# WeatherWallpaper

Janis Fix, Leon Gieringer

TINF18B3

Advanced Software Engineering

25. Mai 2021

# Inhaltsverzeichnis

**1 Einleitung** **1**

**2 Clean Architecture** **2**
2.1 Vorher . . . 3
2.2 Nachher . . . 3
2.2.1 Domain Code . . . 3
2.2.2 Application Code . . . 3
2.2.3 Adapters . . . 3
2.2.4 Plugins . . . 4
2.2.5 Mögliche Erweiterungen/Herausforderungen . . . 4

**3 Entwurfsmuster** **5**

**4 Programming Principles** **6**
4.1 SOLID . . . 6
4.1.1 Single Responsibility Principle . . . 6
4.1.2 Open/Closed Principle . . . 7
4.1.3 Liskov Substitution Principle . . . 8
4.1.4 Interface Segregation Principle . . . 8
4.1.5 Dependency Inversion Principle . . . 8
4.2 GRASP . . . 8
4.3 DRY . . . 8

**5 Refactoring** **9**

**6 Unit Tests** **10**
6.1 ATRIP . . . 10
6.2 Beispiel für Unit-Tests und Mocks . . . 11
6.3 Code Coverage . . . 12

# Listings

4.1 Verletzung des Open/Closed-Principle im ConfigValidator . . . . . 7
4.2 Entwicklung mit Erweiterung für den ConfigValidator . . . . . 8
6.1 Unit-Test für den ConfigValidator . . . . . 11
6.2 Unit-Test für den ImageHandler mit Mock . . . . . 12

# 1 Einleitung

Hier steht meine Einleitung

## 2 Clean Architecture

Softwareprodukte entwickeln sich immer weiter, was heute noch State of the art ist, kann in ein paar Jahren schon wieder durch eine neue Technologie ersetzt worden sein. Deshalb ist es wichtig seine Anwendung so gut wie möglich für Technologie-Änderungen von außen vorzubereiten. Um dies zu ermöglichen, muss in bei den Architekturentscheidung acht gegeben werden. Dabei sollte man sich nicht an äußere Abhängigkeiten binden, sondern diese austauschbar machen. Dabei teilt sich der Quellcode einer Anwendung in mindestens zwei Schichten ein. Der langfristig bestehende Quellcode der Anwendung, sowie der kurzlebige Quellcode der äußeren Abhängigkeiten. Zu diesen äußeren Abhängigkeiten kann beispielsweise eine API gehören. Dieser Schichtenaufbau ist vergleichbar mit einer Matrjoschka oder einer Zwiebel.

Wie bei einer Matrjoschka/Zwiebel auch, muss die Dependency Rule erfüllt sein, damit die Schichten klar aufgeteilt sind und äußere Schichten (relativ) einfach ausgetauscht werden können. Die Dependecy Rule sagt aus, dass Abhängigkeiten immer nur von außen nach innen gehen dürfen. Wenn ein äußerer Zwiebelring ausgetauscht wird, soll dies keine Änderung bzw. Anpassung an einem weiter innen liegenden Zwiebelring bewirken.

Eine Applikation lässt sich in fünf Schichten einteilen, das sind (von innen nach außen):

- Der Abstraction Code
  - Dieser beinhaltet Code, der sowohl für die eigene Problemdomäne, als auch andere Problemdomänen wichtig sein kann. Hierzu zählen beispielsweise mathematische Grundlagen, wie Vektoren o.Ä.
- Der Domain Code
  - Diese Schicht beinhaltet hauptsächlich Entitäten und sollte sich am wenigsten ändern.
- Der Application Code
  - Im Application Code sind die einzelnen Use Cases wieder zu finden und implementiert damit die Geschäftslogik der Anwendung. Hier werden also die einzelnen Use Cases umgesetzt.
- Die Adapters
  - Diese Schicht handelt, wie der Name schon sagt, als Adapter zwischen den äußeren Plugins und den inneren Schichten. Dabei kann beispielsweise eine Formatkonvertierung stattfinden. Ein Beispiel hierfür wäre eine Web-API, die in der Plugin-Schicht angeordnet ist und ein JSON-String zurückliefert. Der Adapter ist dann für die Konvertierung des JSON-Objekts zu dem Format der Anwendung (z.B. C#-Objekt) zuständig. Ziel der Adapter ist es, die inneren und äußeren Schichten zu entkoppeln.
- Die Plugins
  - Diese Schicht darf keine Anwendungslogik enthalten, da die Plugins jederzeit änderbar sein müssen. Hier steht quasi nur Pure Fabrication Code. Wird beispielsweise das World-Wide-Web durch eine besondere neue Technologie ersetzt, müssen die Web-APIs ausgetauscht werden. Dies sollte sich nicht auf die anwendungsspezifische Geschäftslogik auswirken.

Die Applikation wird in einer vier Schichtenarchitektur umgesetzt. Dabei wird auf die erste Schicht, den Abstraction Code, verzichtet.

## 2.1 Vorher

Ein UML-Diagramm mit der Situation vor dem Implementieren der Clean Architecture ist auf unserem Git-Repository hier zu finden. Abhängigkeiten von innen nach außen, die die Dependecy Rule brechen, sind dick und rot hinterlegt. Auf die Abhängigkeitspfeile in die innerste Schicht wurde für die Leserlichkeit verzichtet.

## 2.2 Nachher

Neben der Implementierung der Clean Architecture wurden natürlich auch noch andere Änderungen beispielsweise für die Programming Principles umgesetzt. Daher haben sich manche Klassen aufgeteilt. Das aktuelle Diagramm der Anwendung findet man hier. Hierbei ist erkennbar, dass nun keine Abhängigkeitspfeile von einer inneren zu einer äußeren Schicht gehen und diese mithilfe der Dependency Inversion umgedreht wurden. Damit ist die Depedency Rule erfüllt. Im Folgenden wird nochmals genauer auf die einzelnen Schichten eingegangen.

### 2.2.1 Domain Code

In dieser Schicht reihen sich die Entitäten der Anwendung ein. Dazu gehört die `Config`, in der das Zeitintervall und der Standort gespeichert ist, sowie die `Weather-` und `ImageResponse`, welche die für die Anwendung nötigen Daten der APIs wiedergeben. In der Klasse `CountryArrays` sind Länder und ihre Abkürzungen hinterlegt. Die `WeatherInterpretation` wird für die Interpretierung des Wetters verwendet und die `BadConfigException` ist ein eigener Exception-Typ, der angibt, dass die Konfiguation fehlerhaft ist. Im UML-Diagramm wird erneut auf die Abhängigkeitspfeile in die innerste Schicht verzichtet, damit es lesbarer bleibt.

### 2.2.2 Application Code

Die zentrale Klasse der Anwendung und des Application Codes ist der `Refresher`. Mithilfe des Refreshers lässt sich, über Delegation an Helferklassen, das Desktophintergrundbild an die aktuelle Wettersituation anpassen. Dafür muss beispielsweise das aktuelle Wetter interpretiert werden, so dass man eine deskriptive Darstellung des Wetters zum Abfragen der Bild-API hat. Dies wird von der `WeatherInterpreter`-Klasse erledigt. Ein weiterer Use-Case ist die Validierung der vom Nutzer eingegebenen Konfiguration. Darum kümmert sich der `ConfigValidator`. Dafür arbeitet sie mit den `IValidationAspects` zusammen. Näheres dazu ist bei dem OCP beschrieben. Die Klassen `ScreenChangeWorker` und `UpdateTimer` werden für das aktualisieren mithilfe des Refreshers verwendet. Ersterer wird verwendet, um ein manuelles Aktualisieren in einem neuen Thread zu gewährleisten, damit die GUI nicht blockiert wird. Zweiterer ist für das zyklische Aktualisieren des Hintergrunds zuständig. In dieser Schicht sind ebenfalls die benötigten Interfaces für den Zugriff auf äußere Schichten vorhanden, so dass die Dependency Rule eingehalten wird.

### 2.2.3 Adapters

In der Adapter-Schicht sind sowohl die Adapter für die Wetter- und Bild-API, als auch für die Konfiguration vorhanden. In dem `Image-` und `WeatherHandler` werden die JSON-Objekte, die von der API kommen, zu C#-Objekten gemapt und andersrum mithilfe einer Liste von strings die Routenparameter zusammengebaut und an die API weitergegeben. Der `ConfigHandler` wandelt die als JSON gespeicherte Konfigurationsdatei in ein C#-Objekt um, damit diese geladen werden kann. In die andere Richtung wird zum Abspeichern der Konfiguration aus dem C#-Objekt ein string gemacht. Eine weitere Klasse, die sich in die Adapter-Schicht einordnen lässt,

ist der `MainWindowController`. Hierbei werden die Eingaben der GUI weiter gereicht. Ein Beispiel hierfür ist das manuelle Aktualisieren des Hintergrunds. Dabei gibt die GUI dieses Signal an den `MainWindowController` weiter, welcher dann mithilfe des `ScreenChangeWorkers` den Hintergrund aktualisiert.

### 2.2.4 Plugins

In der Plugin-Schicht sind, wie vorher bereits beschrieben unsere `Image-` und `WeatherAPICaller` angesiedelt, die falls eine Technologie für die API entwickelt wird bzw. verwendet werden soll nur diese ausgetauscht werden müssen. Dazu gliedert sich auch der `DownloadHelper` ein, der ein Bild aus dem Internet herunter läd. Der `ImageWriter` und `FileAccessor` können einerseits ein Bild auf die Festplatte schreiben, bzw. generell Dateien auf der Festplatte lesen und schreiben. Sollte sich das Dateisystem oder Speichermedium ändern, müsste der Code angepasst werden und ist dementsprechend in der äußersten Schicht. Auch der `BackgroundChanger` ordnet sich in er Plugin-Schicht ein, da dieser nur für das Windows-Betriebssystem den Hintergrund ändern kann. Möchte man das Betriebssystem ändern, so müsste auch er angepasst werden. Selbes gilt für den `StartupHelper`, der die Anwendung in den Autostart des Windows-Systems setzt bzw. von dort wieder entfernt. Zu guter Letzt ist die GUI Teil der Plugin-Schicht. Sollte in Zukunft beispielsweise auf eine Web-Oberfläche anstelle einer aktuellen WindowsForms-Oberfläche gewechselt werden, so müsste dieses Plugin angepasst werden.

### 2.2.5 Mögliche Erweiterungen/Herausforderungen

Da beim Beginn des Projektes die API-Responses (`WeatherResponse` und `ImageResponse`) auch für die Verarbeitung im Code weiterverwendet wurden, sind sie aktuell als Entität im Domain-Code hinterlegt. Sollte sich nun allerdings das Format der API bzw. die API generell ändern, so müssten neue Klassen zum Mappen hinzugefügt werden. Diese neuen Klassen würden sich dann in der Adapter-Schicht befinden und der jeweilige Adapter (z.B. `WeatherHandler`) müsste dann erst das JSON-Objekt auf diese Klasse mappen. Nach dem Mapping schreibt der `WeatherHandler` dann die benötigten Informationen in das `WeatherResponse`-Objekt, damit dieses dann intern weiterverwendet werden kann.

# 3 Entwurfsmuster

- >= 1 Entwurfsmuster einsetzen und begründen
- UML-Diagramm vorher und nachher

# 4 Programming Principles

## 4.1 SOLID

### 4.1.1 Single Responsibility Principle

Das Single Responsibility Principle (SRP) wird auch als Prinzip der einzigen Zuständigkeit bezeichnet. Dementsprechend soll eine Klasse nur einen Grund haben, um geändert werden zu müssen. Dadurch erhält jedes Objekt eine klar definierte Aufgabe. Durch das Anwenden dieses Prinzips, wird die Seperation of Concerns (SoC) umgesetzt. Sollte das Prinzip der Single Responsibility verletzt sein, so lässt sich dies relativ einfach mit der Antwort auf die Frage „Was macht die Klasse?“ herausfinden. Sollte sich eine Konjunktion in der Antwort auf diese Frage befinden, kann man davon ausgehen, dass das SRP verletzt ist.

Als Beispiel für die Anwendung des SRP wird der ehemalige `MainController` betrachtet. Die Klasse als UML ist in Abbildung 1 zu sehen. Auf die Frage „Was macht die Klasse?“ lassen sich vier Antworten finden:

- Die generelle Logik, um den Hintergrund zu aktualisieren.
- Das Halten und Verwalten des Timers für die zyklische Aktualisierung des Hintergrundbilds.
- Die Logik, um das Hintergrundbild manuell in einem neuen Thread zu aktualisieren.
- Das Weitergeben einer neuen Config zum Speichern bzw. das Abrufen der aktuellen Config (Adapter-Tätigkeit).

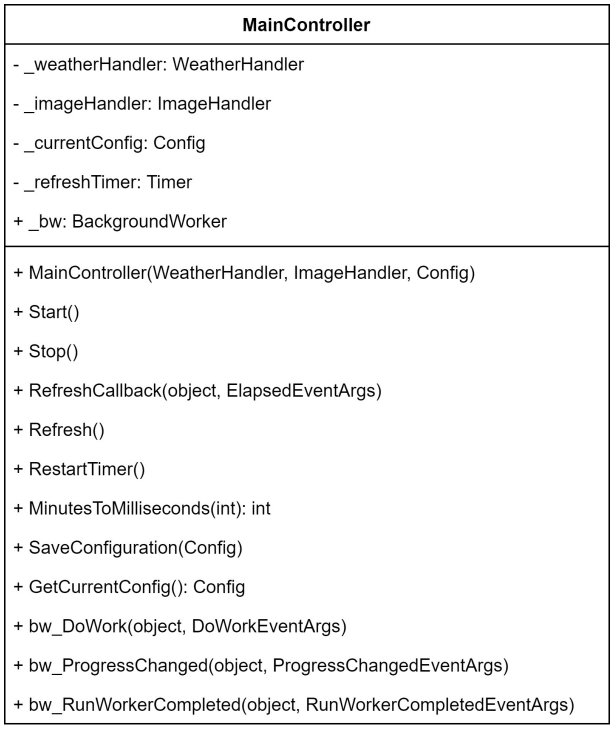


Abbildung 1: MainController in UML-Form

Um diese verschiedenen Responsibilities in neue Klassen aufzuteilen wurden vier neue Klassen entwickelt. Diese werden im Folgenden kurz erläutert. Im `Refresher` ist die generelle Logik, um den Hintergrund zu aktualisieren ausgelagert. Der `UpdateTimer` übernimmt das zyklische

Aktualisieren des Hintergunds und der `ScreenChangeWorker` das manuelle Aktualisieren im neuen Thread. In dem neuen `MainWindowController` wird das Weitergeben der GUI-Inputs an die inneren Schichten umgesetzt. Diese vier Klassen sind hier auf dem UML-Diagramm der Anwendung zu erkennen. Hierbei ist erkennbar, dass der `MainWindowController`, durch seine Adaptertätigkeiten auch in die Adapter-Schicht bei der Clean Architecture übergegangen ist.

Dann noch ConfigHandler bzw Adapter und Zustand

### 4.1.2 Open/Closed Principle

Das Open/Closed Principle beschreibt, dass Softwäre-Entitäten offen für Erweiterung aber geschlossen bezüglich Veränderung sein sollen. Dementsprechend soll bestehender Code nicht mehr geändert werden. Bei neuen bzw. geänderten Anforderungen wird der bestehende Code also nicht angepasst/geändert, sondern lediglich erweitert.

Bei unserer Anwendung identifiziert man eine mögliche Entwicklung mit Erweiterung klar bei der Validierung der Konfiguration, hier wird also das OCP verletzt. Gerade wenn sich diese in der Zukunft nochmals anpassen sollte, weil bspw. noch weitere Präferenzen des Nutzers/der Nutzerin erfasst werden sollen. In Listing 4.1 ist erkennbar, dass der Code zum Überprüfen angepasst

```
1  public static bool ValidateInputs(Config conf)
2  {
3      if (conf.Interval < 10 || conf.Interval > 300)// check for right interval range
4      {
5          throw new BadConfigException("Intervall im falschen Wertebereich." +
6                                       " Intervall muss zwischen 10 und 300 liegen.");
7      }
8      // check if city contains only letters
9      if (!Regex.IsMatch(conf.Location.City, @"^[a-zA-Z -']+$"))
10     {
11         throw new BadConfigException("Stadtname beinhaltet unbekannte Zeichen.");
12     }
13     return true;
14 }
```

Listing 4.1: Verletzung des Open/Closed-Principle im ConfigValidator

werden müsste, falls eine neue Anforderung an die Konfiguration hinzugefügt wird. Daher wird der bisherige `ConfigValidator` umgeschrieben und hält nun eine `List<IValidationAspect>`. In dieser Liste sind alle Validierungs-Aspekte gespeichert gegen die die Eingabe getestet werden soll. Das Interface `IValidationAspect` definiert eine Methode `Validate(Config)`, welche validiert, ob die Regel eingehalten wurde. Sollte dies nicht der Fall sein, wird eine Exception geworfen. Nun muss im `ConfigValidator` lediglich über alle registrierten `IValidationAspects` iteriert werden und mit der übergebenen Config die `Validate`-Methode aufgerufen werden. Das Hinzufügen einer neuen Anforderung ist nun simpel über das Schreiben einer neuen Klasse möglich. Diese muss `IValidationAspect` implementieren und auf den `ConfigHandler` registriert werden. Der neue `ConfigValidator` und ein Beispiel für einen `IValidationAspect` ist in Listing 4.2 zu sehen. Hierbei ist zu beachten, dass bei negativem Testergebnis eine Exception geworfen (und kein false zurückgegeben) wird, um die genaue Fehlerursache dem Nutzer/der Nutzerin mitzuteilen. Wird keine Exception geworfen, ist mit der Konfiguration alles in Ordnung.

Ein Punkt an dem das OCP nicht erfüllt ist, ist der `WeatherInterpreter`. Sollten in Zukunft weitere Daten zur Interpretation des Wetters dazu kommen, so müsste der Code des `WeatherInterpreters` angepasst und verändert werden.

```
// ConfigValidator
private List<IValidationAspect> _validationAspects;

public void Register(IValidationAspect aspect)
{
    _validationAspects.Add(aspect);
}
public bool ValidateInputs(Config conf)
{
    foreach (var aspect in _validationAspects)
    {
        aspect.Validate(conf);
    }
    return true;
}
// Validation for City-Name
public class IsCorrectCity : IValidationAspect
{
    public void Validate(Config conf)
    {
        // check if city contains only letters
        if (!Regex.IsMatch(conf.Location.City, @"^[a-zA-Z -']+$"))
        {
            throw new BadConfigException("Stadtname beinhaltet unbekannte Zeichen.");
        }
    }
}
```

Listing 4.2: Entwicklung mit Erweiterung für den ConfigValidator

### 4.1.3 Liskov Substitution Principle

Das Liskov Substitution Principle sagt aus, dass Subtypen sich so wie ihr Basistyp verhalten müssen. Subtypen dürfen daher lediglich die Funktionalität ihres Basistyps erweitern, aber nicht einschränken. Das Liskov Substitution Principle ist in unserer Anwendung erfüllt, da abgesehen von den verwendeten Interfaces keine Vererbung verwendet wird.

### 4.1.4 Interface Segregation Principle

### 4.1.5 Dependency Inversion Principle

## 4.2 GRASP

## 4.3 DRY

# 5 Refactoring

- Code Smells identifizieren
- >= 2 Refactoring anwenden und begründen

# 6 Unit Tests

Insgesamt wurden 29 Unit-Test geschrieben. Im Folgenden werden auf Einzelheiten zu den Unit-Tests eingegangen. Als Test-Framework wird das Framework XUnit verwendet.

## 6.1 ATRIP

Die entwickelten Unit-Tests befolgen die ATRIP-Regeln. Das bedeutet also, dass sie...

- Automatic, also eigenständig ablaufen und ihre Ergebnisse selbst prüfen. Dies wird durch das Testing-Framework XUnit gewährleistet.
  - In Abbildung 2 wird der Test-Explorer dargestellt. Darüber lassen sich die Tests ausführen und die Ergebnisse überprüfen.
- Thorough, also gründlich (genug) sind und die wichtigsten Funktionalitäten prüfen. Dazu gehört bei unserem Use-Case:
  - Die Analyse der Wetterdaten,
  - Die Validierung der vom Nutzer eingegebenen Konfiguration,
  - Das Decodieren der Konfiguration,
  - Die Verarbeitung der Daten der APIs, sowie die Fehlerbehandlung der APIs
- Repeatable, also jederzeit (automatisch) ausführbar sind. Dabei wird beispielsweise im Falle des WeatherInterpreterTest darauf geachtet, dass der Test nicht von der aktuellen Systemzeit abhängig ist.
  - Dieser Unit-Test ist hier zu finden.
- Independent, also unabhängig voneinander in beliebiger Reihenfolge ausführbar sind. Kein Test ist von dem Ergebnis oder dem Ablauf eines anderen Tests abhängig.
- Professional, also einfach lesbar und verständlich sind.
  - Ein Beispiels-Unit-Test ist hier zu finden.

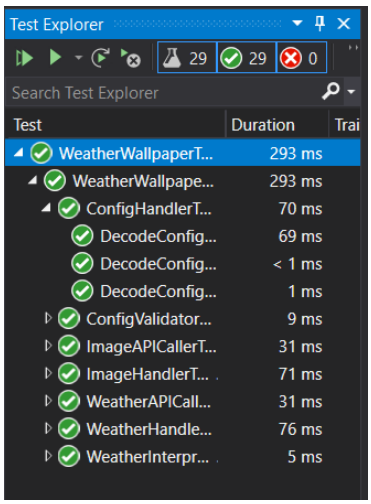


Abbildung 2: Unit-Test Ergebnisse im Test-Explorer

## 6.2 Beispiel für Unit-Tests und Mocks

Die Unit-Test wurden, sofern möglich, in der AAA-Normalform entwickelt. Bei Unit-Tests, die Exceptions erwarten musste der Act- und Assert-Schritt teilweise zusammengeführt werden. In Listing 6.1 wird einerseits gezeigt, wie der zu testende ConfigValidator im Konstruktor vor jedem Testdurchlauf neu initialisiert wird und die `ValidationAspects` registriert werden. Im Test selbst wird eine fehlerhafte Konfiguration erzeugt, da im Stadtnamen Zahlen vorhanden sind. Daraufhin wird überprüft, ob die Validierung die richtige `Exception` wirft und ob die `ExceptionMessage` richtig ist, also der Fehler korrekt erkannt wurde.

```
public ConfigValidatorTest()
{
    // ConfigValidator is needed in every unit test, so we initialize it here
    _configValidator = new ConfigValidator();
    _configValidator.Register(new IsCorrectCity());
    _configValidator.Register(new IsCorrectInterval());
}

[Fact]
public void ValidateInputsFalseCity()
{
    // Arrange
    const string city = "F4k3 ci7y";
    const string country = "DE";
    const int interval = 10;
    Config conf = new Config()
    {
        Interval = interval,
        Location = new Location()
        {
            City = city,
            CountryAbrv = country
        }
    };
    const string actualExceptionMessage
        = "Stadtname beinhaltet unbekannte Zeichen.";
    // Act, Assert
    var ex = Assert.Throws<BadConfigException>(
        () => _configValidator.ValidateInputs(conf));
    Assert.Equal(actualExceptionMessage, ex.Message);
}
```

Listing 6.1: Unit-Test für den ConfigValidator

Mock-Objekte werden verwendet um einzelne Klassen isoliert zu testen und damit lediglich die einzelne Unit, also Einheit zu testen. Diese Mock-Objekte ersetzen die eigentlichen Abhängigkeiten der Klasse. Dabei wird ein Mock vorerst eingelernt, um mindestens die notwendige Funktionalität der Abhängigkeit bereitstellen zu können und nach dem Verwenden verifiziert, um sicher zu gehen, dass die gemockte Methode auch aufgerufen wurde. Auch hierbei wurde versucht die extra Schritte Capture und Verify, sofern möglich, aufzuzeigen und zu verwenden. Dies war beispielsweise bei dem Mocken eines `HttpClient` nicht möglich, da die-

ser kein Interface anbietet, dass sich mocken lässt. Daher wird hierbei auf das NuGet-Paket `RichardSzalay.MockHttp` zurückgegriffen, welches hierfür einen Workaround bietet. Das Paket ermöglicht allerdings leider kein Verifizieren. Als Beispiel für Mocks wird in Listing 6.2 ein Test für den `ImageHandler` dargestellt. Dabei wird das Interface `IAPICaller` gemockt und die in Zeile x definierte `correctApiResponse` beim Aufruf der `Get`-Methode des API-Callers zurückgegeben. Durch den Einsatz des Mocks, lässt sich die Funktionalität des `ImageHandlers` testen ohne einen realen API-Caller zu verwenden. Am Schluss wird überprüft, ob das Ergebnis des Aufrufs mit dem erwarteten, eingegebenen Ergebnis übereinstimmt.

```
[Fact]
public void GetImageDataSuccessful()
{
    // Capture
    var correctApiResponse = new ImageResponse()
    {
        Results = new List<Images>{
            new Images { Links = new Links
            {
                Download = "https://unsplash.com/photos/XxElwSAHOAA/download"
            }}}
    };
    var responseJson = JObject.FromObject(correctApiResponse);
    var api = new Mock<IAPICaller>();
    api.Setup(caller => caller.Get(It.IsAny<string>()))
        .Returns(Task.FromResult(responseJson)).Verifiable();
    // Arrange
    var handler = new ImageHandler(api.Object);
    string queryString = "?query=doesnt matter";
    // Act
    var result = handler.GetImageData(queryString);
    // Assert
    Assert.Equal(result.Results.First().Links.Download,
        correctApiResponse.Results.First().Links.Download);
    // Verify
    api.Verify();
}
```

Listing 6.2: Unit-Test für den ImageHandler mit Mock

## 6.3 Code Coverage

Mithilfe der Visual Studio 2019 Enterprise Version lässt sich die Code Coverage für das Projekt ermitteln. Dabei erreicht WeatherWallpaper eine Code Coverage von knapp 42%. Dies ist in Abbildung 3 zu sehen. Hierbei ist nur das blau hinterlegte Projekt zu beachten, da das andere Projekt das Test-Projekt selbst ist. Des Weiteren bietet das Visual Studio Tool die Möglichkeit einzusehen, welche Zeilen von den Tests abgedeckt werden und welche nicht. Zeilen die nicht abgedeckt werden, werden rot hinterlegt und abgedeckte Zeilen blau, wie in Abbildung 4 zu sehen.

| Hierarchy | Not Covered (Blocks) | Not Covered (% Blocks) | Covered (Blocks) | Covered (% Blocks) |
|---|---|---|---|---|
| janis_DESKTOP-4OTT554 2021... | 354 | 27,48 % | 934 | 72,52 % |
| weatherwallpaper.dll | 349 | 57,97 % | 253 | 42,03 % |
| weatherwallpapertest.dll | 5 | 0,73 % | 681 | 99,27 % |

Abbildung 3: Code Coverage Ergebnisse

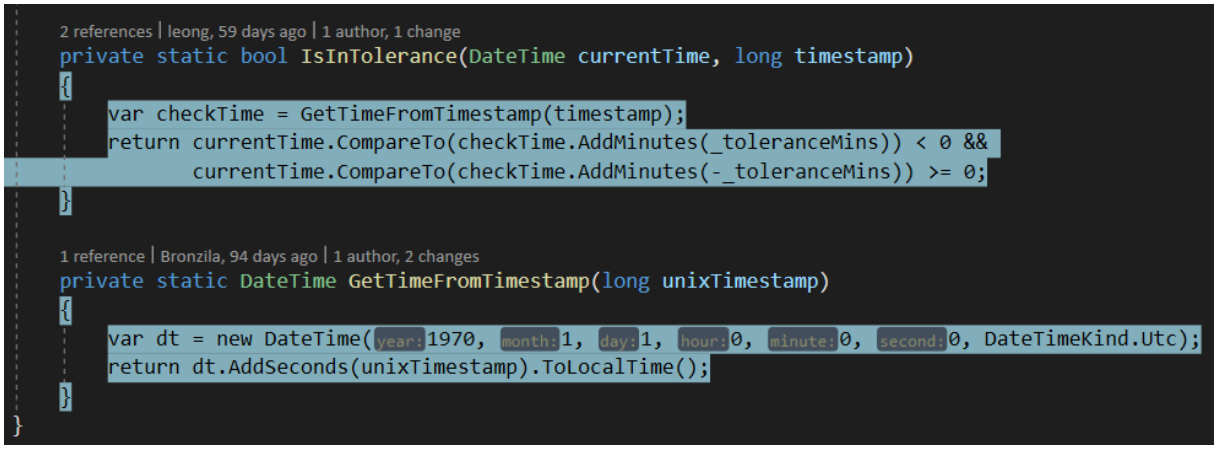


Abbildung 4: Code Coverage Highlighting