WeatherWallpaper

Janis Fix, Leon Gieringer

TINF18B3

Advanced Software Engineering

30. Mai 2021

Inhaltsverzeichnis

Listings

1 Einleitung

Aufgrund einer globalen Pandemie und weit verbreiteten Schutzmaßnahmen waren viele Arbeitnehmer, Studierende und sogar auch Schüler gezwungen, ihre tagtägliche Arbeit nun von Zuhause aus zu verrichten. Nach längerer Selbststudie mussten wir leider alle merken, dass Home-Office nicht „Arbeiten von der eigenen Wohnung aus“ sondern eher „Wohnen auf der Arbeit“ ist.

Viele arbeiten deutlich länger, da die geographische Trennung zwischen *Zuhause* und *Arbeit* fehlt. Vor allem in den Wintermonaten macht es sich bemerkbar, wenn die überhaupt wenigen Sonnenstunden eines Tages von Teams-Meetings durchzogen sind. Um dieser tristen und grauen Welt etwas Farbe einzuhauchen gibt es nun: **WeatherWallpaper**

Dieses Programm bezieht aktuelle Wetterdaten eines definierten Standortes und zeigt je nach Wetterlage und Uhrzeit ein passendes Bild als Hintergrund an. Dadurch gleicht sich der Desktop eines jeden Nutzers an die aktuelle natürliche Stimmung, wodurch ein einheitliches Gefühl zwischen Arbeitsplatz und Umwelt vermittelt wird. Dies wiederum wertet das Klima nicht nur bei der Arbeit, sondern auch Zuhause spürbar auf.

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 UML-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

Entwurfsmuster befassen sich mit der Lösungsansätzen für wiederkehrende Entwurfsprobleme. Sie sind somit eine Standardvorlage um Probleme zu umgehen. Dies geschieht auf einer höheren Abstraktion als objekt-orientierte Programmierung selbst.
Grundlegend lassen diese Entwurfsmuster in vier Kategorien einordnen: Erzeugungsmuster (Erstellung von Instanzen), Strukturmuster (Vereinfachung des Designs), Verhaltensmuster (Zusammenarbeit von Komponenten) und Nebenläufigkeitsmuster (Multithreading).

Im folgenden wird die Anwendung eines Verhaltensmusters in unserem Programm erklärt.

Beobachter Das im Code identifizierte Entwurfsmuster ist das objektbasierte Verhaltensmuster „Beobachter“. Hierbei registriert sich in der `Start()` Methode der Klasse `UpdateTimer` dieser auf das `Elapsed` Event des `Timers`. Hierbei wird die Methode `RefreshCallback()` angemeldet. Diese delegiert den Methodenaufruf an die Klasse `Refresher`, um das Hintergrundbild zu ändern. Der `Timer` ist das Subjekt, bei welchem sich der `UpdateTimer` anmeldet.

Das folgende UML Diagramm zeigt beide Klassen und ihren Zusammenhang:

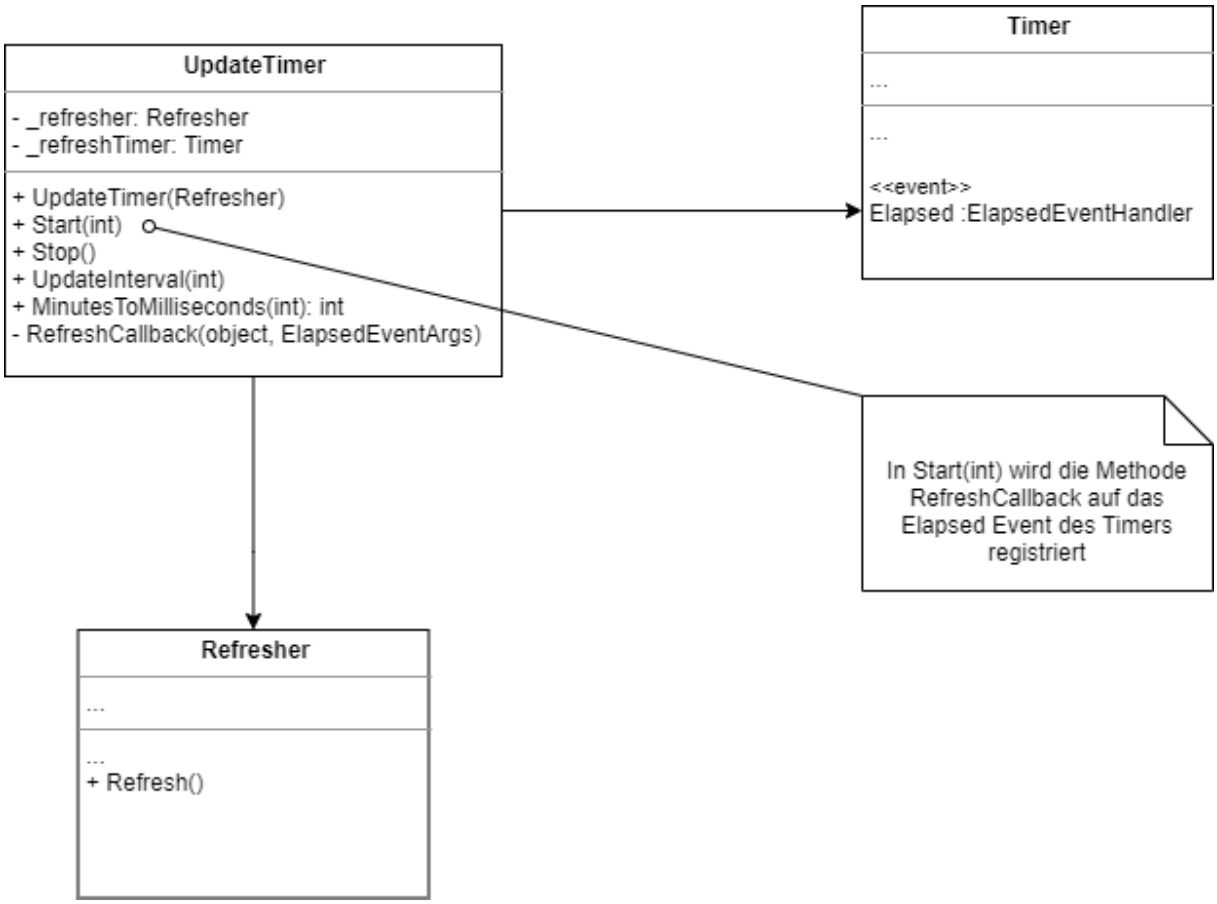

Abbildung 1: UML Diagramm des Entwurfsmuster „Beobachter“

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 2 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).

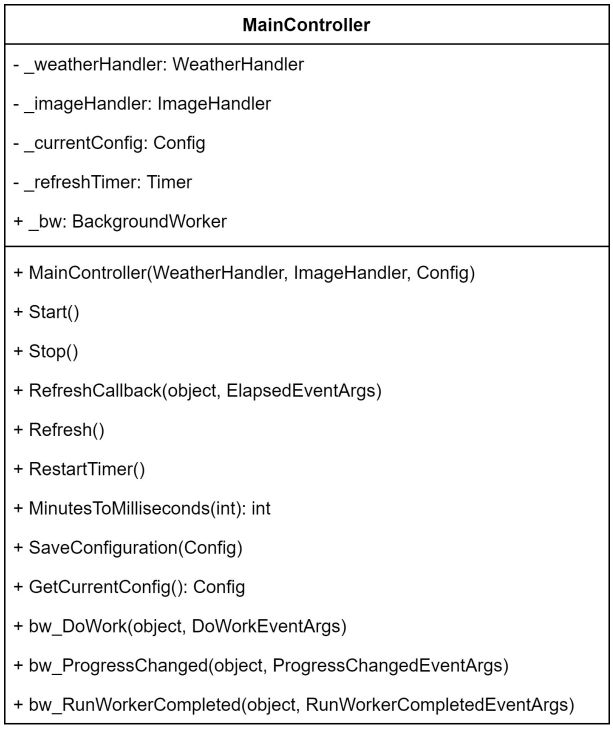

Abbildung 2: MainController in UML-Form

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

Das Interface Segregation Principle sagt aus, dass Interfaces passgenau für die aufrufenden Clients sein muss. Das Interface darf also keine Details beinhalten, die der Client gar nicht benötigt. Die Interfaces unserer Anwendung folgen diesem Prinzip. Anhand des Interfaces `IFileAccessor` kann man die Entwicklung hin zum Erfüllen des Prinzips gut nachvollziehen, da das Interface `IFileAccessor` von zwei Klienten verwendet wird. Dabei benötigt einerseits der `ConfigHandler` den Zugriff auf Dateien (Lesen und Schreiben) und andererseits der `DownloadHelper` um das Hintergrundbild zu speichern. Das Interface `IFileAccessor` umfasst daher sowohl das Lesen/Schreiben von Dateien als auch das Schreiben von Bildern. Dieses Interface wurde dementsprechend, um das Interface Segregation Principle zu erfüllen, in zwei Interfaces aufgeteilt. Einerseits das Interface `IImageWriter` und andererseits das (bereits bestehende) Interface `IFileAccessor`. Wie der Name schon sagt, kümmert sich das Interface `IImageWriter` um das Schreiben von Bil-

dern und das Interface `IFileAccessor` mit dem Lesen und Schreiben von Dateien. In diesem Commit ist das Segmentieren dieser beiden Interfaces zu erkennen. Generell haben einige Interfaces das ISP nicht erfüllt, allerdings war keins dieser Beispiele so eindringlich wie das Obere, da die meisten Interfaces nur ein aufrufenden Client hatten und auf diesen einzelnen Client nicht passgenau zugeschnitten waren. Mittlerweile wurden auch die Interfaces, die das Prinzip nicht erfüllt haben so umgeschrieben, dass sie es erfüllen.

4.1.5 Dependency Inversion Principle

Mit dem Dependency Inversion Principle wird versucht höhere Ebenen von niedrigeren Ebenen zu entkoppeln. Dabei gilt die Regel, dass Abstraktionen nicht von Details abhängen sollen, sondern Details von Abstraktionen abhängen sollten, da Abhängigkeiten auf konkrete Klassen stark koppeln. Sollte dieses Prinzip nicht erfüllt sein, so würden Änderungen in der niedrigeren Ebene Änderungen in der höheren Ebene hervorrufen. Eine Verletzung dieses Prinzips wird mithilfe der Dependency Injection aufgelöst. Dabei sind die Klassen der höheren und niedrigeren Ebene abhängig von einem Interface anstelle von einer konkreten Klasse. Die Referenz auf die Instanz erhalten die Klassen der höheren Ebene dann meist im Konstruktor. Die Verletzungen des Dependency Inversion Principle sind bei der Implementierung der Clean Architecture klar auffindbar. Sobald eine innere Schicht eine Abhängigkeit auf eine äußere Schicht hat, wird das Prinzip verletzt. In Kapitel 2.1 sind diese Verletzungen nochmals kurz erläutert. Im Folgenden wird beispielhaft die Abhängigkeit vom damaligen `MainController` (mittlerweile umbenannt in `Refresher`) auf den `WeatherHandler`. Hier wird das Prinzip verletzt, da sich der `Refresher` in der Application Code-Schicht befindet und der `WeatherHandler` in der Adapter-Schicht. Um dies zu beheben, wird das Interface `IWeatherHandler` eingesetzt. Der `WeatherHandler` implementiert dieses und der `Refresher` hat nur noch die Abhängigkeit auf ein `IWeatherHandler`. Das tatsächliche Objekt erhält er dann im Konstruktor über Dependency Injection. In Abbildung 3 ist dieser kleine Ausschnitt aus dem UML-Diagramm vereinfacht dargestellt und mit Vorher/Nachher betitelt.

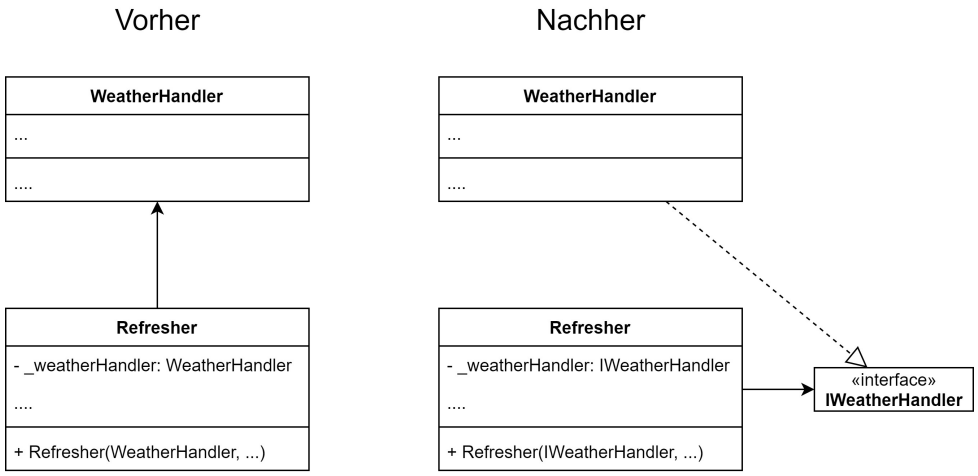

Abbildung 3: Beispielhafte Verletzung und Verbesserung des DIP

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 4.3 ist die Klasse `ImageHandler` vereinfacht dargestellt.

```
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 4.3: Alte Implementierung des ImageHandlers mit hoher Kopplung und niedriger Kohäsion

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 aktuelle Wetterdaten zum Bau des Strings

benötigt wird, muss diese der Funktion `GetImageData()` als Parameter übergeben werden an 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 4.4 gezeigt[1].

Ähnliches wurden ebenfalls für die `WeatherHandler` Klasse mit der Funktion `LocationAsRoute-`

```
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 4.4: Überarbeitung des ImageHandlers mit niedrigerer Kopplung und hoher Kohäsion

`Attribute()` 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.

[1] Diese Änderungen wurden im Commit `https://github.com/Bronzila/WeatherWallpaper/commit/25e8c58a47945383746d8151ed4bfbed01b1d24c` durchgeführt.

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 4.5 ist der Zusammenhang von `IValidationAspekt` 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.

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 4.5 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 ist über das Dependency Inversion Prinzip (siehe Kapitel 4.1.5) 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.

```
// ConfigValidator
public bool ValidateInputs(Config conf)
{
        foreach (var aspect in _validationAspects)
        {
                aspect.Validate(conf);
        }
        return true;
}

// Implementation of IValidationAspect
public class IsCorrectCity : IValidationAspect
{
    public void Validate(Config conf)
    {
        if (!Regex.IsMatch(conf.Location.City, @"^[a-zA-Z -']+$"))
        {
            throw new BadConfigException("Stadtname beinhaltet unbekannte
                Zeichen.");
        }
    }
}
```

Listing 4.5: ConfigValidator als Problemdomäne und IsCorrectCity als Technologiewissen

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 Beschaffungsart keinerlei Schaden beim `ImageHandler` verursachen, da eine klare gemeinsame Schnittstelle mit gegebenen Parametertypen und Rückgabetypen 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 aus-

gewählte 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 HTTPClienten 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[2].

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[3].

[2]Diese Änderung wurden im Commit `https://github.com/Bronzila/WeatherWallpaper/commit/fb4546895d2241b8ea2391991f102bcd8f2fb685` durchgeführt.

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

5 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.

5.1 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 4 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.

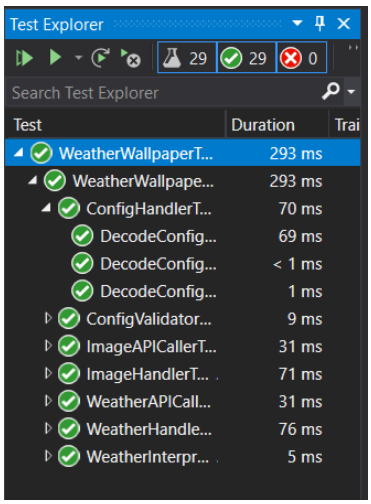

Abbildung 4: Unit-Test Ergebnisse im Test-Explorer

5.2 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 5.1 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.

```
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 5.1: 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 5.2 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.

```
1  [Fact]
2  public void GetImageDataSuccessful()
3  {
4      // Capture
5      var correctApiResponse = new ImageResponse()
6      {
7          Results = new List<Images>{
8              new Images { Links = new Links
9              {
10                 Download = "https://unsplash.com/photos/XxElwSAHOAA/download"
11             }}}
12     };
13     var responseJson = JObject.FromObject(correctApiResponse);
14     var responseJsonString = responseJson.ToString();
15     var api = new Mock<IAPICaller>();
16     api.Setup(caller => caller.Get(It.IsAny<string>()))
17         .Returns(Task.FromResult(responseJsonString)).Verifiable();
18     // Arrange
19     var handler = new ImageHandler(api.Object);
20     string queryString = "?query=doesnt matter";
21     // Act
22     var result = handler.GetImageData(queryString);
23     // Assert
24     Assert.Equal(result.Results.First().Links.Download,
25         correctApiResponse.Results.First().Links.Download);
26     // Verify
27     api.Verify();
28 }
```

Listing 5.2: Unit-Test für den ImageHandler mit Mock

5.3 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 5 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 6 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 5: Code Coverage Ergebnisse

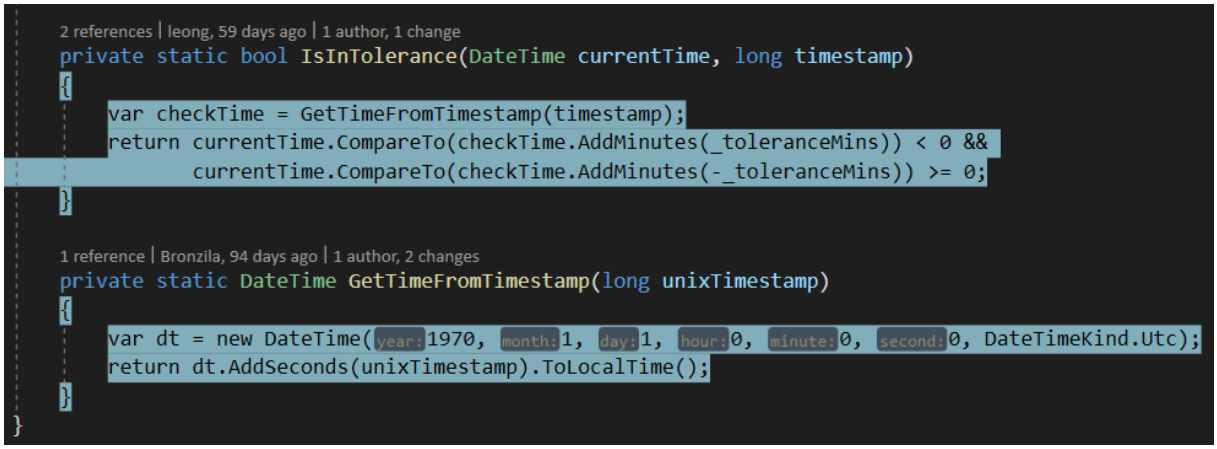

Abbildung 6: Code Coverage Highlighting

6 Refactoring

Bei Refactoring geht es um die Umgestaltung zur Verbesserung der Codequalität, wobei die Gesamtfunktionalität bestehen bleibt. Im Folgenden werden vier Code Smells und Refactoring Methoden angewandt und beschrieben.

6.1 Duplicated Code

Ein bekannter Code Smell ist doppelt vorhandener Code. Dies hat zur Folge, dass dieser Code auch an beiden Stellen immer gleich gewartet werden muss, also auch doppelten Aufwand fordert. Vergisst man hierbei eine der Stellen, sind beispielsweise Sicherheitslücken weiterhin vorhanden. Trotzdem ist dieser Smell sehr einfach zu beseitigen: man muss den Code zusammenführen.

Wie im Kapitel 4.3 bei dem Prinzip von DRY bereits erklärt, wurden die zwei Implementierungen des Interfaces `IAPICaller` zu einer Klasse zusammengefasst. Dies war ebenfalls eine Code-Duplizierung, welche durch den dort referenzierten Commit gelöst wurde.

6.2 Large Class

Dieser Code Smell beschäftigt sich mit der Größe einer Klasse. Bei einer zu großen Klasse kann es häufig sein, dass das SRP von SOLID (siehe Kapitel 4.1.1) verletzt oder die Grundkonzepte bei GRASP (siehe 4.2.1) - Low Coupling und High Cohesion - nicht erfüllt werden. Daher muss dieser Smell untersucht und bei Problemen die Klasse in Teilfunktionalitäten aufgeteilt werden. Bei der Vorstellung des SRP wurde eine solche Aufteilung vorgenommen und bereits genauer erklärt. Zusammengefasst wurde die Klasse `MainController` in vier neue Klassen aufgeteilt, da das SRP nicht mehr erfüllt war. Hierbei wurde in Refresh-Logik, Timer-Verwaltung, manuelle Aktualisierung mit BackgroundWorker und der GUI-Logik unterschieden.

6.3 Rename Method

Dies ist zwar kein eigener „spezieller“ Code Smell, jedoch werden mit dieser Methode unpassende Methodennamen geändert. Eigentlich liegt hier der Smell *Code Comments* zugrunde, also dass Funktionen einen undeutlichen Namen besitzen und unter anderem deshalb Kommentare im Code nötig sind.
In unserem Fall wurde es jedoch angewendet, da zwei eigenständige Methoden unterschiedlicher Klassen zwar in der Praxis ähnliches machen, aber unterschiedlich benannt wurden. Es handelt sich hierbei um die Klassen `ImageHandler` und `WeatherHandler`. Beide müssen mit ihrer Information jeweils einen Route-String zusammenbauen. Hierfür existierte beim `ImageHandler` die Funktion `GetQueryStringFromAttributes(WeatherInterpretation)` und beim `WeatherHandler` die Funktion `BuildRouteString(List<string>)`. Aus Gründen der Übersichtlichkeit und Leserlichkeit wurde für beide der Name `BuildRouteString()` normalisiert[4].

6.4 Error Code

Hierbei handelt es sich um schlechtes Code-Design. Wird bei einem Fehler spezielle Rückgabewerte geliefert, müssen diese bei jedem Aufruf abgeprüft werden. Dadurch existiert nicht nur zusätzlicher Code, sondern es wird nicht in Normalfall und Fehlerfall unterschieden. Hierfür können Exceptions ausgelöst werden. Diese Methode nennt man *Replace Error Code with Exception*

[4]Dieses Refactoring wurde im Commit `https://github.com/Bronzila/WeatherWallpaper/commit/8229814773f5c1a59d02521d873cab393e4536bd` durchgeführt.

In der Klasse `ConfigHandler` wurde beim Lesen und Dekodieren einer gespeicherten Konfiguration `null` zurückgegeben, falls hierbei ein Fehler auftrat. Die alte Version des Lesevorgangs wurde im Codebeispiel 6.1 dargestellt. Falls dies beim Aufruf nicht überprüft und abgefangen wird, können NullPointerExceptions die Ausführung des Programms stoppen.
Daher wurden diese Funktionen nach der Methode überarbeitet; die `BadConfigException()` mit entsprechender Nachricht wird ausgelöst[5]. Dies ist im Listing 6.2 deutlich gemacht.

```
1  public JObject ReadConfigFromFile()
2  {
3      string jsonString = null;
4
5      try
6      {
7          jsonString = _fileAccessor.ReadFile(_configPath);
8      }
9      catch (Exception e)
10     {
11         System.Diagnostics.Trace.WriteLine("Couldnt read config file");
12         System.Diagnostics.Trace.WriteLine(e.Message);
13     }
14
15     JObject conf = string.IsNullOrEmpty(jsonString) ? null : JObject.Parse(jsonString);
16
17     return conf;
18 }
```

Listing 6.1: `ReadConfigFromFile()` mit `null` als Rückgabewert

```
1  public JObject ReadConfigFromFile()
2  {
3      try
4      {
5          return JObject.Parse(_fileAccessor.ReadFile(_configPath));
6      }
7      catch
8      {
9          throw new BadConfigException("Einlesen der gespeicherten Konfiguration " +
10                                       "nicht moeglich.");
11     }
12 }
```

Listing 6.2: `ReadConfigFromFile()` überarbeitet nach Replace Error Code with Exception

6.5 Weitere

Die folgenden Code Smells wurden zwar analysiert, jedoch entweder nicht direkt umgesetzt oder sind nicht existent.

[5]Diese Änderungen finden sich im Commit `https://github.com/Bronzila/WeatherWallpaper/commit/e3367e008dd79b735d30f64282d8edb929810693` wieder.

Long Method Die Long Method, ähnlich wie *Long Class*, bezieht sich auf die Größe einer Methode. Hierbei führen lange Methoden zu schlechterer Lesbarkeit und man spart sich Kommentare. Dies wurde zu teilen bei dem `WeatherHandler` gefunden, welcher die Methode `GetWeatherData(Config)` besitzt (siehe Listing 6.3). Hier könnte man die Zeilen 3-5 zusammengefasst werden als eigene Funktion, welche sich um den Zusammenbau des Routestrings kümmert, wie es beim `ImageHandler` der Fall ist (siehe Listing 4.4).
Diese Klassen haben jedoch ein komplett anderes Problem, und zwar *Duplicated Code*. Ähnlich wie oben bei diesem Code Smell beschrieben, müssten beide Klassen zusammengefasst werden. Dies wurde aus zeitlichen Gründen nicht weiter verfolgt.

```
1  public WeatherResponse GetWeatherData(Config currentConfig)
2  {
3      var routeAttributes = new List<string>();
4      routeAttributes.Add(LocationAsRouteAttribute(currentConfig.Location));
5      string attributes = BuildRouteString(routeAttributes);
6      var response = _api.Get(attributes);
7      response.Wait();
8      var responseAsJObject = JObject.Parse(response.Result);
9      return responseAsJObject.ToObject<WeatherResponse>();
10 }
```

Listing 6.3: `GetWeatherData(Config)`

Shotgun Surgery Dieser Code Smell basiert eigentlich auf anderen Smells. Um eine Änderung durchführen zu können, muss der Code hier an vielen verschiedenen Stellen im Programm geändert werden. Dies wurde von uns nicht eindeutig als solches identifiziert, jedoch kann es im Zusammenhang mit dem, in *Long Method* beschriebenen Problem auftreten.

Switch Statement `Switch`-Anweisungen wurden nicht verwendet.

Code Comments Erklärende Kommentare wurden nur im äußersten Notfall eingesetzt. Es ist keine weitere Reduzierung dieser notwendig.