

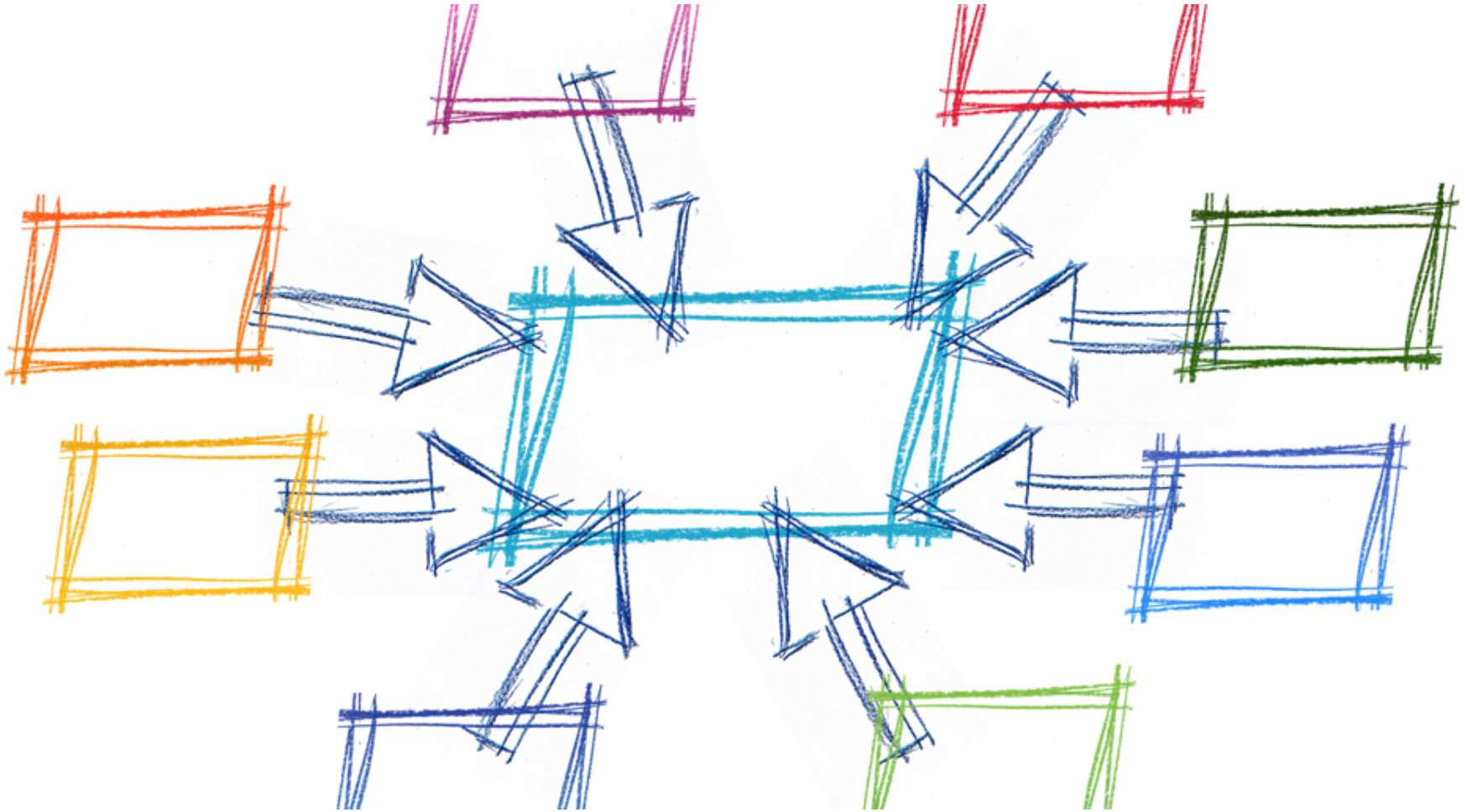
Lektionen 17 - 20



Agenda

- Abhängigkeiten auflösen
- Fake / Mock / Stub
- FakeItEasy (Isolation Framework)
- Dependency Injection
- Instruktionen zum Testat

Abhängigkeiten und Unit Tests



Bildquelle: <https://shazwazza.com/post/easily-setup-your-umbraco-installation-with-ioc-dependency-injection/>

Überprüfen... aber was?

```
public bool IsFileValid(string path)
{
    var fileAnalyzer = new FileAnalyzer();
    var isValidFormat = fileAnalyzer.CheckFormat(path);
    var isAuthorized = fileAnalyzer.CheckPermissions(path);

    return isValidFormat && isAuthorized;
}
```

Aufruf

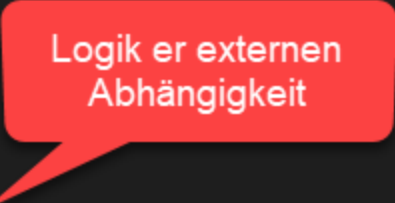
Logik

Was der Unit Test nicht prüfen soll...

```
public bool IsFileValid(string path)
{
    var fileAnalyzer = new FileAnalyzer();

    var isValidFormat = fileAnalyzer.CheckFormat(path);
    var isAuthorized = fileAnalyzer.CheckPermissions(path);

    return isValidFormat && isAuthorized;
}
```



Logik er externen Abhängigkeit

Warum?

Abhängigkeiten auflösen


- Klassen mit *externen* Abhängigkeiten können nur mit Integrationstests getestet werden.

```
public bool IsValid(string path)
{
    var fileInfo = new FileInfo(path);

    // überprüfen, ob das Format korrekt ist
    // CODE - Zur Überprüfung des Formats
    var isValidFormat = true;

    //überprüfung der Zugriffsrechte
    // CODE - Zur prüfung der Rechte
    var isAuthorized = true;

    return isValidFormat && isAuthorized;
}
```



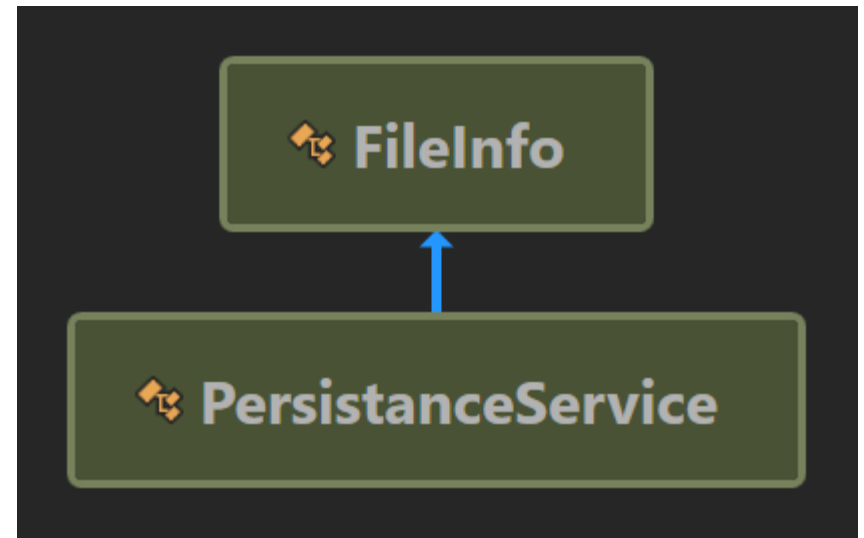
Definition «externe Abhängigkeit»

Eine *externe Abhängigkeit* ist ein Objekt in unserem System, mit dem der zu testende Code interagiert und über das wir keine Kontrolle haben. (Typische Beispiele sind das Dateisystem, Threads, Hauptspeicher, Zeit usw.)

Quelle: «The Art of Unit Testing, S. 77, 2. Auflage 2015, mitp Verlags GmbH & Co. KG»

Abhängigkeiten auflösen

Versucht man nun die Methode [IsValid] zu testen, wird auch eine externe Abhängigkeit «FileInfo» getestet. Dies «schreit» nach einem Integrationstest!



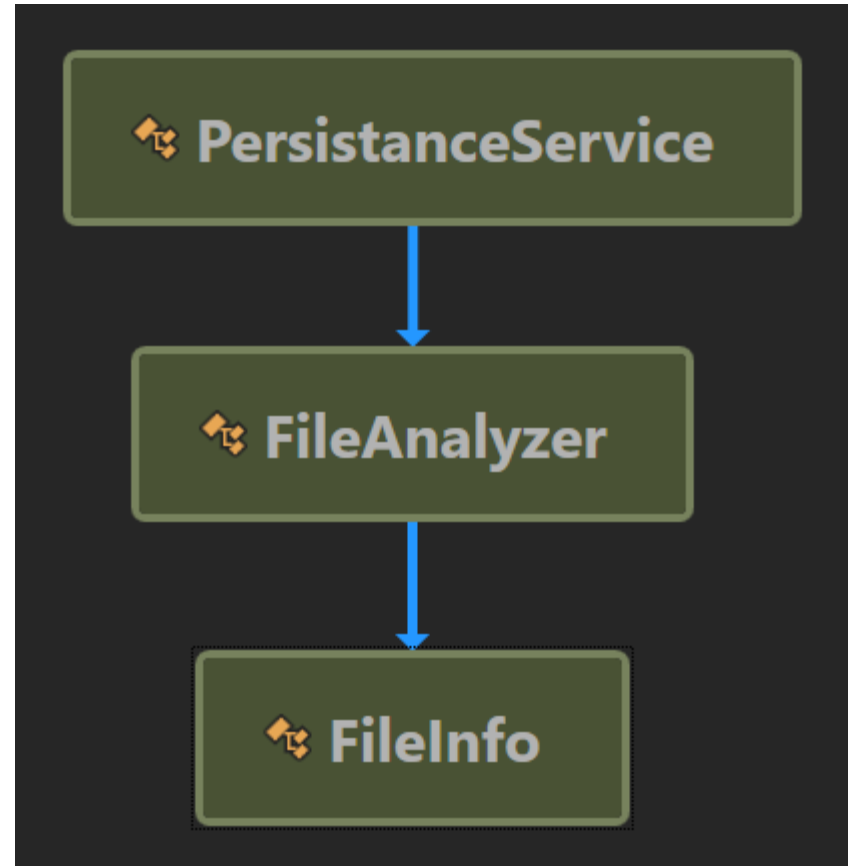
Abhängigkeiten auflösen

Das Hinzufügen einer Indirektionsschicht kann die Abhängigkeiten auflösen

Wir möchten nun für Testzwecke die [FileAnalyzer]-Klasse durch einen Fake ersetzen können, über den wir die Kontrolle haben

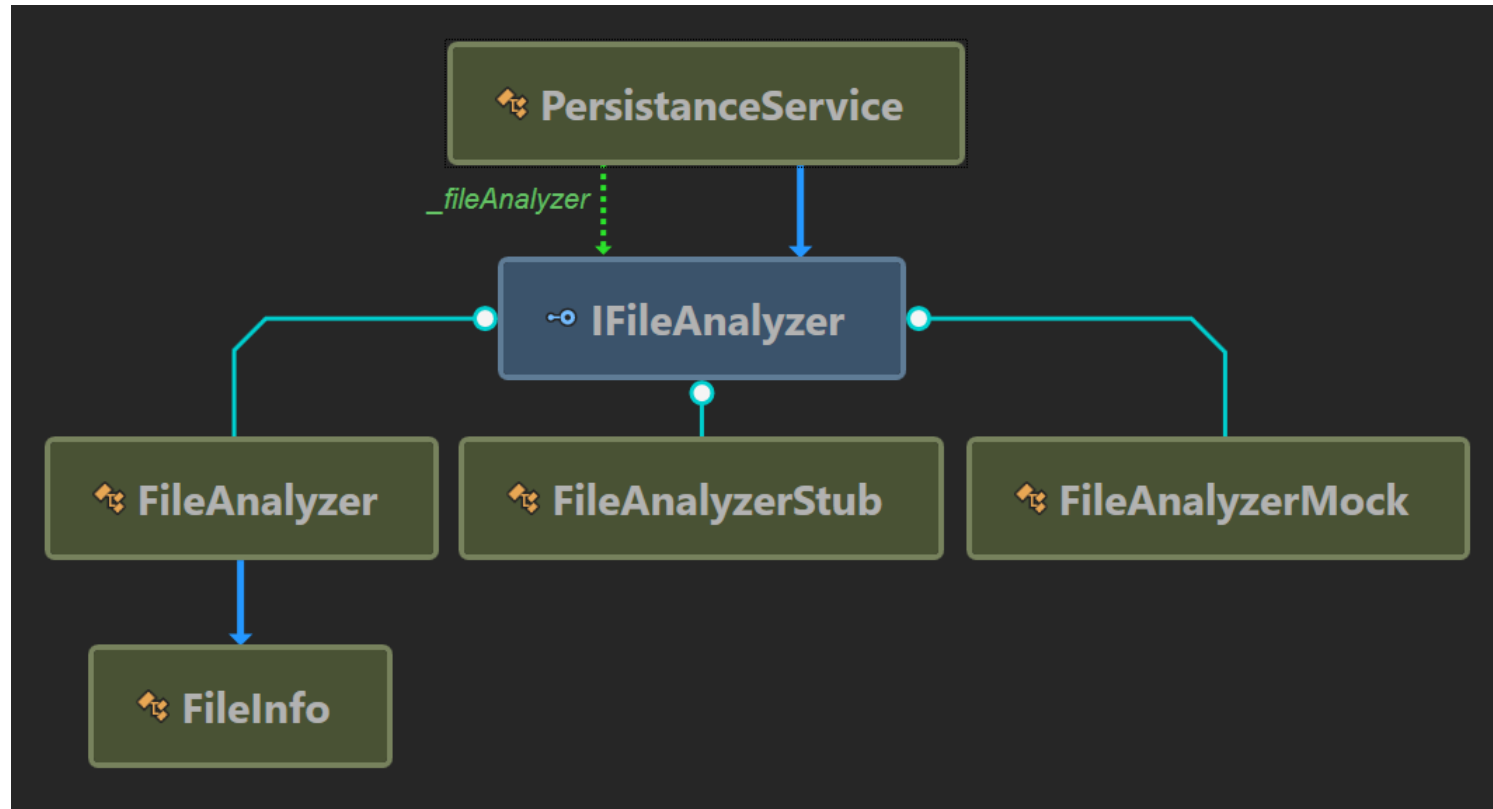
z.B.

- **StubFileAnalyzer**
- **MockFileAnalyzer**



Abhängigkeiten auflösen

Dazu verwendet man das Pattern «**Dependency Injection**»
Die Klasse [FileAnalyzer] verbirgt die Operationen zu der externen Abhängigkeit.





Bildquelle: <https://hu.depositphotos.com/102275398/stock-illustration-cartoon-chipmunk-holding-peanut-on.html>

Definition «Stub»

Ein *Stub*(*Stummel*) ist ein kontrollierbarer Ersatz für eine vorhandene Abhängigkeit (ein Collaborator) im System. Durch die Verwendung eines Stubs kann der Code getestet werden, ohne die Abhängigkeit direkt handhaben zu müssen.

Quelle: «The Art of Unit Testing, S. 78, 2. Auflage 2015, mitp Verlags GmbH & Co. KG»

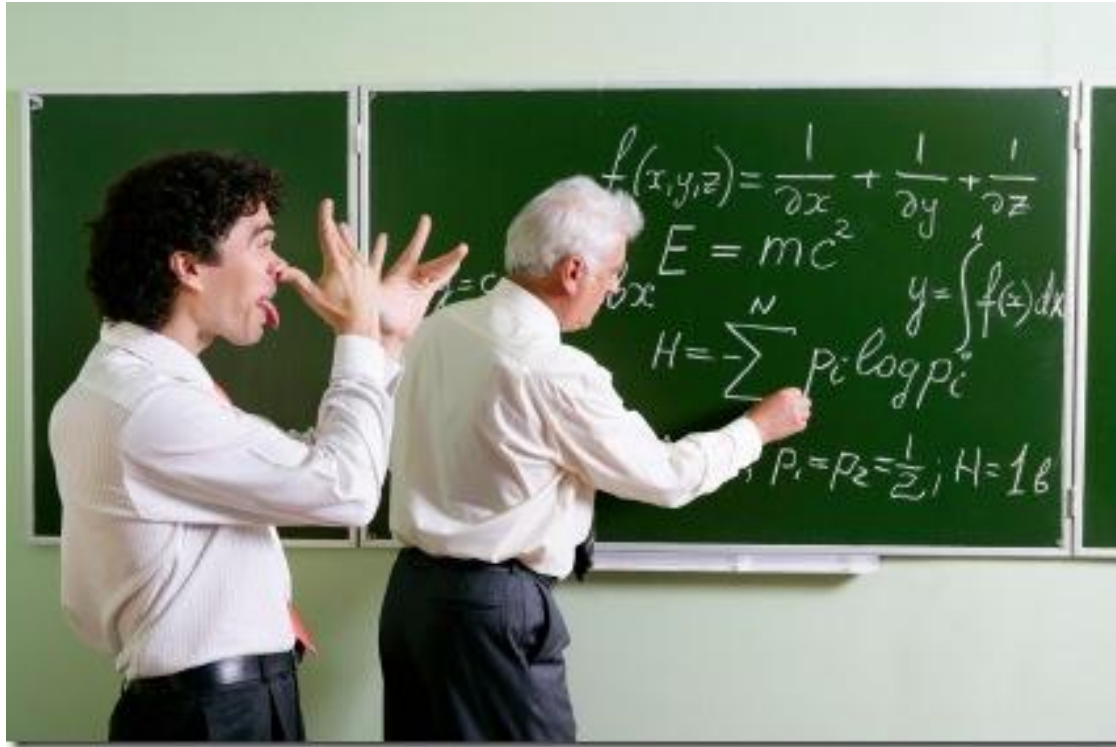
Stub-Beispiel

```
public FileAnalyzerStub(bool isValidFormat, bool isAuthorized)
{
    _isValidFormat = isValidFormat;
    _isAuthorized = isAuthorized;
}

4 references
public bool CheckFormat(string path)
{
    return _isValidFormat;
}

4 references
public bool CheckPermissions(string path)
{
    return _isAuthorized;
}
```

Mock



Bildquelle: <https://www.hanselman.com/blog/MoqLinqLambdasAndPredicatesAppliedToMockObjects.aspx>

Definition «Mock»

Ein *Mock(Nachahmung)-Objekt* ist ein nachgeahmtes Objekt im System, das entscheidet, ob ein Unit Test funktioniert hat oder fehlgeschlagen ist. Es macht dies, indem es verifiziert, ob das zu testende Objekt das Fake-Objekt wie erwartet aufgerufen hat. Gewöhnlich gibt es nicht mehr als einen Mock pro Test.

Quelle: «The Art of Unit Testing, S. 109, 2. Auflage 2015, mitp Verlags GmbH & Co. KG»

Mock-Beispiel

```
public bool CheckFormatCalled { get; private set; }

2 references
public bool CheckPermissionsCalled { get; private set; }

4 references
public bool CheckFormat(string path)
{
    CheckFormatCalled = true;
    return true;
}

4 references
public bool CheckPermissions(string path)
{
    CheckPermissionsCalled = true;
    return true;
}
```


Fakes



Bildquelle: <http://blog.wolfmillionaire.com/beware-fake-instagram-accounts/>

Definition «Fake»

Ein *Fake*(*Fälschung*) ist ein allgemeiner Begriff, der benutzt werden kann, um entweder ein Stub- oder Mock-Objekt (handgeschrieben oder nicht) zu bezeichnen, denn beide sehen aus wie das echte Objekt. Ob es sich bei dem Fake um einen Stub oder einen Mock handelt, hängt davon ab, wie er im aktuellen Test verwendet wird. Wenn er dazu benutzt wird, eine Interaktion zu überprüfen, ist er ein *Mock-Objekt*. Anderenfalls handelt es sich um einen *Stub*.

Quelle: «The Art of Unit Testing, S. 109, 2. Auflage 2015, mitp Verlags GmbH & Co. KG»

Stub vs. Mock



Bei einem Stub wird mit dem Assert eines Unit Tests nur das SUT überprüft. D.h. ein Stub dient nur dazu, dass das SUT korrekt auszuführen werden kann, es muss nicht direkt mit dieser Abhängigkeit interagiert werden.

Stub vs. Mock



Bei einem Mock wird mit dem Assert eines Unit Tests überprüft, ob das SUT mit diesem korrekt interagiert. D.h. ein Mock dient z.B. dazu, zu überprüfen, ob die Methoden auf der Abhängigkeit korrekt aufgerufen wurden.

Was hat ein Saum mit einem Unit Test zu tun?



Bildquelle: https://www.eileenfisher.com/repair-and-care/how-to-spot-quality-the-anatomy-of-a-seam/?__store=en&__from_store=default

Definition «Seam»

Seams (Naht/Saum) sind Stellen in Ihrem Code, in die Sie eine andere Funktionalität einfügen können, wie etwa Stub-Klassen. Sie könnten auch einen Konstruktor-Parameter hinzufügen, eine Property mit öffentlichem set, Sie könnten eine Methode virtuell machen, damit sie überschrieben werden kann, oder einen Delegaten als Parameter oder Property von aussen verfügbar machen, damit von ausserhalb der Klasse auf ihn zugegriffen werden kann...

Quelle: «The Art of Unit Testing, S. 82, 2. Auflage 2015, mitp Verlags GmbH & Co. KG»


Seam-Beispiel

```
public class PersistenceService
{
    private readonly IFileAnalyzer _fileAnalyzer;

    4 references
    public PersistenceService(IFileAnalyzer fileAnalyzer)
    {
        _fileAnalyzer = fileAnalyzer;
    }

    4 references
    public bool IsFileValid(string path)
    {
        var isValidFormat = _fileAnalyzer.CheckFormat(path);
        var isAuthorized = _fileAnalyzer.CheckPermissions(path);

        return isValidFormat && isAuthorized;
    }
}
```



A red speech bubble with the word "Seam" inside points to the `IFileAnalyzer fileAnalyzer` parameter in the constructor of the `PersistenceService` class.

Abhängigkeiten auflösen

Es gibt drei Arten wie man Seams einbauen kann:

- *Injektion eines Fakes auf Konstruktor-Ebene*
- *Injektion eines Fakes per Setter/Getter*
- *Injektion eines Fakes unmittelbar vor einem Methodenaufruf*

Nach «The Art of Unit Testing, S. 83, 2. Auflage 2015, mitp Verlags GmbH & Co. KG»

Injektion eines Fakes auf Konstruktor-Ebene 1/2

```
// #####
// # Injektion eines Fakes auf Konstruktor-Ebene
// #####

// Lösung:
public class FileAccessManager implements AccessManager {
    public boolean isValid(String path) {
        // überprüfe ob die Datei existiert
    }
}

public class AlwaysValidFakeFileAccessManager implements AccessManager {
    public boolean isValid(String path) {
        return true;
    }
}

public interface AccessManager {
    boolean isValid(String path);
}

public class PersistenceService {
    private AccessManager accessManager;

    public PersistenceService(AccessManager accessManager) {
        this.accessManager = accessManager;
    }

    public boolean isFileNameValid(String path) {
        boolean isValid = this.accessManager.isValid(path);
        return isValid;
    }
}
```

Injektion eines Fakes auf Konstruktor-Ebene 2/2

```
// Lösung (Test):
public class PersistenceServiceTest {

    @Test
    public void isFileNameValid_WithStub_ReturnsTrue() {
        // arrange
        AlwaysValidFakeFileManager fake = new AlwaysValidFakeFileManager();
        PersistenceService sut = new PersistenceService(fake);

        // act
        boolean result = sut.isFileNameValid("FakePath");

        // assert
        assertThat(result, is(true));
    }
}
```

Injektion eines Fakes per Setter/Getter 1/2

```
// #####  
// # Injektion eines Fakes per Setter/Getter  
// #####  
  
// Lösung (Implementierung):  
public class PersistenceService {  
    private AccessManager accessManager;  
  
    public PersistenceService() {  
        this.accessManager = new FileAccessManager();  
    }  
  
    public AccessManager getAccessManager() {  
        return this.accessManager;  
    }  
  
    public void setAccessManager(AccessManager accessManager) {  
        this.accessManager = accessManager;  
    }  
  
    public boolean isFileNameValid(String path) {  
        boolean isValid = this.accessManager.isValid(path);  
        return isValid;  
    }  
}
```

Injektion eines Fakes per Setter/Getter 2/2

```
// Lösung (Test):
public class PersistenceServiceTest {

    @Test
    public void isFileNameValid_WithStub_ReturnsTrue() {
        // arrange
        PersistenceService sut = new PersistenceService();

        AlwaysValidFakeFileManager fake = new AlwaysValidFakeFileManager();
        sut.setAccessManager(fake);

        // act
        boolean result = sut.isFileNameValid("FakePath");

        // assert
        assertThat(result, is(true));
    }
}
```

Injektion eines Fakes vor einem Methodenaufruf 1/2

```
// #####
// # Injektion eines Fakes vor einem Methodenaufruf
// #####

// Lösung (Implementierung):
public final class AccessManagerFactory {
    private static AccessManager customAccessManager;

    public static void setCustomAccessManager(AccessManager accessManager) {
        customAccessManager = accessManager;
    }

    public static AccessManager create() {
        if(customAccessManager != null) {
            return customAccessManager;
        }

        return new FileAccessManager();
    }
}

public class PersistenceService {
    public boolean isFileNameValid(String path) {
        AccessManager accessManager = AccessManagerFactory.create();
        boolean isValid = accessManager.isValid(path);
        return isValid;
    }
}
```

Injektion eines Fakes vor einem Methodenaufruf 2/2

```
// Lösung (Test):
public class PersistenceServiceTest {

    @Test
    public void isFileNameValid_WithStub_ReturnsTrue() {
        // arrange
        PersistenceService sut = new PersistenceService();

        AlwaysValidFakeFileManager fake = new AlwaysValidFakeFileManager();
        AccessManagerFactory.setCustomAccessManager(fake);

        // act
        boolean result = sut.isFileNameValid("FakePath");

        // assert
        assertThat(result, is(true));
    }
}
```

Übung – Abhängigkeiten auflösen

Laden Sie sich den Source Code herunter und versuchen Sie die einzelnen Schritte nachzuvollziehen. <https://github.com/michikeiser/ZbW.Testing.Dependencies>

Vorgehen:

1. Problem
2. Indirektionsschicht
3. Aufgelöst

WICHTIG: Die Methodik müssen Sie verstanden haben, sonst können wir nicht weitermachen.

Stellen Sie Fragen wo nötig.

Arbeitsform: Einzelarbeit
Zeit (Vorbereitung): 15 Minuten
Besprechung / Feedback: In der Klasse

Fragen?



Bildquelle: <https://www.wt-hahn.de/fragen-antworten/>

Übung – Abhängigkeiten auflösen

- Öffnen Sie die Solution unter «4_Aufgabe». Refactorn Sie den Code, damit Sie alle nötigen Unit Tests erstellen können.

Arbeitsform: Einzelarbeit
Zeit (Vorbereitung): 25 Minuten
Besprechung / Feedback: In der Klasse

Probleme mit handgeschriebenen Mocks und Stubs

- Es braucht Zeit, um die Mocks und Stubs zu schreiben.
- Es ist schwierig, Stubs und Mocks für Klassen und Interfaces zu schreiben, die viele Methoden haben.
- Um den Zustand für mehrfache Aufrufe einer Mock-Methode zu speichern, müssen Sie eine Menge Standardcode innerhalb der manuellen Fakes schreiben.
- Wenn Sie verifizieren wollen, dass alle Parameter vom Aufrufer korrekt an eine Methode übergeben wurden, müssen Sie zahlreiche Asserts schreiben. Das ist ein Langweiler.
- Es ist schwierig, den Code von Mocks und Stubs für andere Tests wiederzuverwenden.
- Der grundlegende Krempel funktioniert, aber sobald Sie es mit mehr als zwei oder drei Funktionen im Interface zu tun haben, fängt es an, mühsam zu werden.

Quelle: «The Art of Unit Testing, S. 121, 2. Auflage 2015, mitp Verlags GmbH & Co. KG»

Isolation Framework (FakeItEasy)

- Webseite (<https://fakeiteasy.github.io/>)
- Doku (<https://fakeiteasy.readthedocs.io/en/stable/>)
- Nuget Package (FakeItEasy)



Fakes erstellen

- Bei FakeItEasy geschieht alles über die Klasse «A»
- Erstellung eines Fakes aus einem Interface
 - `var foo = A.Fake<IFoo>();`

Was kann alles gefaked werden?

- Interfaces
- Klassen die...
 - nicht «**sealed**» sind
 - nicht «**statisch**» sind
 - Die keinen «**private Konstruktor**» aufweisen
- Delegate

Welche Members können überschrieben werden?

- Wird ein Fake als Stub verwendet, müssen die Properties/Methoden überschrieben werden können. Damit dies gemacht werden kann, muss eine der folgenden Bedingungen erfüllt sein:
 - Virtual
 - Abstract
 - Interface Methode

Standardverhalten eines Fakes

- Alle überschreibbaren Properties/Methoden werden so überschrieben, dass sie einen leeren Methodenrumpf aufweisen.
- Methoden, welche etwas zurückgeben müssen geben den Wert von «default(T)» zurück.

Dies gilt nur, wenn nichts anderes definiert wurde.

- Strict Fakes werden auch unterstützt
 - `var foo = A.Fake<IFoo>(x => x.Strict());`

Stub - Aufruf parametrisieren

- Methoden

- `A.CallTo(() => fakeShop.GetTopSellingCandy()).Returns(lollipop);`
- `A.CallTo(() => fakeShop.AddNumber(DateTime.MaxValue)) .Throws(new InvalidDateException("the date is in the future"));`

- Properties

- `A.CallTo(() => fakeShop.Address).Returns("123 Fake Street");`

- Events

- `robot.FellInLove += Raise.With(someEventArgs);`

Mock – Aufruf parametrisieren

- Methoden

- `A.CallTo(() => foo.Bar()).MustHaveHappened();`
- `A.CallTo(() => foo.Bar()).MustNotHaveHappened();`
- `A.CallTo(() => foo.Bar()).MustHaveHappenedOnceExactly();`
- `A.CallTo(() => foo.Bar()).MustHaveHappenedOnceOrMore();`
- `A.CallTo(() => foo.Bar()).MustHaveHappenedOnceOrLess();`
- `A.CallTo(() => foo.Bar()).MustHaveHappenedTwiceExactly();`
- `A.CallTo(() => foo.Bar()).MustHaveHappenedTwiceOrMore();`
- `A.CallTo(() => foo.Bar()).MustHaveHappenedTwiceOrLess();`
- `A.CallTo(() => foo.Bar()).MustHaveHappened(4, Times.Exactly);`
- `A.CallTo(() => foo.Bar()).MustHaveHappened(6, Times.OrMore);`
- `A.CallTo(() => foo.Bar()).MustHaveHappened(7, Times.OrLess);`

Analyzer

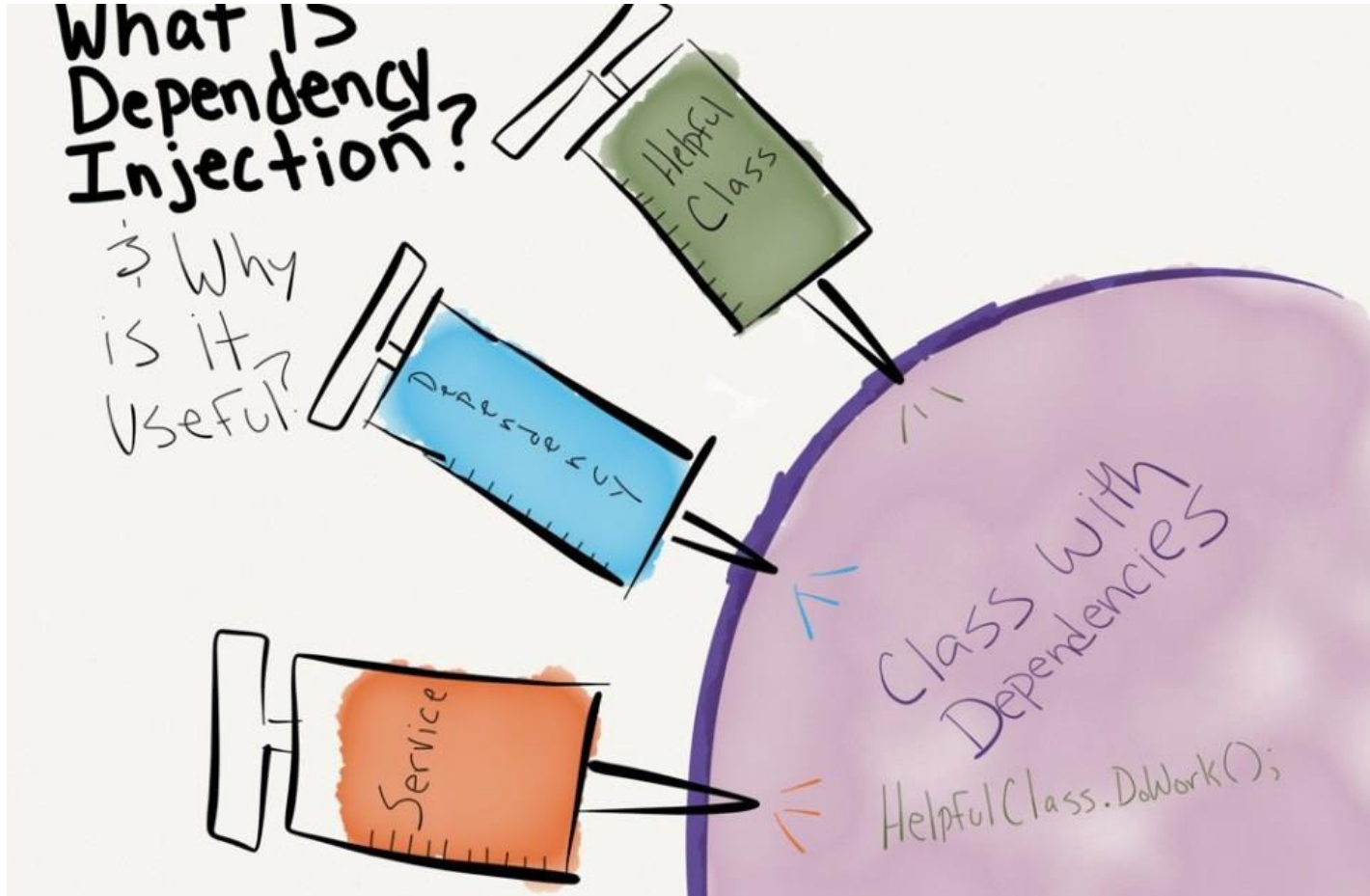
- FakeItEasy bietet einen Roslyn Analyser an, welcher die nicht korrekte Verwendung von FakeItEasy zur Compilezeit anzeigen kann.
- Dazu muss das Nuget-Package «FakeItEasy.Analyzer» installiert werden

Übung – Isolation Framework

- Laden Sie sich den Code herunter und ersetzen Sie die manuell geschriebenen Fakes mit FakeItEasy.
- <https://github.com/michikeiser/ZbW.Testing.Isolation>

Arbeitsform: Einzelarbeit
Zeit (Vorbereitung): 25 Minuten
Besprechung / Feedback: In der Klasse

Dependency Injection



Bildquelle: <http://www.ada-nedu.com/the-pretty-thing-called-dependency-injection/>

Dependency Injection – Metapher

Ausgangslage (Code ohne DI)



Das Problem ist nun, dass wir zwischen dem [Lehrling] und Der [Schaufel] eine Beziehung haben.

```
1 reference
public class Lehrling
{
    private Schaufel meineSchaufel;

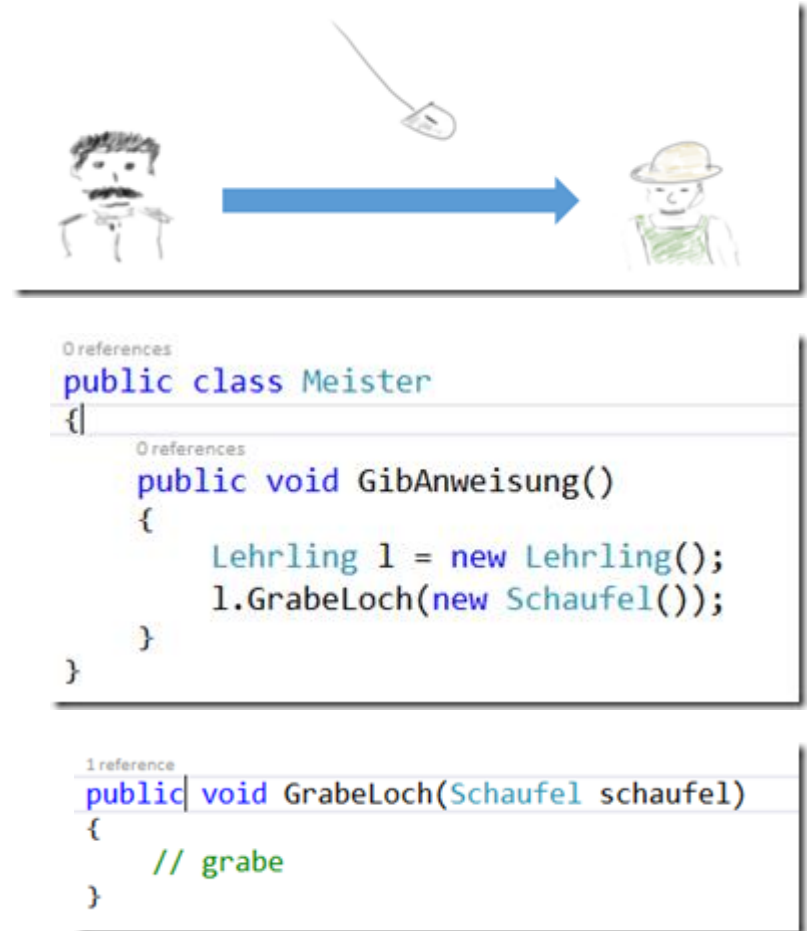
    0 references
    public Lehrling()
    {
        meineSchaufel = new Schaufel();
    }

    0 references
    public void GrabeLoch()
    {
        // graben
    }
}
```

Quelle: <https://blogs.msdn.microsoft.com/dmx/2014/10/14/was-ist-eigentlich-dependency-injection-di/>

Dependency Injection – Metapher

Besser wäre, wenn die [Schaufel] von aussen übergeben werden könnte.



Quelle: <https://blogs.msdn.microsoft.com/dmx/2014/10/14/was-ist-eigentlich-dependency-injection-di/>

Dependency Injection – Metapher

Besser wäre, wenn der [Lehrling]
Mit jedem Werkzeug arbeiten könnte.



```
1 reference
interface IGrabable
{
    1 reference
    void Buddel();
}
```

```
public class Schaufel : IGrabable
{
    3 references
    public void Buddel()
    {
    }
}
```

```
public class Lehrling
{
    1 reference
    public void GrabeLoch(IGrabable grabewerkzeug)
    {
        // grabe
        grabewerkzeug.Buddel();
    }
}
```

Quelle: <https://blogs.msdn.microsoft.com/dmx/2014/10/14/was-ist-eigentlich-dependency-injection-di/>

Dependency Injection – Metapher

Der [Meister] kann nun frei entscheiden mit welchem Werkzeug der [Lehrling] arbeiten soll.



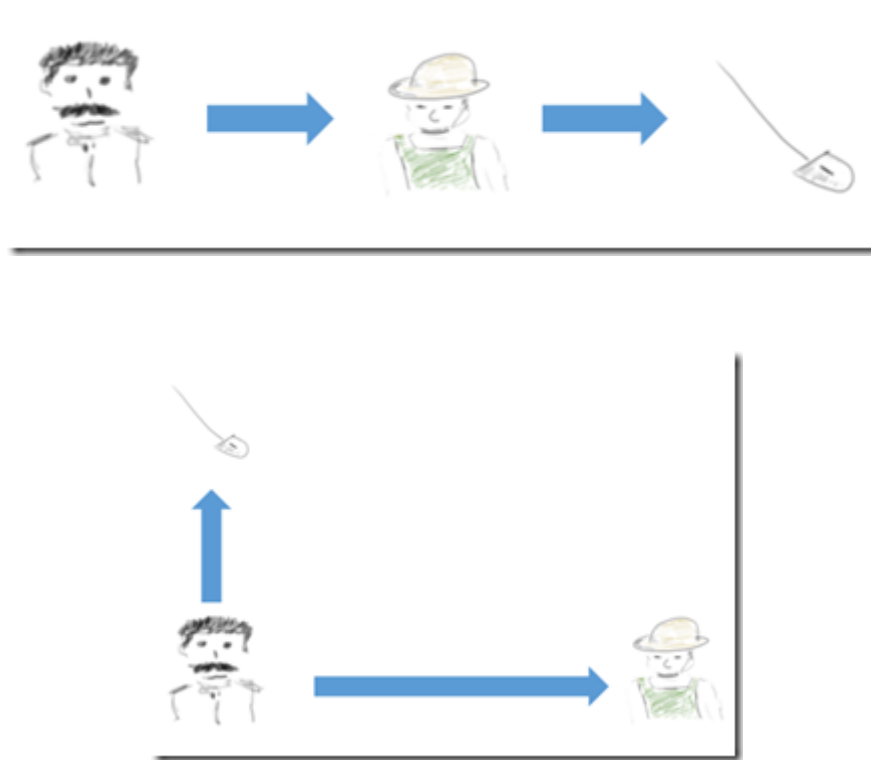
```
public class Meister
{
    public void GibAnweisung()
    {
        Lehrling l = new Lehrling();

        Spaten spaten = new Spaten();
        Schaufel schaufel = new Schaufel();
        l.GrabeLoch(spaten);
        l.GrabeLoch(schaufel);
    }
}
```

Quelle: <https://blogs.msdn.microsoft.com/dmx/2014/10/14/was-ist-eigentlich-dependency-injection-di/>

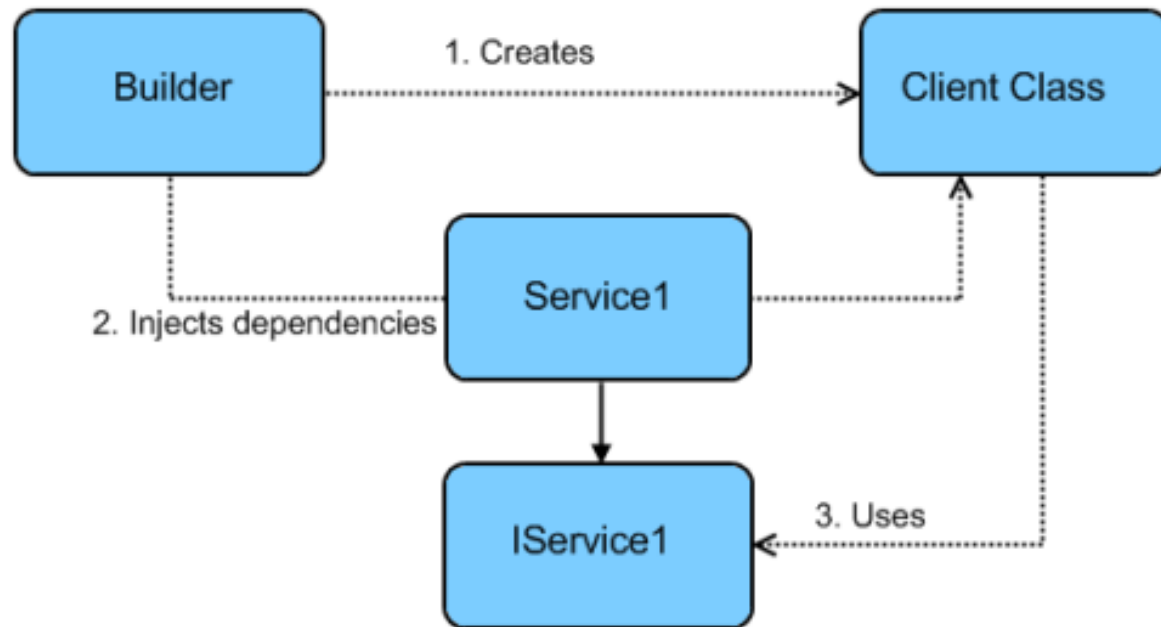
Dependency Injection – Metapher

Die Abhängigkeit wurde nun geändert:



Quelle: <https://blogs.msdn.microsoft.com/dmx/2014/10/14/was-ist-eigentlich-dependency-injection-di/>

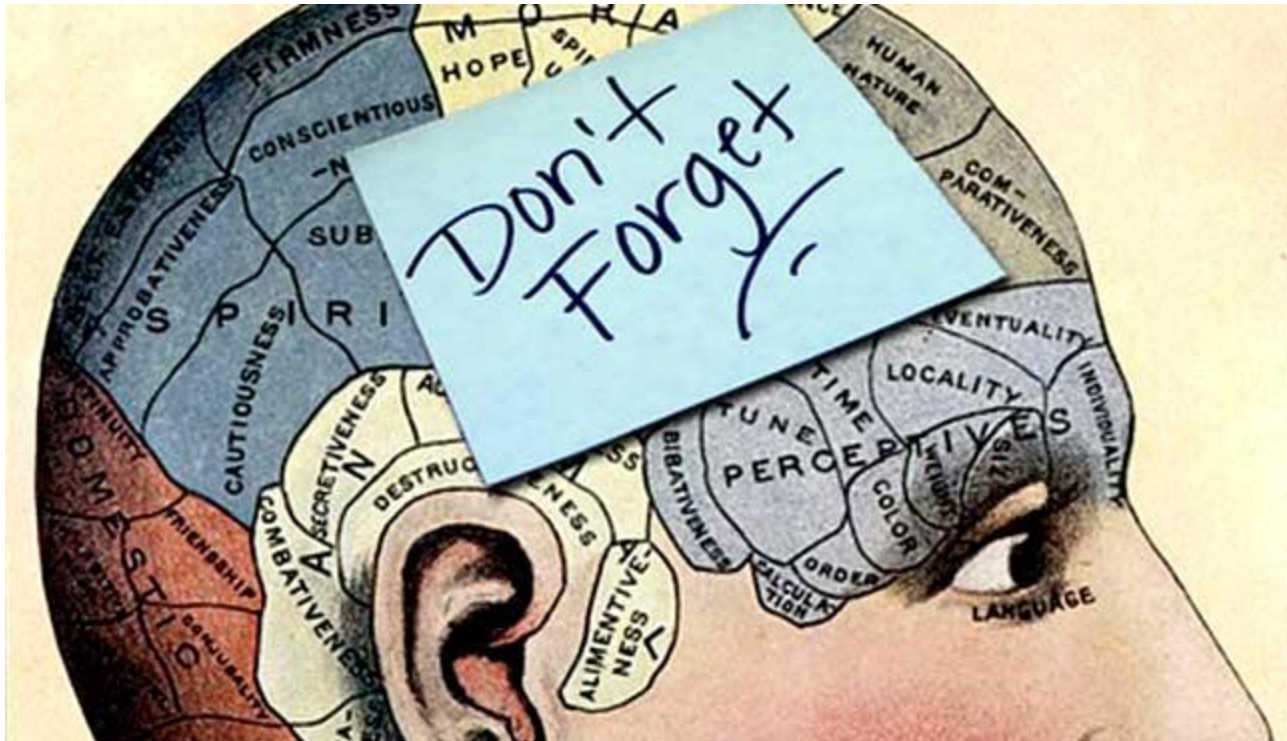
Dependency Injection - Pattern



Testat – Dokumenten Management System

- Erstellen Sie einen Fork von folgendem Repository
 - <https://github.com/michikeiser/ZbW.Testing.Dms>
- Lösen Sie alle User Stories gemäss «Aufgaben.docx»
- Erstellen Sie alle nötigen Unit- sowie Integration Tests bis spätestens Sonntag, 23.09.2018
- Geben Sie unter michi.keiser@gmail.com ihr Benutzernamen bekannt.

Was nehmen wir vom Unterricht mit?



Bildquelle: <http://floraremedia.com/keep-mind-memory-sharp/>

Hausaufgaben

- Unterricht nachbearbeiten