

# **Programmmentwurf einer Todo-Liste in der Programmiersprache Microsoft C#**

im Kurs

Advanced Software Engineering

vorgelegt von

Michaela Fleig und Mohammad Mehjazi

Matrikel-Nr. 8079678 und 3164982

## Eigenständigkeitserklärung

(gemäß §5(3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. 9. 2015)

Ich versichere hiermit, dass ich meine Projektarbeit) mit dem Thema: „Programmmentwurf einer Todo-Liste in der Programmiersprache Microsoft C#“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 31.05.2021

---

Ort, Datum

Unterschrift

Karlsruhe, den 31.05.2021

---

Ort, Datum

Unterschrift

## Inhaltsverzeichnis

Eigenständigkeitserklärung.....	i
Inhaltsverzeichnis .....	i
Abbildungsverzeichnis .....	ii
1. Bearbeitung der Aufgabenstellung .....	1
1.1. Auswahl geeigneter Methoden .....	1
1.2. Formulierung der verwendeten Algorithmen in einer Programmiersprache...	1
1.3. Testen und Überprüfen der Ergebnisse.....	2
2. Dokumentation des Programms.....	3
2.1. Unit Test .....	3
2.2. Programming Principles .....	6
2.3. Refactoring .....	11
2.4. Clean Architecture .....	15
2.5. Entwurfsmuster.....	19
Quellenverzeichnis .....	22

## Abbildungsverzeichnis

Abbildung 1: Abhängigkeit des Testprojekts zu Modul Adapter.....	4
Abbildung 2: Coverage der Unit Tests.....	5
Abbildung 3: „using“-Direktiven in Visual Studio.....	9
Abbildung 4: UML-Diagramm der Abhängigkeiten der Benutzeroberflächen.....	10
Abbildung 5: Die Architektur nach Vorschlag der Clean Architecture.....	15
Abbildung 6: Beziehungen des Moduls zueinander.....	16
Abbildung 7: Cluster Call Butterfly Ansicht des Moduls „Plugin“.....	17
Abbildung 8: Cluster Call Butterfly Ansicht des Moduls „Kern“.....	17
Abbildung 9: Cluster Call Butterfly Ansicht des Moduls „Adapter“.....	17
Abbildung 10: Abstrakte Ansicht der Aufrufe.....	18
Abbildung 11: Detailliertere Ansicht der Aufrufe.....	18
Abbildung 12: Objekt Referenzen der Instanz myGui1 von Gui1.....	20
Abbildung 13: Mit und ohne Observer Pattern.....	21

## 1. Bearbeitung der Aufgabenstellung

### 1.1. Auswahl geeigneter Methoden

Der Programmentwurf soll eine Anwendung sein, die dem Benutzer in Form einer Todo-Liste behilflich ist. Dabei wird eine größere Menge potenzieller Benutzer angesprochen, die unterschiedliche Erfahrungsstände und Kenntnisse im Bezug zur Nutzung eines Computers haben. Daher sollen die Benutzeroberfläche und Anwendungssteuerung sehr einfach gehalten werden. Damit das Programm auch einem internationalen Publikum ermöglicht werden kann, ist die Benutzeroberfläche in englischer Sprache gehalten.

Da der Fokus innerhalb der begrenzten Arbeitszeit auf der Erstellung der Anwendungsführung und den unterliegenden Algorithmen liegt, wird die Programmiersprache Microsoft C# eingesetzt. Damit ist es über eine Visual Studio Distribution möglich, auf einfache Weise die Benutzeroberfläche zu erstellen. Es wurde die IDE Visual Studio 2019 Community Edition verwendet, das eingebundene Ziel-Framework ist .NET Core 3.1, um das Programm als eine Windows-basierte Anwendung zu kompilieren. Visual Studio ermöglicht die Erstellung einer .exe-Datei, die auf jedem Betriebssystem mit der entsprechend installierten .NET Core Version oder einer dazu kompatiblen Version funktionsfähig ist.

### 1.2. Formulierung der verwendeten Algorithmen in einer Programmiersprache

Die Anwendung ermöglicht dem Benutzer über eine Schaltfläche „To Do!“ mehrere Todo-Listen zu erstellen mit jeweils einer aktuell fixen Anzahl von Unterpunkten. Diese können über eine Eingabefläche „Calendar“ mit einem Fälligkeitsdatum, einem Namen, einer optionalen Beschreibung und einer Markierung, dass der zu erstellende Termin als Wecker gestellt werden soll, mit Inhalt befüllt werden. Der Wecker kontrolliert dabei bei jedem Programmaufruf, ob der Termin bereits eingetroffen ist. Falls ja, wird der Benutzer über eine Schaltfläche mit dem Namen, der optionalen Beschreibung und dem Datum erinnert.

Die beiden zeitlich aktuellen Unterpunkte werden nur mit ihrem Namen unter der Bezeichnung „Upcoming Events!“ übersichtlich dargestellt. Unter „Task Highlights“ werden die Termine dargestellt, die über die initiale Ansicht im

entsprechenden Todo-Tab als wichtig markiert werden können. Auch hier ist eine begrenzte Ansicht von zwei Elementen möglich.

Die initiale Ansicht enthält die Übersicht über die offenen Tabs und der darin enthaltenen Unterpunkte, mit Highlight-Markierung, Name und fälligem Datum, nach zeitlicher Erstellung sortiert. Der Benutzer kann sich auch über eine Schaltfläche „Log In!“ einloggen. Dabei wird sein Windows-Benutzerkontenname und –bild verwendet. Hier steht dem Benutzer nun die Möglichkeit zur Verfügung, den Entwicklern Feedback zu geben. Über dieselbe Schaltfläche, die nun „Log Out!“ heißt, kann sich der Benutzer wieder abmelden. Über die Schaltfläche „X“ kann das Programm vollständig geschlossen werden. Die erstellten Unterpunkte werden beim nächsten Starten der Anwendung wieder geladen.

Nachdem zunächst verschiedene Ansätze vorhanden waren, in welchem Speicherformat die Speicherung der Eingabedaten erfolgen soll (Excel, standardisiertes Format wie JSON oder Datenbank-Format) wurde auf einen einfachen Ansatz zurückgegriffen. Mit der nicht-standardisierten Speicherung in eine Textdatei werden auf Wrapper-Methoden zugegriffen, die über einfache Aufrufe einen schnellen Zugriff auf die Textdatei ermöglichen.

Die Anforderung an das Projekt sind 20 Klassen. Diese konnten durch geschickte Programmierung auf 17 Klassen reduziert werden. Unnötig eingeplante Klassen wurden entfernt, da diese in der aktuellen Projekt-Version für die geforderten Funktionalitäten nicht benötigt werden.

### **1.3. Testen und Überprüfen der Ergebnisse**

Die Anwendung soll durch Unit Tests auf seine Funktionalität überprüft werden. Über eine anschließende Code Coverage Analyse wird die Menge der getesteten Funktionen im Hinblick auf den gesamten Code überprüft und gibt einen Indikator an, wie viel bereits (erfolgreich) getestet wurde.

## 2. Dokumentation des Programms

### 2.1. Unit Test

Durch das Testen einzelner Funktionsblöcke wird eine korrekte Ausführung dieser sichergestellt. Diese Eigenschaft darf nicht durch das Verschieben von Code-Teilen durch beispielsweise Refactoring verloren gehen. Unit Tests stellen auf einfache Art und Weise sicher, dass die Methoden weiterhin ihre Funktionalität behalten, auch wenn sie in ihrer semantischen Darstellung verändert werden. Ein Unit Test ist für die Überprüfung der Funktion einer Methode zuständig. Der Unit Test wird idealerweise vor der Erstellung der eigentlichen Methode geschrieben. Er wird nach dem Standard-Pattern AAA (Arrange, Act und Assert) geschrieben. Dieses Muster wird auch ATRIP genannt. Jede dieser Teilkomponenten ist für den nach ihr benannten Bereich der Testmethode verantwortlich.

Die Unit Tests werden mit dem Test Framework xUnit durchgeführt. Die zehn Tests sind mit Github-Tag b15a921 hochgeladen.

xUnit.net ist ein kostenloses Open-Source-Community-Testtool für .NET Framework.

XUnit.net wurde vom ursprünglichen Erfinder von NUnit v2 geschrieben und ist die neueste Technologie zum Testen von C#, F#, VB.NET und anderen .NET-Sprachen. Außerdem kann xUnit.net mit ReSharper, CodeRush, TestDriven.NET und Xamarin verwendet werden.

Nachfolgend sollen zwei Unit Tests am Beispiel des Programmentwurfs dargestellt werden. Dem anschließend werden die Referenzen zu den restlichen acht Unit Tests stehen.

`SplitDataFromStorageName`

Die Testmethode testet die entsprechend gleichnamige Methode im Projekt „LISTED“, das in der Testklasse Gui1\_Test das Modul Plugin referenziert. Über Arrange werden String-Variablen und Objekte angelegt. Über Act wird die Methode `SplitDataFromStorageName` mit den zuvor erzeugten Variablen und Objekten aufgerufen. Assert überprüft den zu erwartenden Wert der in Arrange erstellt wurde mit dem Rückgabewert des Methodenaufrufs in Act. (Github-Tag b15a921).

## StorageDate

Die Testmethode testet ebenfalls die gleichnamige Methode im Projekt, die in der Klasse `Storage_Test` als `xUnit-Tag[Fact]` markiert das Modul `Plugin` referenziert. Über `Arrange` werden verschiedene Zeiten im Format `DateTime` angelegt. In `Act` wird einer der erstellten Werte in den entsprechenden Wert der Klasse `Storage` geschrieben und in `Assert` einmal auf seine Gleichheit mit dem erwarteten Wert und einmal auf seine Ungleichheit getestet. Als Speicher soll die Klasse `Storage` auch auf Ungleichheit überprüft werden, damit keine falschen Werte oder Formate gesichert werden. (Github-Tag `b15a921`).

Am Beispiel der Schicht `Adapter` soll in Abbildung 1 grafisch die Abhängigkeit des Testprojektes zum Hauptprojekt „LISTED“ dargestellt werden.

`Adapter` wird von `LISTED_Testing` für die Tests und von `Kern` für Funktionalitäten referenziert. `Adapter` referenziert selbst `Plugin` und die `DLL Forms`, die generalisierte Klassen für die Benutzeroberflächen beinhaltet.

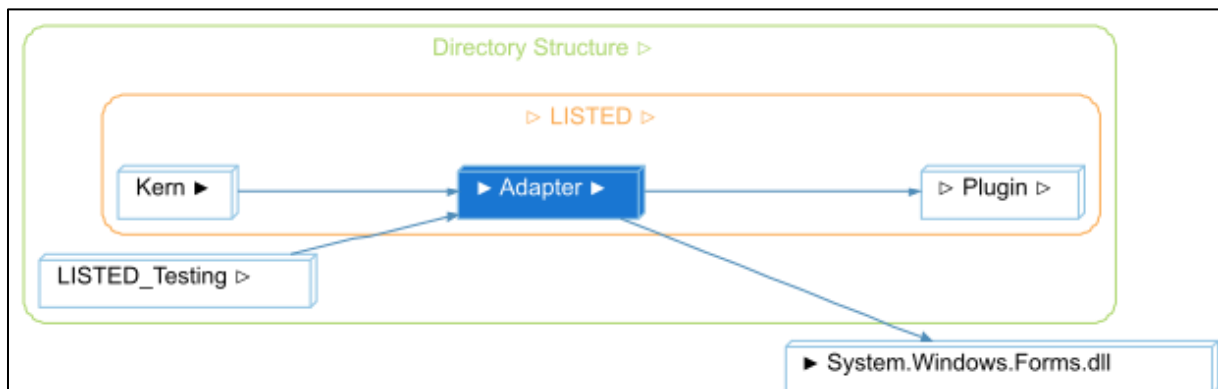


Abbildung 1: Abhängigkeit des Testprojekts zu Modul `Adapter`

Die Benutzeroberfläche kann nur mit dem automatisierten UI Test Framework getestet werden, welches kostenpflichtig in der Professional Version von Visual Studio angeboten wird. Daher soll direkt auf die benötigten Instanzen der Klasse `RichTextBox` zugegriffen werden.

## Mocks

Für Tests ohne Abhängigkeiten sind die Tests basierend auf dem AAA-Prinzip durchgeführt, mit Abhängigkeiten werden sie zunächst mit Mocks zu realisieren versucht, konnte aber nicht beendet werden, da es keine Möglichkeit gab, eine Setup-



Methode in xUnit zu verwenden, welche die Werte des Mocks einschränken. Stattdessen wird direkt auf eine Datei zugegriffen, auf die die Methoden getestet werden.

Mit den beiden Methoden `v_OutputHandler_ReceiveDataStream` und `int_IoDatastorage_WriteDataToStorage` des Unit Test Projektes war der Versuch eines Mocks gegeben. Dabei sollen die Methoden den Zugriff auf eine externe Datei, die als Datenablage für User-Eingaben dient, nachgebildet werden, ohne auf die eigentliche Datei zuzugreifen.

Das Mocken konnte leider nicht erfolgreich durchgeführt werden, da die Interfaces nicht vom Framework als solche erkannt wurden (Github-Tag b15a921).

## Test Coverage

Code Coverage-Ergebnisse				
Asus_DESKTOP-PL8REKR 2021-05-28 19_48_...				
Hierarchie	Nicht abgedeckt (Blöcke)	Nicht abgedeckt (% Blöcke)	Abgedeckt (Blöcke)	Abgedeckt (% Blöcke)
Asus_DESKTOP-PL8REKR 2021-0...	764	46,53 %	878	53,47 %
└─ listed.dll	762	48,97 %	794	51,03 %
└─ LISTED.Plugin	115	100,00 %	0	0,00 %
└─ Test_LISTED	438	36,44 %	764	63,56 %
└─ Gui1	436	36,33 %	764	63,67 %
└─ Program	2	100,00 %	0	0,00 %
└─ Test_LISTED.Controller	11	42,31 %	15	57,69 %
└─ ControlGui1	4	100,00 %	0	0,00 %
└─ Init	5	100,00 %	0	0,00 %
└─ InputHandler	0	0,00 %	9	100,00 %
└─ OutputHandler	2	25,00 %	6	75,00 %
└─ Test_LISTED.Kern	19	100,00 %	0	0,00 %
└─ Tab	2	100,00 %	0	0,00 %
└─ TabsControl	17	100,00 %	0	0,00 %
└─ Test_LISTED.Model	18	54,55 %	15	45,45 %
└─ IoDatastorage	4	25,00 %	12	75,00 %
└─ Storage	3	50,00 %	3	50,00 %
└─ UserLogin	11	100,00 %	0	0,00 %
└─ Test_LISTED.View	161	100,00 %	0	0,00 %
└─ Gui2	161	100,00 %	0	0,00 %
└─ listed_testing.dll	2	2,33 %	84	97,67 %
└─ LISTED_Testing	2	2,33 %	84	97,67 %
└─ Gui1_Test	0	0,00 %	48	100,00 %
└─ IoHandler_Test	0	0,00 %	6	100,00 %
└─ IoHandler_Test_Mocks	0	0,00 %	26	100,00 %
└─ Storage_Test	2	33,33 %	4	66,67 %

Abbildung 2: Coverage der Unit Tests

Abbildung 2 zeigt die Abdeckung der erstellten zehn Tests auf dem Projekt „LISTED“. Die Coverage der Unit Tests wurde mit dem in Visual Studio Enterprise 2019

enthaltenen Analyseprogramm „Code Coverage“ erstellt. Jede Testmethode stellt hier eine getestete Methode des Gesamtprojektes dar.

### 2.2. Programming Principles

Programming Principles stellen als Design Prinzipien eine gute Basis für sauberen Code in Softwaresystemen unterschiedlicher Größe dar. Dabei gibt es unter anderem die SOLID-Prinzipien, die angeben, wie Funktionen und Datenstrukturen in Klassen sortiert werden und wie diese Klassen untereinander kommunizieren sollen. Dabei stellt eine Klasse eine Gruppierung von Funktionen und Daten dar. Ziele dieser Prinzipien sind die Toleranz bei Veränderungen am Code, das vereinfachte Verständnis des Codes und das Entstehen einer Basis von Komponenten, die von vielen Softwaresystemen verwendet werden können. Als Mid-Level-Software helfen sie Programmierern, die auf Modul Ebene arbeiten, indem sie auf Code-Ebene eine mögliche Struktur mit Modellen und Komponenten definieren. Die SOLID-Prinzipien beschreiben, wie einzelne Bereiche der Außenform zueinanderstehen müssen. Das beschreibt nicht das Zueinanderstehen der Bereiche innerhalb des Systems. Diese werden von den Komponentenprinzipien beschrieben, auf die hier nicht weiter eingegangen werden soll. [1]

#### SRP

So wird das Single Responsibility Principle (SRP) als logische Folgerung aus Conways Gesetz verstanden und im Programmentwurf realisiert. Das hat zur Folge, dass jedes Modul nur einen Grund hat, verändert zu werden. Die Struktur ist von der Kommunikationsstruktur, der sie nutzenden Organisation beeinflusst.

Als Beispielimplementierung können die Methoden `SplitDataFromStorageName` und `SplitDataFromStorageDate` aus Klasse `Gui1` gesehen werden. Beide Methoden haben jeweils nur eine Aufgabe, einmal den Namen und einmal das Datum aus dem Datenstrom zu extrahieren. Zunächst waren beide Funktionalitäten sequenziell nacheinander in einer Methode `Button1_Click` zu finden (Github-Tag `f5d1184`). Nach Anwendung des SRP-Prinzips wurden unter anderem diese Methoden extrahiert (Github-Tag `54c15af` ).

Das Prinzip legt die Grundlage für nachfolgenden drei Prinzipien dar. Ein Modul darf nur von einem Aufrufenden verwaltet werden. Das bedeutet nicht, dass eine Funktion

nur eine Funktionalität haben darf (wird beim Refactoring verwendet). Das Prinzip bildet eine Änderungsachse, die für die Architekturgrenzen zuständig ist. [1]

### **OCP**

Das Open Closed Principle (OCP) ist ein erstrebenswerter Ansatz der Softwareprogrammierung. Bei jeder Veränderung am bisherigen Code durch beispielsweise Hinzufügen einer Eingabefläche muss bestehender Code mindestens dahingehend verändert werden, dass die Eingabefläche nach Inhalt der Speicher-Datei initial leer oder befüllt sein muss.

Mit der Verwendung des .NET Frameworks und der Möglichkeit, Benutzeroberflächen zu erstellen, muss bei jedem Hinzufügen eines neuen Elements der UI der bestehende Code angepasst werden. Da dies das Framework übernimmt, soll an dieser Stelle nicht weiter darauf eingegangen werden.

Durch das Trennen von Funktionalitäten soll das OCP-Prinzip weiterhin unterstützt werden. Beispielsweise durch die Methoden `btnTodo_Click` oder `btnHome_Click` werden die Aufgaben mit dem Hintergrund getrennt gehalten, dass bei Hinzufügen oder Entfernen der sie aufrufenden Methoden nur die jeweilige Methode gelöscht werden muss.

Das Software-Artefakt soll für Erweiterungen offen, jedoch für Änderungen am bestehenden Code verschlossen sein. Die Architektur kann als versagt deklariert werden, wenn bei Änderungen große Änderungen an der gesamten Software vorgenommen werden müssen. Hier ist die Reihenfolge der Abhängigkeiten entscheidend und wirkt als Schutzmechanismus einer Komponente vor einer zu ändernden Komponente. Das Ziel ist hierbei, ein System einfach erweitern zu können, ohne große Änderungen am gesamten System vornehmen zu müssen. Dazu muss das System in Komponenten zerlegt werden, die in einer Hierarchie der Abhängigkeiten eingeordnet werden. [1]

### **LSP**

Das Liskov Substitution Principle (LSP) legt die Definition von Subtypen fest. Zur einfacheren Wartbarkeit des Softwaresystems sollen seine Teile veränderlich und austauschbar sein. Dabei soll ein Teil möglichst unabhängig von anderen Teilen arbeiten können.

Die Verwendung des objektorientierten .NET-Frameworks stellt das LSP dar. Als Beispiel können hier die Erstellung von Benutzeroberflächen (Gui1, Gui2, GuiAlarm) und die Erstellung von Objekten nach selbst definierten Klassen (Storage, UserLogin) genannt werden.

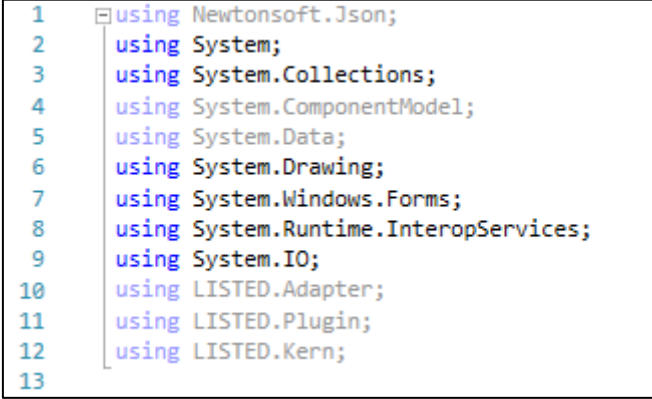
So stellt jedes Objekt der Benutzeroberflächen eine Instanz einer Klasse des Frameworks dar. Diese Instanzen des Softwaresystems sind dadurch variabel und austauschbar und ermöglichen eine gute Wartbarkeit.

Das Prinzip kann architekturweit ausgebreitet werden. Es steht als Leitlinie für die Vererbung und als Prinzip für das Software-Design eines Systems in Bezug auf Interfaces und Implementierungen. Ein Nutzer ist von einer vererbten Implementierung oder einem Interface abhängig. [1]

### **ISP**

Das Interface Segregation Principle (ISP) beschreibt das Entfernen nicht genutzter Teile und die auf sie zeigenden Abhängigkeiten. Es sollen nur Bereiche referenziert werden, die von der aufrufenden Methode verwendet werden können. Methoden sollen nicht an einem gemeinsamen Ort gelagert werden, nur in den Modulen, die die gemeinsamen Parameter benötigen. Das bedeutet für die Architektur, dass nur Module eingebunden werden sollen, die wirklich für die Funktionalität des Moduls gebraucht wird. [1]

Durch die Verwendung der Entwicklungsumgebung Visual Studio werden nicht verwendete Verweise auf weitere Dateien entsprechend gekennzeichnet. Abbildung 3 stellt diese Markierung dar. Nicht verwendete Verweise der „using“-Direktiven werden weniger kontrastreich angezeigt.



```
1  using Newtonsoft.Json;
2  using System;
3  using System.Collections;
4  using System.ComponentModel;
5  using System.Data;
6  using System.Drawing;
7  using System.Windows.Forms;
8  using System.Runtime.InteropServices;
9  using System.IO;
10 using LISTED.Adapter;
11 using LISTED.Plugin;
12 using LISTED.Kern;
13
```

Abbildung 3: „using“-Direktiven in Visual Studio

Dies hilft, nicht verwendete Abhängigkeiten zu entfernen und das ISP-Prinzip erfolgreich anzuwenden. Damit werden keine Dateien referenziert, die möglicherweise eine Klasse oder Methode derselben Bezeichnung aufweisen und somit das Build-System sowie den Entwickler verwirren können (Github-Tag 11fc788).

ISP beinhaltet weitergehend auch die Entfernung von Abhängigkeiten des Projektes. Es sollen nur DLLs, Frameworks und Pakete eingebunden werden, die auch verwendet werden.

### **DIP**

Das Dependency Inversion Principle (DIP) beschreibt die Trennung der Abhängigkeiten von High-Level-Policies zu Low-Level-Policies. Das bedeutet, Details sollen von Policies abhängig sein, die Generalisierung einer Funktionalität ist nicht von ihrer konkreten Implementierung abhängig. Dieses Prinzip soll mit der Clean Architecture realisiert werden.



Das Prinzip beschreibt, dass Abhängigkeiten im Code nur auf Generalisierungen zeigen sollen, nicht auf Konkretisierungen. Eine stabile Architektur vermeidet Abhängigkeiten an flüchtige Konkretisierungen und verwendet stattdessen stabile, abstrakte Interfaces. Dennoch können DIP Abhängigkeiten nie ganz eliminiert werden, diese Menge sollte jedoch minimiert und vom restlichen System getrennt werden. Dieses Prinzip ist das am besten sichtbare Organisationsprinzip einer Architektur. Abhängigkeiten dürfen nur in eine Richtung, vom Konkreten zum Abstrakten führen. [1]

### **2.3. Refactoring**

Refactoring verändert das Programm in kleinen Schritten, dass wenn ein Fehler passiert, ist es einfach, den Bug zu finden. Das Refactoring hilft interne Änderungen der internen Struktur des Systems zu verstehen und günstiger zu verändern, ohne sein beobachtbares Verhalten zu ändern. Das Verhalten soll in mehreren kleineren Teilschritten zur Erhaltung des Verhaltens möglich sein, wobei zu jedem Zeitpunkt ein Beenden des Refactoring möglich sein soll. Refactoring wird als das Säubern und Reorganisieren der Codesbasis verstanden und ähnelt einer Optimierung der Performance. Refactoring ist kein Zusatz von Funktionalität, sondern soll im ständigen Wechsel mit der Funktionalität angewandt werden. Durch Refactoring kann die Software auf aktuelle Nöte passend ausgelegt werden. Sobald Änderungen notwendig und verstanden werden, kann die Architektur durch Refactoring an die Neuerungen angepasst werden. Auch zusätzliche Parameter, die für die aktuelle Softwareversion nicht verwendet werden, können identifiziert und aus dem System entfernt werden. Dabei werden die Prinzipien „Simple Design“, „Inkrementelles Design“ und „Yagni“ (engl. „you ain’t going to need it“, „Du wirst es nicht brauchen.“) verwendet. Diese basieren auf dem Verständnis, dass mit einem Problem besser umgegangen werden kann, wenn es später besser verstanden habe. Aktuelle Schwierigkeiten und ungenutzten Code können durch das inkrementelle Refactoring-Verfahren entfernt werden. Auch die Performance kann durch Refactoring erhöht werden, wenn die Architektur oder Teilbereiche überdacht und umgeschrieben werden. Ein klarer Code erleichtert das Hinzufügen von Features und dem Programmierer sind offene Optionen offensichtlicher und verständlicher. Das Refactoring kann durch die Entwicklungsumgebung unterstützt werden, wie unter der eingesetzten Visual Studio

2019 Community Edition. Diese unterstützt automatische und manuelle Refactorings.  
[2]

Nachfolgend sollen beispielhaft durchgeführte Refactorings beschrieben werden, um die prinzipielle Vorgehensweise darzustellen.

### **Refactoring**

Ausgangsversion: f5d1184

Viele if-Bedingungen, um jede Textbox auf ihren aktuellen Inhalt hin zu überprüfen und zu beschreiben. Danach wird der Name und das Datum des Termins formatiert und in eine Datenablage geschrieben. Es soll überprüft werden, ob das Datum des Termins mit dem heutigen Datum übereinstimmt. Falls ja, wird die nächste freie Textbox der Gruppe „Upcoming Events!“ mit dem Namen des Termins beschrieben.

Es existieren sehr viele ähnliche Bedingungen, die sich inhaltlich nur gering voneinander unterscheiden.

Schritt-Version: 9c11979

Über die Extract Method werden lange Funktionen kleiner gemacht, indem eine mehrfache Verwendung derselben Funktionalität durch Decomposition aufgelöst wird.

Es werden die if-Bedingungen extrahiert und über einen Switch-Case über Count gelöst. Count wird hier noch immer in der Bedingung selbst verwendet. Weitere Bedingungen sind noch offen.

Schritt-Version: e076a78

Der Code aus der if-Bedingung zur Variablen Count wird gelöscht. Auch die Debug-Anweisungen werden gelöscht, sie werden hier nicht mehr benötigt.

Es ist noch viel doppelter Code vorhanden. Da die Funktionalität so an keiner weiteren Stelle verwendet wird, soll kein unnötiger Code für mögliche zukünftige Anwendungsfälle geschrieben werden (YAGNI). Aktuell ist dies die beste Lösung. [2]

Schritt-Version: 59c9d9f

Es wird offensichtlich nicht benötigter Code gelöscht. Es existieren noch immer Bedingungen mit ähnlichen Abfragen. Im vorherigen Schritt wurde beschrieben, dass diese Bedingungen nicht extrahiert werden. Das angewendete YAGNI-Prinzip



beschreibt auch, dass zu einem späteren Zeitpunkt die Situation mit veränderten Umständen betrachtet, doch zu einem Refactoring führen kann. Über das Refactoring-Prinzip Extract Method sollen diese Bedingungen extrahiert und mit Übergabeparameter an diese neue Methode gesteuert werden.

Schritt-Version: a6c65ed

Es werden weitere if-Bedingungen extrahiert und über Remove Variable können Methoden direkt aufgerufen werden, ohne zusätzliche Variablen zu verwenden.

Schritt-Version: each19

Über eine erneute Anwendung von Remove Variable werden Variablen nur in ihren Anwendungsfällen bekannt. Das bedeutet, nur in der extrahierten Methode soll die Variable existieren.

Schritt-Version: 883b515, a724813

Auch hier wird jeweils das Prinzip Remove Variable angewandt.

Schritt-Version: c95225c

Die Reihenfolge der if-Statements wird dahingehend verändert, dass der Code beim Build möglichst schnell durchlaufen wird.

Schritt-Version: f41042f

Nun wird das Prinzip Extract Method angewandt.

Schritt-Version: be62f25

Als weiteren Schritt des Refactorings werden dem Code Kommentare hinzugefügt. Das erleichtert das Verständnis für jeden Leser des Codes. Da der Kommentar recht kurzgehalten ist, ist hier kein Code Smell zu den Kommentaren als Dispensable (zu Entbehrendem) vorhanden.

Finale Schritt-Version: a881f0c

In der finalen Version wurde noch ein Rename Method als Teil der Simplifying Method Calls angewandt, indem ein veränderter Name gewählt wurde.

Ausgangsversion: 1b20c72

Als weiteren Schritt des Refactorings werden dem Code unnötige Kommentare gelöscht, um die Verwirrung zu verhindern.

Schritt-Version: ce14b3b

Hier werden unbenutzte Variablen gelöscht, um den Code im kompakten Zustand zu beibehalten.

Schritt-Version: 2e6a092 und Schritt-Version: 54c15af

Als nächstes werden hier auch ein paar lange Funktionen mit dem „Extract Method“ kleiner gemacht, indem wie Vorhin eine mehrfache Verwendung derselben Funktionalität durch Decomposition aufgelöst wird.

Schritt-Version: 42f9436

Der Code aus der if-Bedingung zur ReadDataFromStorage wird in catch-Block umgewandelt.

Finale Schritt-Version: 544bc03

Auskommentierter nicht genutzter Code-Block wird gelöscht.

### **Code Smells**

Als Code Smells werden Bereiche im Code bezeichnet, die auf unschönen Code hinweisen, der möglicherweise refactored werden soll. Es gibt beispielsweise die Gruppen der Bloaters, die den Code „aufblasen“ und unleserlich machen. Es gibt aber auch die Change Preventers, die Änderungen am Quellcode erschweren.

Das Projekt beinhaltet beispielsweise die Methode `load_Database` der Klasse `Gui1`. Diese enthält zu wenige Kommentare. Durchsprechende Namen müssen hier keine weiteren Kommentare eingefügt werden.

Ein weiteres Beispiel für die Identifizierung des Code Smells Shotgun Surgery, die durch Extract Method angewandt werden konnte. Dazu wurde der User-Login-Teil in eine eigene Methode der Klasse `UserLogin` extrahiert. Diese enthält nun nur den Aufgabenbereich des Logins.

## 2.4. Clean Architecture

Die entwickelte Anwendung kann anhand der nachfolgenden Abbildungen dargestellt werden.

Abbildung 5 zeigt die Darstellung der Anwendung nach Einteilung in drei entsprechende Schichten der Clean Architecture „Application-Code“, „Adapter“ und „Plugin“.

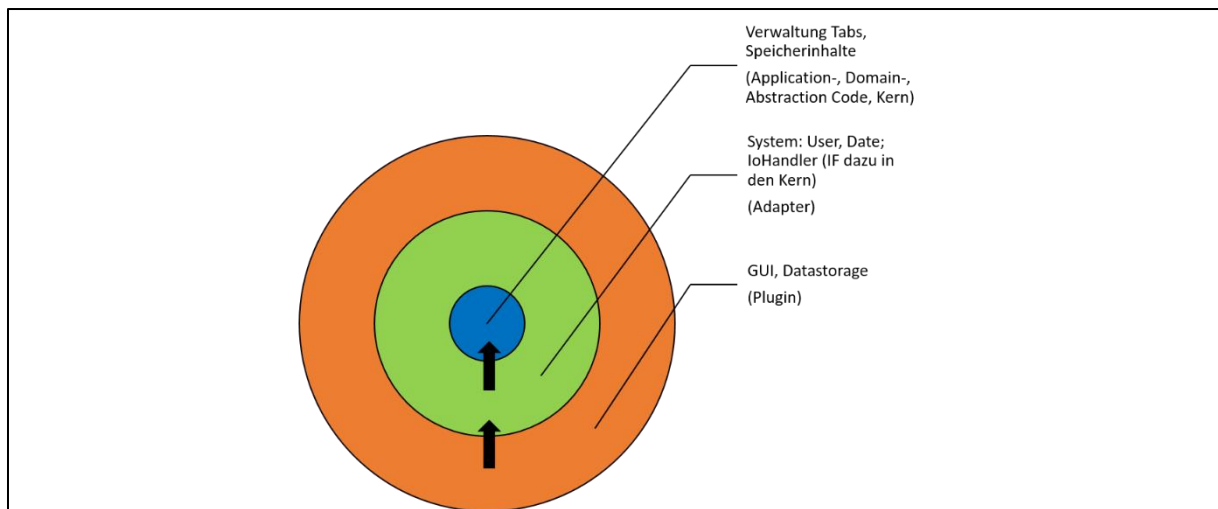


Abbildung 5: Die Architektur nach Vorschlag der Clean Architecture

Die Abhängigkeiten zwischen den Schichten sollen entsprechend der gewählten Architektur-Vorgabe von außen nach innen gewählt werden (Dependency Inversion). [1]

Abbildung 6 zeigt die Abbildung des entwickelten Systems mit Beziehungen zueinander. Die Beziehungen als Pfeile mit gegebener Richtung dargestellt. Es fällt auf, dass die Beziehungen zwischen „Adapter“ und „Plugin“ nicht in eine gemeinsame Richtung verlaufen. Das bedeutet, die Clean Architecture ist nicht gegeben.

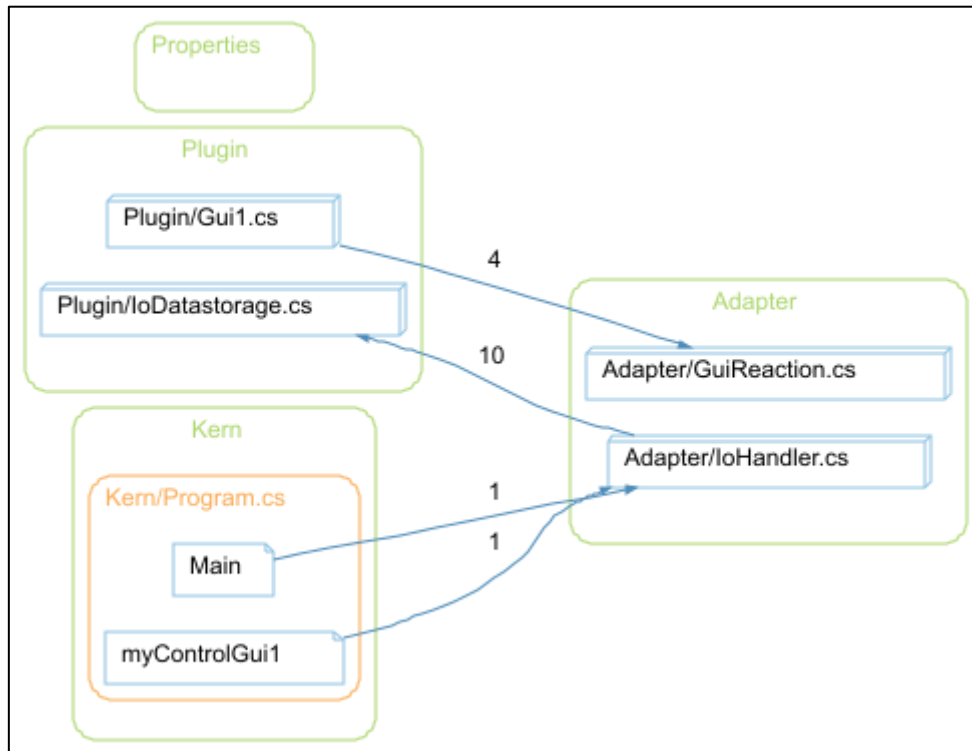


Abbildung 6: Beziehungen des Moduls zueinander

Darauf aufsetzend soll die Richtung des Pfeils zwischen „Plugin/Gui.cs“ und „Adapter/GuiReaction.cs“ invertiert werden, sodass „Adapter“ auf „Plugin“ zugreift.

Dazu wurden entsprechende Schritte in Github-Tag „fa61096“ vorgenommen.

Abbildung 7 stellt die Richtung der Aufrufe zwischen den beiden Schichten „Adapter“ und „Plugin“ dar. Die Pfeile zeigen von „Adapter“ zu „Plugin“. Die Abhängigkeiten können entsprechend [1] invertiert zum Kontrollfluss (Call) betrachtet werden.

Abbildung 8 stellt die Richtung der Aufrufe zwischen den Schichten „Kern“ und „Adapter“ dar. Die Schicht „Kern“ bezeichnet die innerste Schicht „Applikation-Code“.

Wie bereits beschrieben werden die Richtungen des jeweiligen Kontrollflusses invertiert zu den Abhängigkeitsflüssen. Dasselbe gilt für den roten Pfeil in Abbildung 9.

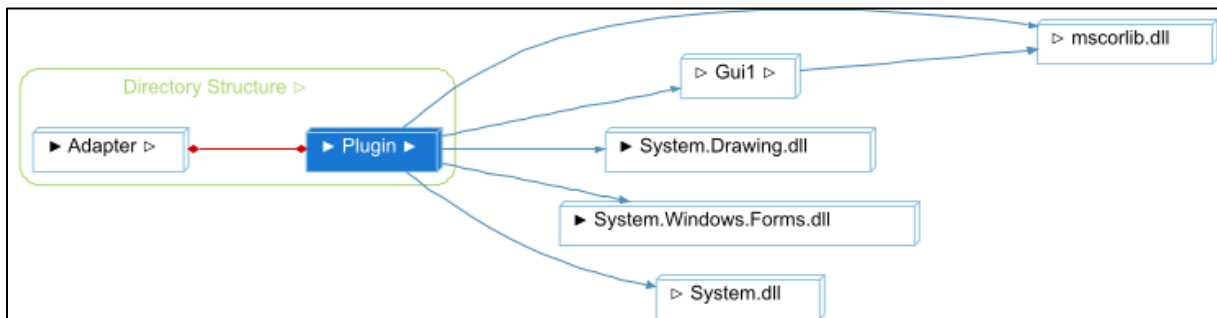


Abbildung 7: Cluster Call Butterfly Ansicht des Moduls „Plugin“

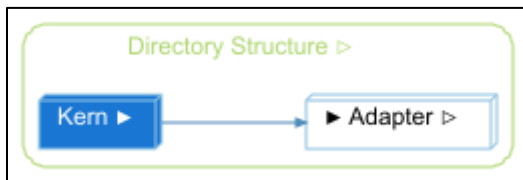


Abbildung 8: Cluster Call Butterfly Ansicht des Moduls „Kern“

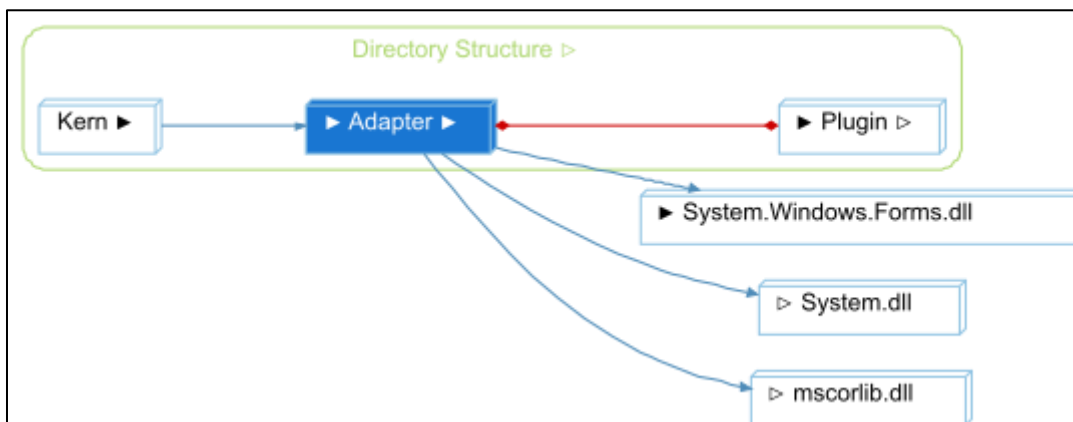


Abbildung 9: Cluster Call Butterfly Ansicht des Moduls „Adapter“

Die vorgenommenen Schritte ergeben die Abhängigkeiten, die in Abbildung 10 abstrakt dargestellt sind.

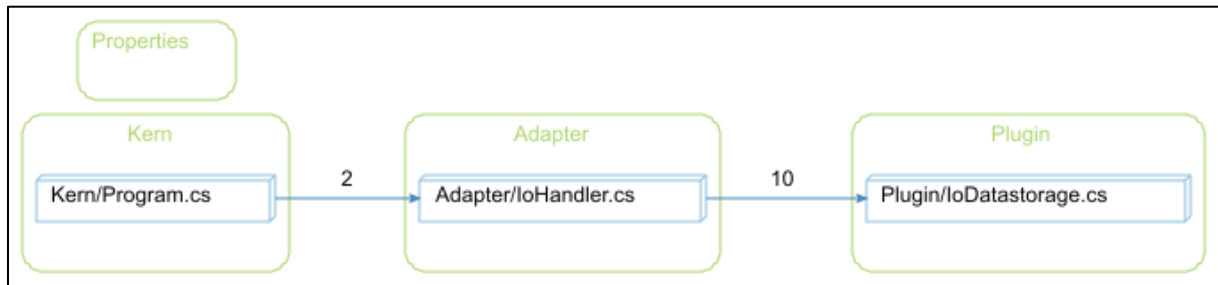


Abbildung 10: Abstrakte Ansicht der Aufrufe

Aus Abbildung Abbildung 11 wird ersichtlich, dass die strukturelle Komplexität nach der McCabe-Metrik (zyklische Komplexität) sehr gering ist. Mit einer Maßzahl von eins oder zwei sind die implementierten Aufrufe sehr einfach gehalten. [3]

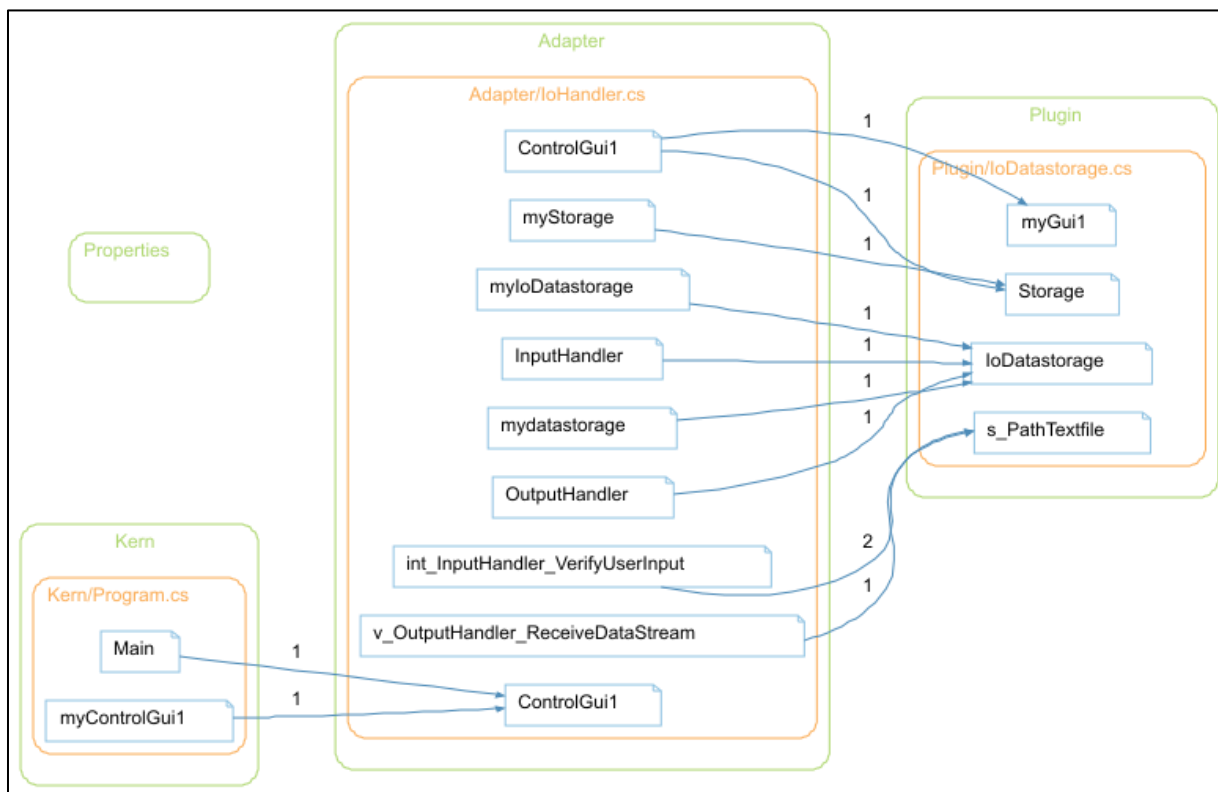


Abbildung 11: Detailliertere Ansicht der Aufrufe

Github-Tag 5c538b7 zeigt die Entwicklung des Projekts in UML-Diagrammen. Bei „...vorher“ wurde Tag b85206e verwendet. Nach diesem Tag (ab 437a6b9) wurde die Clean Architecture realisiert. Mit „ArchInternalDependencies\_Vorher.png“ wird die Notwendigkeit für eine klare Architektur mit vorgegebener Informationsfluss-Richtung und Abhängigkeit dargestellt. Diese befinden sich in den Ordnern „\Diagramme\UML\_Gesamt\_vorher“ und „\Diagramme\UML\_Gesamt\_nachher“.

## 2.5. Entwurfsmuster

Entwurfsmuster beschreiben eine verwendete Mikroarchitektur im Softwaresystem. Auf dieser basierend kann die im vorherigen Kapitel beschriebene Makroarchitektur, die Architektur des vollständigen Systems abstrakt dargestellt werden. [4]

Der Beobachter (engl. Observer) wird zur Weitergabe von (Status-)Änderungen an einem Objekt an abhängige Strukturen verwendet. Der aktuelle Status des Subjekts wird dabei an den Beobachter weitergegeben, damit dieser nicht zyklisch nach Änderungen des Status fragen muss. [5]

Für das Entwurfsmuster soll zunächst ein Observer verwendet werden. Er überwacht bestimmte Elemente der Benutzeroberfläche.

Bisher werden angelegte Termine, „Todos“ über Gui2 vom Benutzer erstellt und über „Speichern“ auf Gui2 übernommen. Mit dem Klicken auf die Schaltfläche „Save“ auf Gui1 (Hauptansicht) wird der gespeicherte Termin in eine der TextBoxen auf Gui1 eingetragen, damit für den Benutzer ersichtlich und in die Datenbank gespeichert.

Um die Nutzungsqualität der Anwendung für den Benutzer zu steigern, sollen Aufgaben intuitiv und effektiv zu erfüllen sein. [6]

Um die genannte Anforderung zu erfüllen, soll das Observer Pattern eine Möglichkeit geben, den zusätzlichen Schritt über die Schaltfläche „Save“ zu entfernen und die Eingabe direkt mit dem ersten „Speichern“ auf Gui2 zu sichern.

Dazu müssen wenige Änderungen in die Architektur eingebracht werden, vgl. dazu Github-Tag „25e7b88“ bzw. „550ca41“.

Die Instanz von Gui1 aus der Klasse IoHandler wird in die Klasse Storage verschoben, damit auch Gui2 darauf zugreifen kann. Dies ist notwendig, damit die ausführende Methode für den Observer `btn_Speichern_Click` auf die Methode `btn_Save_Click` der Instanz von Gui1 zugreifen kann. Damit kann die Eingabe nach dem Anlegen eines Termins in Gui2 über die Schaltfläche „Speichern“ gesichert und angezeigt werden, ohne erneut eine Speicherung zu verlangen.

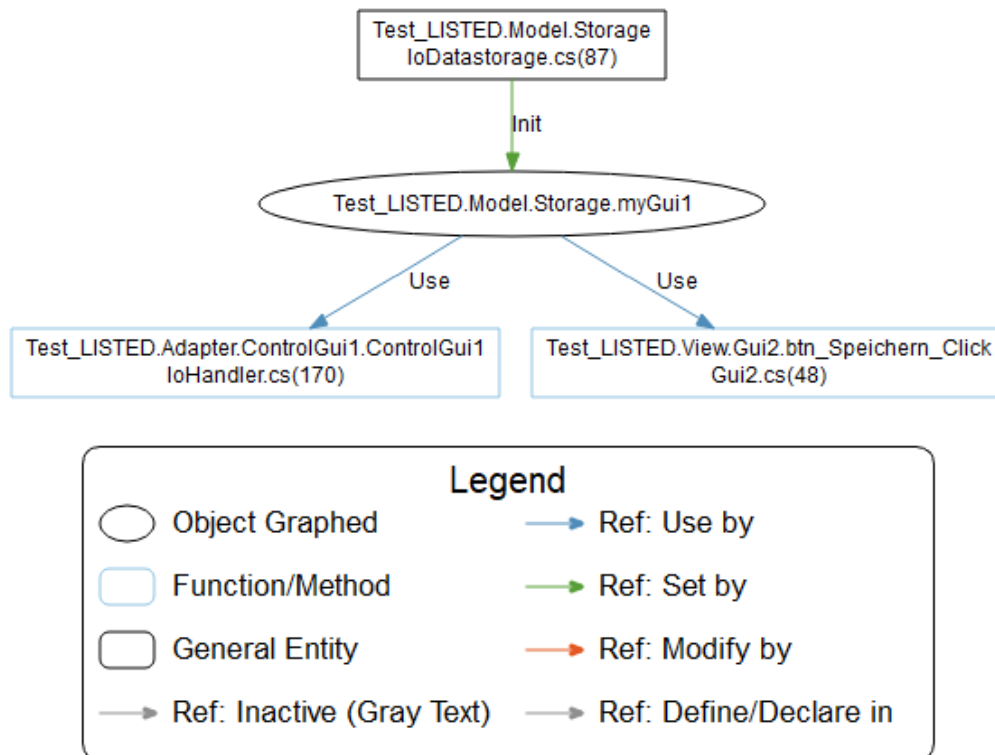


Abbildung 12: Objekt Referenzen der Instanz myGui1 von Gui1

Aus Abbildung 12 wird ersichtlich, dass ControlGui1 und Gui2 direkt auf die Instanz von Gui1 zugreifen. Damit wird das zuvor beschriebene Entwurfsmuster ermöglicht und implementiert.

Vor der Implementierung des Entwurfsmusters benötigt Gui1 die Elemente, die mit Gui2 erzeugt wurden. Mit dem Connector „Usage“ wird in Abbildung 13 auf der rechten Seite veranschaulicht, dass Gui1 nur mit Gui2 seine vollständige Implementierung erreichen kann. Das bedeutet, nur wenn die Eingaben von Gui2 vollständig sind und das Fenster geschlossen wird, kann Gui1 auf die Eingaben zugreifen.

Das UML-Diagramm nach der Implementierung des Observer Patterns ist in Abbildung 13 auf der linken Seite zu sehen. Die Klasse Form enthält die Generalisierung für das Objekt Gui2. In dieser Klasse sind auch Subscriber-Methoden zu finden, mit deren Hilfe bereits ein Teil des Observer Patterns implementiert ist. Gui1 greift nun auf die Methode „Click“ von Gui2 zu (Realisierung). Bei jeder Aktivierung der Methode, also bei jedem Klick auf den Button „Speichern“ werden die Textboxen von Gui1 über den geänderten Status informiert. Daraufhin wird der Inhalt der Textboxen verändert.



Abbildung 13 stellt die Abhängigkeit der Methoden und Elemente in Gui1 vereinfacht dar. Der Aufruf ähnelt im Detail eher einer Aggregation, da ein Element von Gui2 (Button) seine vielfachen Subscriber in Gui1 (Textboxen) informiert.

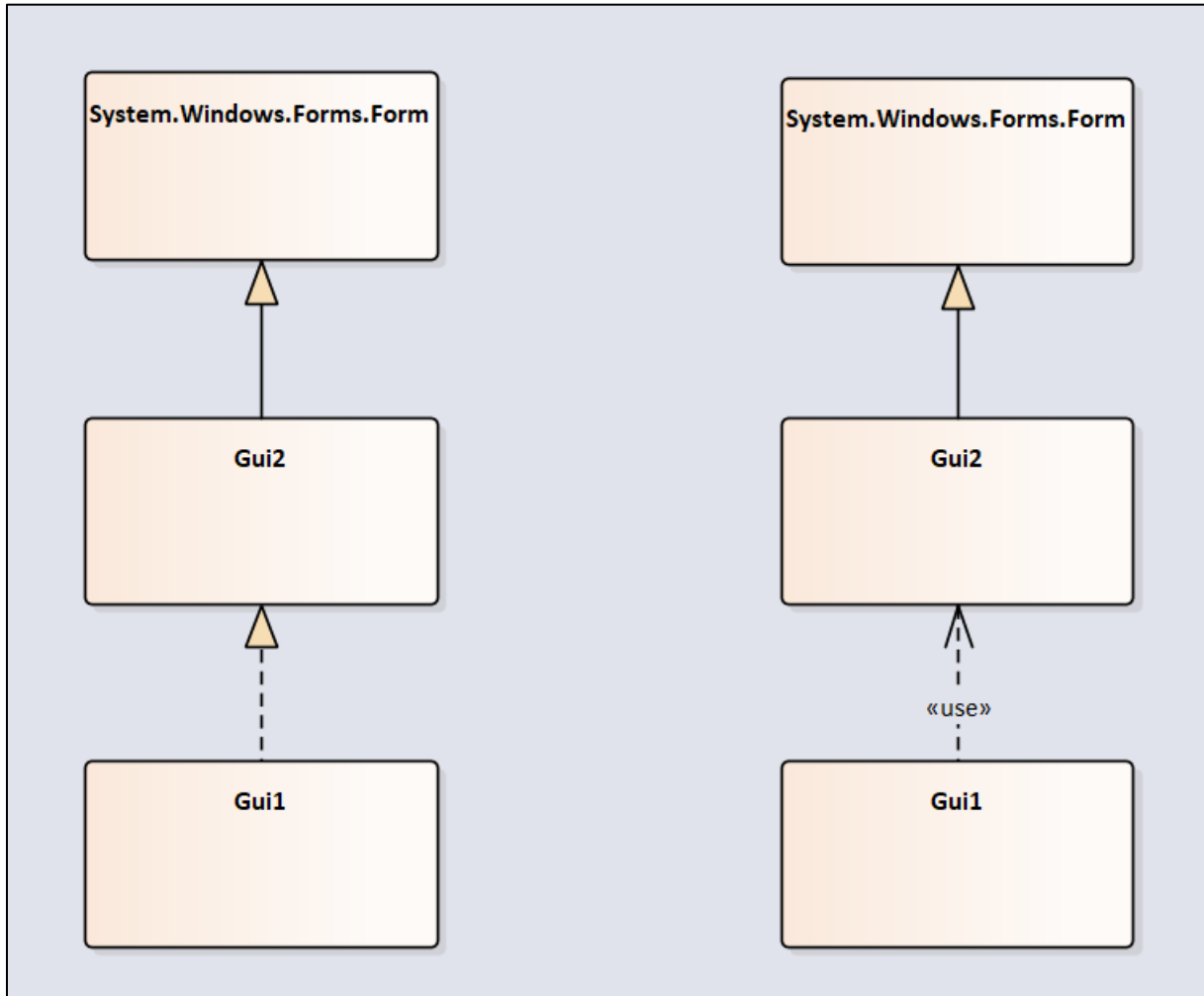


Abbildung 13: Mit und ohne Observer Pattern

## Quellenverzeichnis

1. Clean Architecture: A Craftsman's Guide to Software Structure and Design: Prentice Hall; 2017.
2. Fowler M, Beck K. Refactoring: Improving the design of existing code. Boston: Addison-Wesley; 2019.
3. Dowalil H. Grundlagen des modularen Softwareentwurfs: Der Bau langlebiger Mikro- und Makro-Architekturen wie Microservices und SOA 2.0. 2nd ed. München: Hanser; 2020.
4. Metzner A. Software-Engineering - kompakt. München: Hanser; 2020.
5. Observer. 15.05.2021. <https://refactoring.guru/design-patterns/observer>. Accessed 31 May 2021.
6. Hitzges A. Usability als wesentlicher Erfolgsfaktor für Unternehmenssoftware. Wirtsch Inform Manag. 2016;8:100–8.