

# WeatherWallpaper

Janis Fix, Leon Gieringer

TINF18B3

Advanced Software Engineering

27. Mai 2021

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Clean Architecture</b>	<b>2</b>
2.1	Vorher . . . . .	2
2.2	Nachher . . . . .	2
<b>3</b>	<b>Entwurfsmuster</b>	<b>3</b>
<b>4</b>	<b>Programming Principles</b>	<b>4</b>
4.1	SOLID . . . . .	4
4.1.1	Single Responsibility Principle . . . . .	4
4.1.2	Open/Closed Principle . . . . .	4
4.1.3	Liskov Substitution Principle . . . . .	4
4.1.4	Interface Segregation Principle . . . . .	4
4.1.5	Dependency Inversion Principle . . . . .	4
4.2	GRASP . . . . .	4
4.2.1	Low Coupling & High Cohesion . . . . .	4
4.2.2	Indirection . . . . .	5
4.2.3	Polymorphism . . . . .	6
4.2.4	Pure Fabrication . . . . .	6
4.2.5	Protected Variations . . . . .	7
4.3	DRY . . . . .	8
4.4	YAGNI . . . . .	8
<b>5</b>	<b>Refactoring</b>	<b>9</b>
5.1	Code Smells . . . . .	9
<b>6</b>	<b>Unit Tests</b>	<b>10</b>
6.1	ATRIP . . . . .	10
6.2	Beispiel für Unit-Tests und Mocks . . . . .	10
6.3	Code Coverage . . . . .	10

## Listings

4.1	Alte Implementierung des ImageHandlers mit hoher Kopplung und niedriger Kohäsion . . . . .	5
4.2	Überarbeitung des ImageHandlers mit niedrigerer Kopplung und hoher Kohäsion	6
4.3	ConfigValidator als Problemdomäne und IsCorrectCity als Technologiewissen . .	7
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**

### **2.1 Vorher**

### **2.2 Nachher**

- Schichtarchitektur planen und begründen
- $\geq 2$  Schichten umsetzen

### **3 Entwurfsmuster**

- $\geq 1$  Entwurfsmuster einsetzen und begründen
- UML-Diagramm vorher und nachher

## 4 Programming Principles

### 4.1 SOLID

#### 4.1.1 Single Responsibility Principle

#### 4.1.2 Open/Closed Principle

#### 4.1.3 Liskov Substitution Principle

Das Liskov Substitution Principle ist 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

GRASP (General Responsibility Assignment Software Patterns/Principles) sind Standardlösungen für typische Fragestellungen in der Software Entwicklung. Diese wurden zum ersten mal von Craig Larman in seinem Buch „Applying UML and Patterns“ aus dem Jahre 1995 vorgestellt. Die ersten sechs dieser neun Prinzipien und deren Anwendung in unserem Programm werden im folgenden Verlauf erläutert.

#### 4.2.1 Low Coupling & High Cohesion

Diese beiden Prinzipien sind essentielle Grundkonzepte bei GRASP.

*Low Coupling* ist ein Maß über die Abhängigkeit einer Komponente zu ihrem Umfeld. Das Prinzip ermöglicht vor allem eine höhere Anpassbarkeit der Komponente, ein einfacheres Verständnis über die Funktionsweise dieser und ein leichteres Testen aufgrund geringer Abhängigkeiten zu anderen Komponenten.

*High Cohesion* hingegen gibt Auskunft über den Zusammenhalt innerhalb einer Komponente. Es wird also gemessen, wie eng die Methoden und Attribute innerhalb einer Komponente (bspw. einer Klasse) zusammenarbeiten. Dies reduziert hauptsächlich die Komplexität des Gesamtsystems, da Klassen sinnvoll strukturiert werden. Beide bedingen sich gegenseitig: Code mit hoher Kohäsion besitzt oft eine geringe Kopplung.

Die Klassen `WeatherHandler` und `ImageHandler` erhalten Informationen über die derzeitige Konfiguration bzw. über das aktuelle Wetter. Sie sind dafür zuständig mithilfe der jeweiligen Information eine URL für die spätere API Abfrage zu erstellen. Für das Zusammenstellen der zusätzlichen Abfrageattribute ist die jeweilige Funktion `BuildRouteString()` zuständig. Beim `ImageHandler` beispielsweise wurde diese Funktion außerhalb der eigentlichen Klasse - in der Controller-Klasse `Refresher` - aufgerufen. Dabei wurde ein String mit zusätzlichen Abfrageparameter zusammengebaut und daraufhin die Funktion `GetImageData()` mit eben diesem String als Parameter aufgerufen. Im Codebeispiel 4.1 ist die Klasse `ImageHandler` vereinfacht dargestellt.

Dadurch besitzt die Klasse zwei direkte Kopplungen zur `Refresher` Klasse und gleichzeitig wird die Route-String Funktion nie in der eigenen Klasse aufgerufen. Nach den genannten Prinzipien soll `BuildRouteString()` nicht mehr außerhalb, sondern aus der `GetImageData()` Funktion heraus aufgerufen werden. Da die Interpretation der aktuellen Wetterdaten zum Bau des Strings benötigt wird, muss diese der Funktion `GetImageData()` als Parameter übergeben werden an

---

```
1 public class ImageHandler : IImageHandler
2 {
3     public ImageResponse GetImageData(String QueryString)
4     {
5         var response = _api.Get(QueryString);
6         response.Wait();
7         return response.Result.ToObject<ImageResponse>();
8     }
9
10    public string BuildRouteString(WeatherInterpretation weatherInterpretation)
11    {
12        return $"{"?query={weatherInterpretation.Weather}
13                weatherInterpretation.Daytime} {weatherInterpretation.Feeling}";
14    }
15 }
```

---

Listing 4.1: Alte Implementierung des ImageHandlers mit hoher Kopplung und niedriger Kohäsion

Stelle des fertigen Route-Strings. Somit wird die Abhängigkeit der Klasse zum **Refresher** reduziert und gleichzeitig die Kohäsion erhöht. Die neue Version wird in dem Beispiel 4.2 gezeigt<sup>1</sup>.

Ähnliches wurden ebenfalls für die **WeatherHandler** Klasse mit der Funktion **LocationAsRouteAttribute()** geändert. Auch hier verringert sich die Abhängigkeit und die Kohäsion steigt.

#### 4.2.2 Indirection

Unter *Indirection* versteht man die zentrale Verwaltung von Aufgaben an einzelne Komponenten, wodurch untereinander keine direkte Kopplung bestehen muss. Somit ist es ein Prinzip zur Code-Strukturierung und kann zu geringer Kopplung führen, da eine direkte Abhängigkeit der Klassen untereinander vermieden wird. Gleichzeitig bietet diese Struktur trotzdem weiterhin gute Möglichkeiten Komponenten wiederzuverwenden.

Bei unserem Programmentwurf soll ein Timer nach einem gewissen Intervall alle Aktionen zum Wechsel des Hintergrundbildes durchführen. Da hierbei viele verschiedene Komponenten zusammenhängen, beispielsweise die Benutzeroberfläche, zwei unterschiedliche API-Abfragen und das Herunterladen eines Bildes, wäre das Projekt sehr schnell unübersichtlich und verschachtelt geworden.

Alle Klassen hätten somit untereinander eine Abhängigkeit und Wiederverwendbarkeit wäre nur eingeschränkt möglich. Deshalb wurde die zentrale Klasse **Refresher** mit der Funktion **Refresh()** implementiert. Diese kann vom Timer oder direkt aus der Benutzeroberfläche aufgerufen werden und „delegiert“ die Arbeit an einzelne Klassen. Funktionsaufrufe werden nacheinander ausgeführt und die benötigten Rückgabewerte als Parameter in die nächste Funktion übergeben. Dadurch entsteht eine indirekte Abhängigkeit, wodurch alles übersichtlich bleibt.

---

<sup>1</sup>Diese Änderungen wurden im Commit <https://github.com/Bronzila/WeatherWallpaper/commit/25e8c58a47945383746d8151ed4bfbed01b1d24c> durchgeführt.



---

```

1 public class ImageHandler : IImageHandler
2 {
3     public ImageResponse GetImageData(WeatherInterpretation weatherInterpretation)
4     {
5         var queryString = BuildRouteString(weatherInterpretation);
6         var response = _api.Get(queryString);
7         response.Wait();
8         return response.Result.ToObject<ImageResponse>();
9     }
10
11     public string BuildRouteString(WeatherInterpretation weatherInterpretation)
12     {
13         return $"{"?query={weatherInterpretation.Weather}"
14             {weatherInterpretation.Daytime} {weatherInterpretation.Feeling}";
15     }
16 }

```

---

Listing 4.2: Überarbeitung des ImageHandlers mit niedrigerer Kopplung und hoher Kohäsion

### 4.2.3 Polymorphism

Ebenfalls zur Code-Strukturierung trägt das Prinzip des Polymorphismus bei. Dadurch kann das Verhalten abhängig vom konkreten Typ jeweils geändert werden; Funktionen erhalten somit eine neue Implementierung.

Dies ist bei unserem Programm in der Klasse `ConfigValidator` deutlich zu erkennen. Sie validiert die eingegebene Funktion, wobei mehrere Validierungsaspekte separat in Betracht gezogen werden. Validierungsaspekte sind Implementierungen des Interfaces `IValidationAspect`, welches eine Funktion `Validate(Config)` besitzt.

Diese Aspekte können nun einzeln über die Funktion `Register(IValidationAspect)` zu einer Liste hinzugefügt werden. Über `ValidateInputs(Config)` wird die eingegebene Konfiguration anhand dieser Aspekte überprüft. Dafür wird mit jedem Aspekt in der Liste die jeweilige Funktion `Validate(Config)` aufgerufen. Da die Implementierungen dieser Funktion sich von Typ zu Typ unterscheiden, liegt hier eine indirekte Konditionalstruktur, also Polymorphismus vor. Im Listing 4.3 ist der Zusammenhang von `IValidationAspect` und `ValidateInputs(Config)` deutlich gezeigt.

### 4.2.4 Pure Fabrication

Um eine geringe Kopplung und gleichzeitig hohe Kohäsion erreichen zu können, trennt dieses Prinzip die Problemdomäne von der zugrundeliegenden Technologie. Es entsteht also eine Klasse ohne jeglichen Bezug zum Problem, sie kann somit überall wiederverwendet werden - eine reine Service Klasse bei Domain-Driven-Design. Daher beschreibt das Prinzip den Aufbau der Architektur im Einklang mit den anderen Prinzipien.

Die Klasse `ConfigHandler` ist für die Verwaltung der Konfiguration des Nutzers zuständig. Der Zugriff auf das Dateisystem ist über das Interface `IFileAccessor` ausgelagert. Dadurch ist die Problemdomäne durch den `ConfigHandler` abgedeckt. Er löst das Problem der Verwaltung der Konfiguration und delegiert die eigentliche Umsetzung des Speicherns und Ladens an eine andere Komponente.

---

```
1  // ConfigValidator
2  public bool ValidateInputs(Config conf)
3  {
4      foreach (var aspect in _validationAspects)
5      {
6          aspect.Validate(conf);
7      }
8      return true;
9  }
10
11 // Implementation of IValidationAspect
12 public class IsCorrectCity : IValidationAspect
13 {
14     public void Validate(Config conf)
15     {
16         if (!Regex.IsMatch(conf.Location.City, @"^[a-zA-Z -]+$"))
17         {
18             throw new BadConfigurationException("Stadtname beinhaltet unbekannte
19                 Zeichen.");
20         }
21     }
22 }
```

---

Listing 4.3: ConfigValidator als Problemdomäne und IsCorrectCity als Technologiewissen

Gleichzeitig kümmert sich die Klasse `ConfigValidator`, wie in Kapitel 4.2.3 beschrieben um das Problem der Validierung einer Konfiguration. Hierbei werden verschiedene Validierungsaspekte in Betracht gezogen; jeder Aspekt wird einzeln validiert. Die Methode `ValidateInputs(Config)` befindet sich somit in der Problemdomäne und gibt an, ob alle Eingaben valide sind. Einzelne Aspekte werden über Implementierungen des Interfaces `IValidationAspect` auf Technologie-Ebene untersucht. Die Funktion `Validate(Config)` ist auf jeden Aspekt angepasst und bestätigt dessen Korrektheit. Beide Funktionen sind in der Abbildung 4.3 dargestellt.

#### 4.2.5 Protected Variations

Um Elemente bei der Kopplung mit variierenden Implementierungen zu schützen, soll laut diesem Prinzip über eine gemeinsame Schnittstelle zugegriffen werden. Somit können Veränderungen eines Elementes keinen (unerwünschten) Einfluss auf andere Elemente haben.

Bei der Clean Architecture 2 ist über das Dependency Inversion Prinzip geregelt, dass Abhängigkeiten von innen nach außen zu jeder Zeit über ein Interface umgekehrt werden können und somit der Dependency Rule entsprechen. Dies ist ein generelles Beispiel von Protected Variations, da man somit die innere Schicht über eine gemeinsame Schnittstelle vor einer möglichen Änderung der äußeren Schicht schützt.

Ein genaueres Beispiel ist das Interface `IAPICaller`, welches den Aufruf aus beispielsweise der Klasse `ImageHandler` (Adapter-Schicht) auf die Klasse `APICaller` (Plugin-Schicht) ermöglicht. Es wird also dem `ImageHandler` sichergestellt, dass er über die Funktion `Get(string)` Daten über Bilder erhält. Somit kann eine Änderung der jeweiligen Implementierung der Beschaffungs-

art keinerlei Schaden beim `ImageHandler` verursachen, da eine klare gemeinsame Schnittstelle mit gegebenen Parametertypen und Rückgabetyphen genutzt wird. Er muss somit nicht angepasst werden, wenn eine beschriebene Änderung vorliegt.

Ähnliches gilt auch für das Interface `IBackgroundChanger`. Das Wechseln des Hintergrundbildes funktioniert auf verschiedenen Betriebssystemen (selbst bei verschiedenen Windowsversionen) unterschiedlich. Um diesen Änderungen standhalten zu können, bietet das Interface mit der `Set(string)` Funktion eine einheitliche Schnittstelle. Die jeweilige Implementierung kann nun geändert werden, ohne dass eine Änderung in einer inneren Schicht notwendig ist.

### 4.3 DRY

Das Prinzip *Don't Repeat Yourself* zielt darauf ab, Code-Wiederholungen durch Normalisierung und Abstraktion zu eliminieren. Es wird also darauf geachtet, dass Code nur an einer einzigen Stelle im System geschrieben und verwaltet wird.

Bei WeatherWallpaper werden Anfragen an insgesamt zwei APIs versendet. Hierfür wurde das Interface `IAPICaller` jeweils in den Klassen `ImageAPICaller` und `WeatherAPICaller` implementiert. Beide Klassen waren allgemein sehr ähnlich und unterschieden sich hauptsächlich über den `HttpClient`, welcher logischerweise unterschiedliche Basisadressen erhält um die jeweilige API zu erreichen. Gleichzeitig sind statische Felder, wie z.B. der API-Key oder andere, fix ausgewählte Parameter unterschiedlich. Beide Klassen verfügten über eine `Get()` Funktion, welche die variablen Parameter als String übergeben bekommen hat und die API Anfrage über den `HttpClient` mit entsprechender Adresse ausführt.

Aufgrund dieser Ähnlichkeit wurden beide Klassen als `APICaller` nach dem DRY-Prinzip zusammengefasst. Diese Klasse erhält im Konstruktor zusätzlich zum `HttpClient` eine Liste von Strings mit API-spezifischen Feldern. Diese Felder werden nun im Konstruktor in gleicher Reihenfolge als String zusammengebaut und ersetzen somit die statischen Felder in der Klasse selbst<sup>2</sup>.

### 4.4 YAGNI

Das Prinzip *You ain't gonna need it - du wirst es nicht brauchen* besagt, dass Code nur hingeschrieben wird, wenn dieser auch wirklich nötig ist. Ungenutzter Code muss nämlich zeitintensiv implementiert, getestet, dokumentiert und gewartet werden; er bringt also deutlichen Zusatzaufwand mit sich. Daher versucht dieses Prinzip diesen sogenannten Annahme-Code zu minimieren, bzw. eliminieren.

Ursprünglich war angedacht, dieses Projekt mit Hardware zu verknüpfen, an welcher dann etwa Temperatur oder Wetter allgemein angezeigt werden sollte. Hierfür war im `APICaller` eine `Put()` Methode bereits vorgesehen, jedoch nicht implementiert. Diese wurde nach diesem Prinzip verabschiedet, damit eben kein Annahme-Code mehr verwaltet werden muss<sup>3</sup>.

---

<sup>2</sup>Diese Änderung wurden im Commit <https://github.com/Bronzila/WeatherWallpaper/commit/fb4546895d2241b8ea2391991f102bcd8f2fb685> durchgeführt.

<sup>3</sup>Die Funktion wurde mit der ersten Umstrukturierung des `APICallers` im Commit <https://github.com/Bronzila/WeatherWallpaper/commit/d11c44ef92c85fed125288c195fb36ce595c660d> entfernt.

## 5 Refactoring

- Code Smells identifizieren
- $\geq 2$  Refactoring anwenden und begründen

### 5.1 Code Smells

## 6 Unit Tests

Insgesamt wurden 29 Unit-Test geschrieben. Im Folgenden werden auf Einzelheiten zu den Unit-Tests eingegangen.

### 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.
- 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.
- Independent, also unabhängig voneinander in beliebiger Reihenfolge ausführbar sind. Kein Test ist auf das Ergebnis oder den Ablauf eines anderen Tests abhängig.
- Professional, also einfach verständlich sind.

Hier wahrscheinlich noch Screenshots für die einzelnen Unterpunkte

### 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.

Als Beispiel für Mocks betrachten wir in Listing 6.2 ein Test für den `ImageHandler`. 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.

### 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 1 zu sehen. 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 2 zu sehen.

```

1 public ConfigValidatorTest()
2 {
3     // ConfigValidator is needed in every unit test, so we initialize it here
4     _configValidator = new ConfigValidator();
5     _configValidator.Register(new IsCorrectCity());
6     _configValidator.Register(new IsCorrectInterval());
7 }
8
9 [Fact]
10 public void ValidateInputsFalseCity()
11 {
12     // Arrange
13     const string city = "F4k3 ci7y";
14     const string country = "DE";
15     const int interval = 10;
16     Config conf = new Config()
17     {
18         Interval = interval,
19         Location = new Location()
20         {
21             City = city,
22             CountryAbrv = country
23         }
24     };
25     const string actualExceptionMessage
26         = "Stadtname beinhaltet unbekannte Zeichen.";
27     // Act, Assert
28     var ex = Assert.Throws<BadConfigException>(
29         () => _configValidator.ValidateInputs(conf));
30     Assert.Equal(actualExceptionMessage, ex.Message);
31 }

```

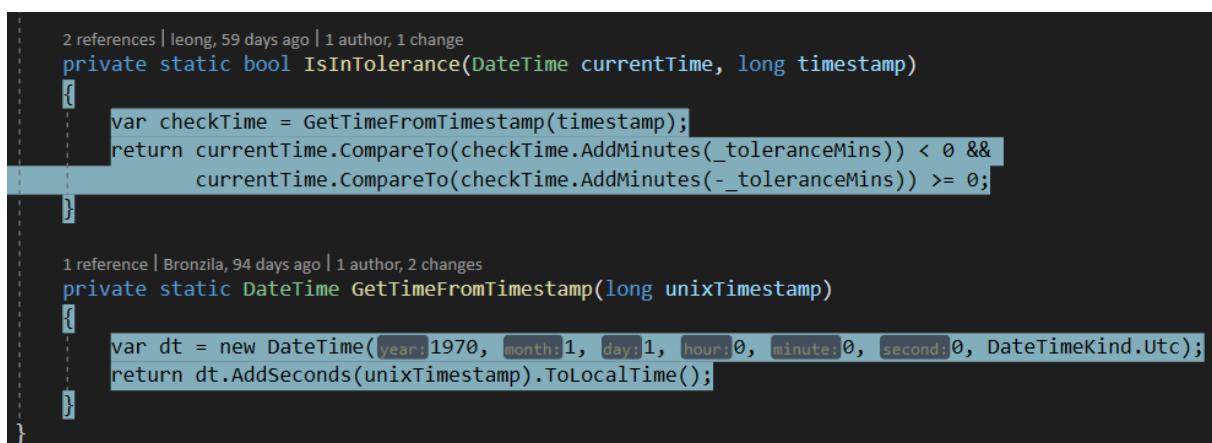
Listing 6.1: Unit-Test für den ConfigValidator

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Bloc...	Covered (% Blocks)
janis_DESKTOP-4OTT554 2021-05-18 19_15_41.coverage	355	27,54 %	934	72,46 %
weatherwallpapertest.dll	5	0,73 %	681	99,27 %
weatherwallpaper.dll	350	58,04 %	253	41,96 %
WeatherWallpaper.Classes.Helpers	115	47,33 %	128	52,67 %
WeatherWallpaper.Classes.Handler	13	16,88 %	64	83,12 %
WeatherWallpaper.Classes.API	0	0,00 %	43	100,00 %
WeatherWallpaper.Classes.Models	0	0,00 %	16	100,00 %
WeatherWallpaper.Classes.Exceptions	0	0,00 %	2	100,00 %
WeatherWallpaper	129	100,00 %	0	0,00 %
WeatherWallpaper.Classes.Background	39	100,00 %	0	0,00 %
WeatherWallpaper.Classes.Controllers	54	100,00 %	0	0,00 %

Abbildung 1: Code Coverage Ergebnisse

```
1 [Fact]
2 public void GetImageDataSuccessful()
3 {
4     //Arrange
5     var correctApiResponse = new ImageResponse()
6     {
7         Results = new List<Images>{
8             new Images { Links = new Links
9             {
10                 Download = "https://unsplash.com/photos/XxElwSAH0AA/download"
11             }}}
12 };
13 var responseJson = JObject.FromObject(correctApiResponse);
14 var api = new Mock<IAPICaller>();
15 api.Setup(caller => caller.Get(It.IsAny<string>()))
16     .Returns(Task.FromResult(responseJson));
17 var handler = new ImageHandler(api.Object);
18 string queryString = "?query=doesnt matter";
19
20 //Act
21 var result = handler.GetImageData(queryString);
22
23 //Assert
24 Assert.Equal(result.Results.First().Links.Download,
25     correctApiResponse.Results.First().Links.Download);
26 }
```

Listing 6.2: Unit-Test für den ImageHandler mit Mock



```
2 references | leong, 59 days ago | 1 author, 1 change
private static bool IsInTolerance(DateTime currentTime, long timestamp)
{
    var checkTime = GetTimeFromTimestamp(timestamp);
    return currentTime.CompareTo(checkTime.AddMinutes(_toleranceMins)) < 0 &&
        currentTime.CompareTo(checkTime.AddMinutes(-_toleranceMins)) >= 0;
}

1 reference | Bronzila, 94 days ago | 1 author, 2 changes
private static DateTime GetTimeFromTimestamp(long unixTimestamp)
{
    var dt = new DateTime(year:1970, month:1, day:1, hour:0, minute:0, second:0, DateTimeKind.Utc);
    return dt.AddSeconds(unixTimestamp).ToLocalTime();
}
```

Abbildung 2: Code Coverage Highlighting