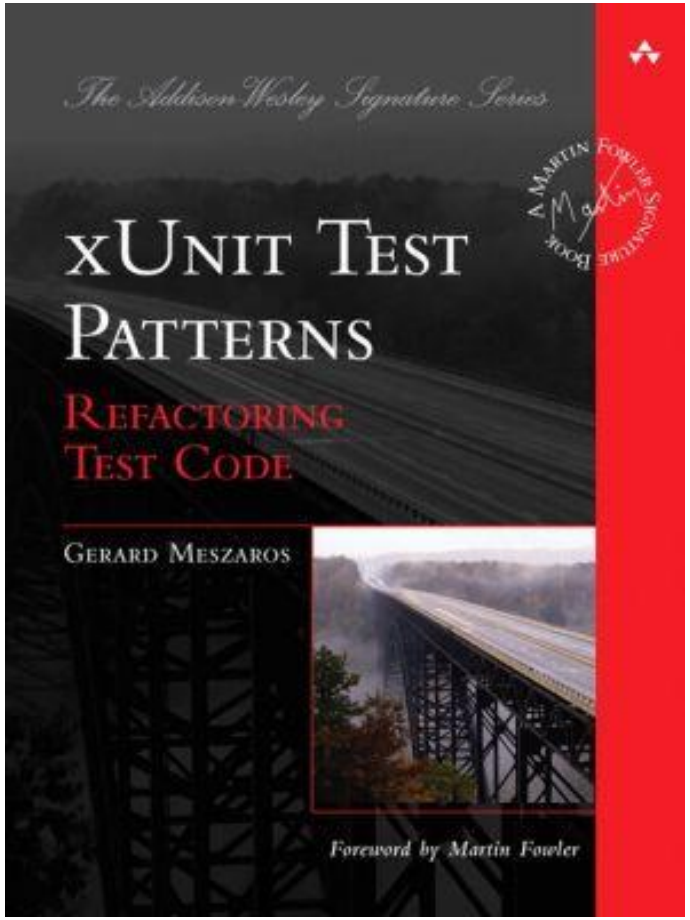
- Häufig tritt bei Anfängern die Frage auf: „Wie kann ich eine Methode auf dem SUT mocken?“
- Manche Testframeworks können das umsetzen – davon kann man aber nicht immer ausgehen.
- Dies ist ein Signal, dass das SUT zu viele unabhängige Funktionalitäten umsetzt und damit nicht dem Single Responsibility Principle folgt.

Problem: FIRST bei End-To-End und Integrationstests mit externen Systemen

- Integrationstests und insbesondere End-To-End Tests sind üblicherweise sehr umfangreich und schwierig für TDD-Anfänger.
- Ich rate Ihnen, TDD zunächst auf Unit Test Ebene zu beginnen und später auf Integrationsebene bzw. End-To-End Ebene fortzusetzen.

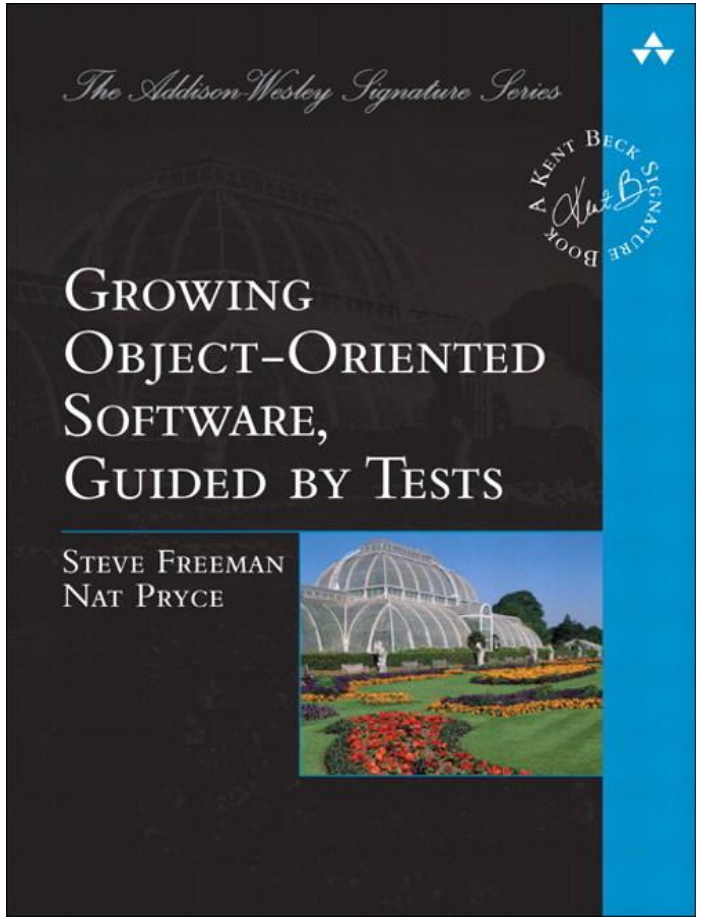
Ganz allgemein

- Da Test Code im Normalfall der erste Client des entsprechenden Produktionscodes ist, wird schlechtes Design schneller aufgedeckt (insbesondere das Dependency Inversion Principle).
- Auch Test Code sollte gepflegt werden: sehen Sie sich dazu die FIRST Principles und xUnit Test Patterns an.



Wie lerne ich, mit TDD umzugehen?

- Growing Object-Oriented Software, Guided By Tests ist ein sehr gutes, praxisorientiertes Buch.
- Im Internet sind viele Katas zu finden, mit denen man fortgeschrittene Programmiermethoden üben kann (bspw. Gilded Rose).



Vielen Dank.

Haben Sie Fragen?