**Programmentwurf**

**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*

**Inhaltsverzeichnis**

[Inhaltsverzeichnis i](#_Toc73288929)

[Abbildungsverzeichnis ii](#_Toc73288930)

[1. Bearbeitung der Aufgabenstellung 1](#_Toc73288931)

[1.1. Auswahl geeigneter Methoden 1](#_Toc73288932)

[1.2. Formulierung der verwendeten Algorithmen in einer Programmiersprache 1](#_Toc73288933)

[1.3. Testen und Überprüfen der Ergebnisse 2](#_Toc73288934)

[2. Programmdokumentation 3](#_Toc73288935)

[2.1. Unit Test 3](#_Toc73288936)

[2.2. Programming Principles 4](#_Toc73288937)

[2.3. Refactoring 9](#_Toc73288938)

[2.4. Clean Architecture 12](#_Toc73288939)

[2.5. Entwurfsmuster 15](#_Toc73288940)

**Abbildungsverzeichnis**

[Abbildung 1: „using“-Direktiven in Visual Studio 6](#_Toc73288495)

[Abbildung 2: UML-Diagramm der Abhängigkeiten der Benutzeroberflächen 7](#_Toc73288496)

[Abbildung 3: Die Architektur nach Vorschlag der Clean Architecture 11](#_Toc73288497)

[Abbildung 4: Beziehungen des Moduls zueinander 12](#_Toc73288498)

[Abbildung 5: Cluster Call Butterfly Ansicht des Moduls „Plugin“ 13](#_Toc73288499)

[Abbildung 6: Cluster Call Butterfly Ansicht des Moduls „Kern“ 13](#_Toc73288500)

[Abbildung 7: Cluster Call Butterfly Ansicht des Moduls „Adapter“ 13](#_Toc73288501)

[Abbildung 8: Abstrakte Ansicht der Aufrufe 14](#_Toc73288502)

[Abbildung 9: Detailliertere Ansicht der Aufrufe 14](#_Toc73288503)

[Abbildung 10: Objekt Referenzen der Instanz myGui1 von Gui1 16](#_Toc73288504)

# **Bearbeitung der Aufgabenstellung**

## **Auswahl geeigneter Methoden**

Der Programmentwurf soll eine Anwendung sein, die in Forme einer Todo-Liste dem Benutzer behilflich ist. Dabei wird eine größere Menge von potenziellen Benutzern angesprochen, die unterschiedliche Erfahrungsstände und Kenntnisse im Bezug zur Benutzung eines Computers haben. Daher sollen die Benutzeroberfläche und Anwendungssteuerung sehr einfach gehalten werden. Das Programm ist auch für außerhalb des deutschen Sprachraums geeignet, da die Benutzeroberfläche auf Englisch gehalten ist.

Da der Fokus in der begrenzten Arbeitszeit auf der Erstellung der Anwendungsführung und den unterliegenden Algorithmen liegt, wurde auf die Programmiersprache Microsoft C# gesetzt. 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 ermöglich die Erstellung einer .exe-Datei, die auf jedem Betriebssystem mit der entsprechend installierten .NET Core Version oder einer dazu kompatiblen Version funktionsfähig ist.

## **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. 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 zum Feedback an die Benutzer zur Verfügung. Über dieselbe Schaltfläche, die nun „Log Out!“ heißt, kann sich der Benutzer wieder abmelden. Über die Schaltfläche „X“ kann das Programm geschlossen werden. Die erstellten Unterpunkte werden beim nächsten Starten der Anwendung wieder geladen.

## **Testen und Überprüfen der Ergebnisse**

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

# **Dokumentation des Programms**

## **Unit Test**

Durch das Testen kleinster 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 semanstischen 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. Jede dieser Teilkomponenten ist für den nach ihrem benannten Teil verantwortlich. Nachfolgend sollen zwei Unit Tests am Beispiel des Programmentwurfs dargestellt werden. Dem anschließend werden die Referenzen zu den restlichen acht Unit Tests stehen.

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

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.

Die Benutzeroberfläche kann nur mit dem 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.

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. In den Github-Tags

## **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. [Clean Architecture, Robert C. Martin, S. 72ff.] Die SOLID-Prinzipien beschreiben, wie einzelne Bereiche der Außenform zueinanderstehen müssen. Das beschreibt nicht das Zueinander stehen der Bereiche innerhalb des Systems. Diese werden von den Komponentenprinzipien beschrieben, auf die hier nicht weiter eingegangen werden soll. [Clean Architecture, Robert C. Martin, S. 103ff.]

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 der Klasse Gui1 gesehen werden. Beide Methoden haben nur eine Aufgabe, einmal den Namen und einmal das Datum aus dem Datenstrom zu extrahieren. Zunächst waren beide Funktionalitäten sequentiell nacheinander in einer Methode Button1\_Click zu finden. Nach Anwendung des SRP-Prinzips wurden unter anderem diese Methoden extrahiert.

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. [Clean Architecture, Robert C. Martin, S. 75ff.]

OCP

Das Open Closed Principle (OCP) ist ein erstrebenswerter Ansatz der Softwareprogrammierung. Bei jeder Veränderung am bisherigen Code durch beispielsweise dem 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 einer Hinzufügung oder Entfernung 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. [Clean Architecture, Robert C. Martin, S. 82ff.]

LSP

Das Liskov Substiution 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 und die Erstellung von Objekten nach selbst definierten Klassen 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. [Clean Architecture, Robert C. Martin, S. 88f.]

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. [Clean Architecture, Robert C. Martin, S. 94ff.]

Durch die Verwendung der Entwicklungsumgebung Visual Studio werden nicht verwendete Verweise auf weitere Dateien entsprechend gekennzeichnet. Abbildung 1 stellt diese Markierung dar. Nicht verwendete Verweise der „using“-Direktiven werden weniger kontrastreich angezeigt.

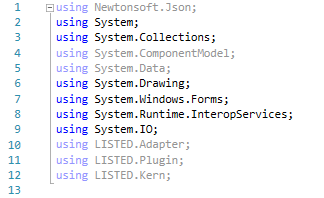


Abbildung 1: „using“-Direktiven in Visual Studio

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.

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.

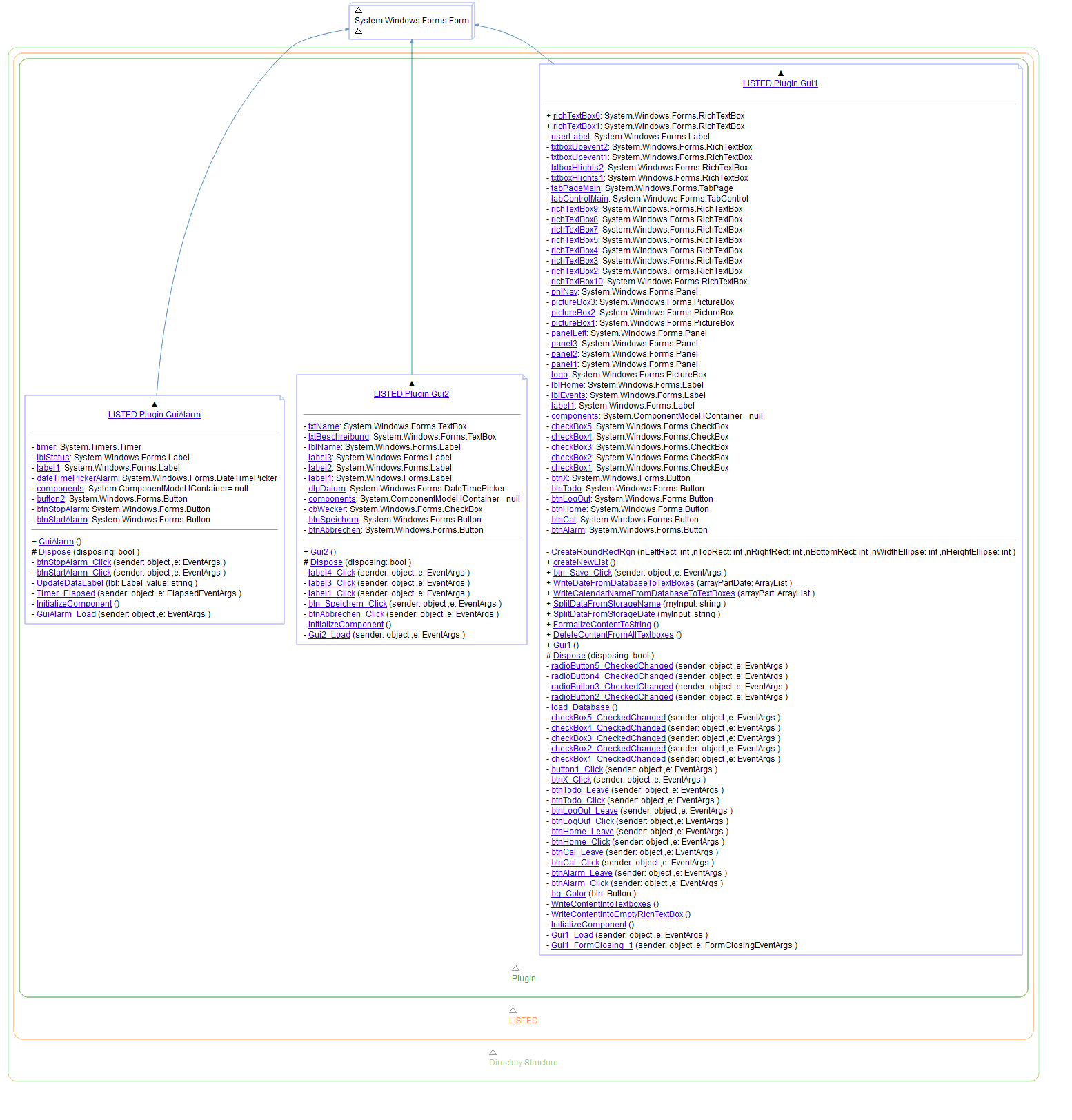


Abbildung 2: UML-Diagramm der Abhängigkeiten der Benutzeroberflächen

Abbildung 2 stellt die Abhängigkeiten der erstellten Benutzeroberflächen von der abstrakten Klasse „Forms“ des verwendeten Frameworks dar. Dabei ist „Forms“ die Generalisierung, ihre konkrete Implementierung erfolgt nach Bedarf in jeder Benutzeroberfläche.

Dasselbe Prinzip gilt für die Klassen, die von „Object“ erben. Auch alle weiteren „using“-Direktiven, wie sie beispielsweise in Abbildung 1 dargestellt wurden, referenzieren nur abstrakte Implementierungen, die im jeweils vorliegenden Dokument erst konkret für den Anwendungsfall 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. [Clean Architecture, Robert C. Martin, S. 98ff.]

## **Refactoring**

Refactoring verändert das Programm in kleinen Schritten, dass wenn ein Fehler passiert, ist es einfach, den Bug zufinden. [Refactoring – Improving the Design of Existing Code, Martin C. Fowler, S. 13] Das Refactoring hilft interne Änderungen der internen Struktur des Systems zu verstehen und günstiger zu verändern, ohne sein beobachtbares Verhalten zu ändern. [S. 56] Das Verhalten soll in mehreren kleineren Teilschritten zur Erhaltung des Verhaltens möglich sein, wobei zu jedem Zeitpunkt ein Beenden des Refactoring möglich sein soll. [S. 56f.] Refactoring wird als das Säubern und Reorganisieren der Codesbasis verstanden und ähnelt einer Optimierung der Performance. [S. 57] Refactoring ist kein Zusatz von Funktionalität, sondern soll im ständigen Wechsel mit der Funktionalität angewandt werden. [S. 57f.] 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. [S. 74ff.] 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. [S. 77ff.] Das Refactoring kann durch die Entwicklungsumgebung unterstützt werden, wie unter der eingesetzten Visual Studio 2019 Community Edition. Diese unterstützt automatische und manuelle Refactorings. [S. 81ff.]

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. [Refactoring – Improving the Design of Existing Code, Martin Fowler, S. 74ff.]

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: eacb19

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.

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.

Schritt-Version: 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.

Schritt-Version: 544bc03

Auskommentierter nicht genutzter Code-Block wird gelöscht.

## **Clean Architecture**

Die entwickelte Anwendung kann anhand der nachfolgenden Abbildungen dargestellt werden.

Abbildung 3 zeigt die Darstellung der Anwendung nach Einteilen in drei entsprechende Schichten der Clean Architecture „Application-Code“, „Adapter“ und „Plugin“.

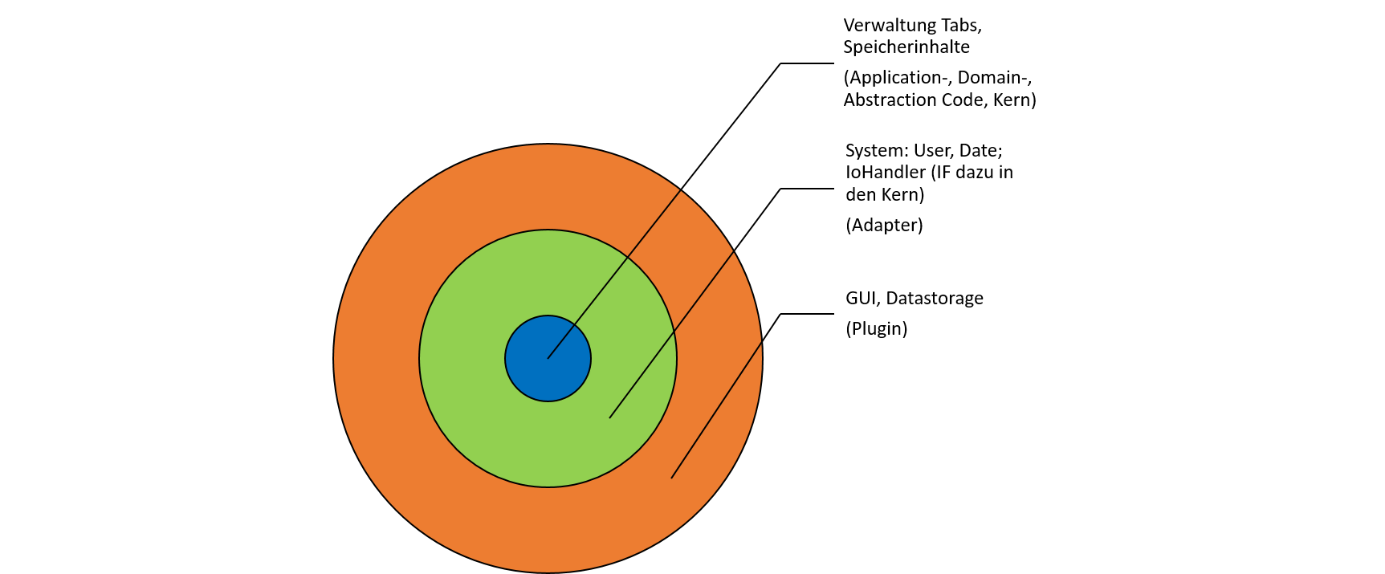


Abbildung 3: Die Architektur nach Vorschlag der Clean Architecture

Die Abhängigkeiten zwischen den Schichten sollen entsprechend der gewählten Architektur-Vorgabe von außen nach innen gewählt werden (Dependency Inversion). [Clean Architecture, Robert C. Martin, S. 60ff.]

Abbildung 4 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.

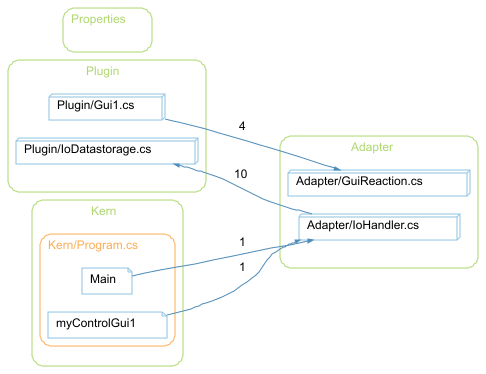


Abbildung 4: Beziehungen des Moduls zueinander

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 5 stellt die Richtung der Aufrufe zwischen den beiden Schichten „Adapter“ und „Plugin“ dar. Die Pfeile zeigen von „Adapter“ zu „Plugin“. Die Abhängigkeiten können entsprechend [Robert C. Martin: Clean Architecture, S. 60ff.] invertiert zum Kontrollfluss (Call) betrachtet werden.

Abbildung 6 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 7.

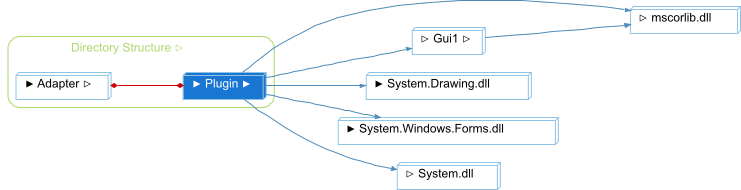


Abbildung 5: Cluster Call Butterfly Ansicht des Moduls „Plugin“

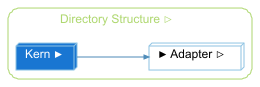


Abbildung 6: Cluster Call Butterfly Ansicht des Moduls „Kern“

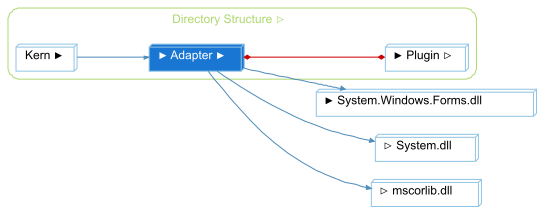


Abbildung 7: Cluster Call Butterfly Ansicht des Moduls „Adapter“

Die vorgenommenen Schritte ergeben die Abhängigkeiten, die in Abbildung 8 abstrakt dargestellt sind.

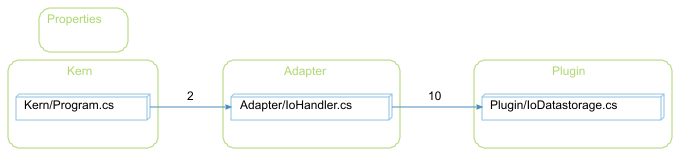


Abbildung 8: Abstrakte Ansicht der Aufrufe

Aus Abbildung Abbildung 9 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. [Modulare Softwarearchitektur, S. 95ff.]

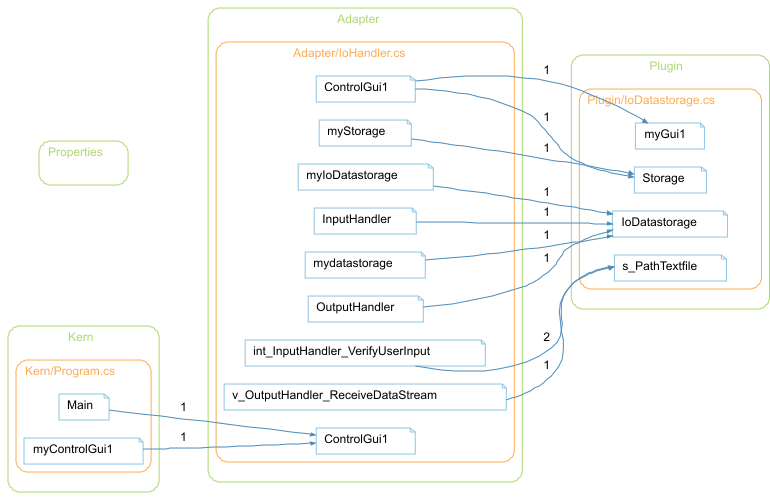


Abbildung 9: Detailliertere Ansicht der Aufrufe

## **Entwurfsmuster**

Entwurfsmuster beschreiben eine verwendete Mikroarchtektur im Software-System. Auf dieser basierend kann die im vorherigen Kapitel beschriebene Makroarchitektur, die Architektur des vollständigen Systems abstrakt dargestellt werden.

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. [Metzner, A. (2020). Software-Engineering - kompakt. S. 119-130]

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. [Usability als Erfolgsfaktor, 3/2016, Prof. Dr. Arno Hitzges]

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.

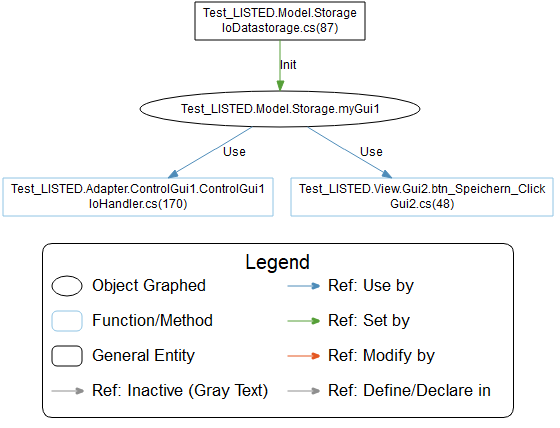
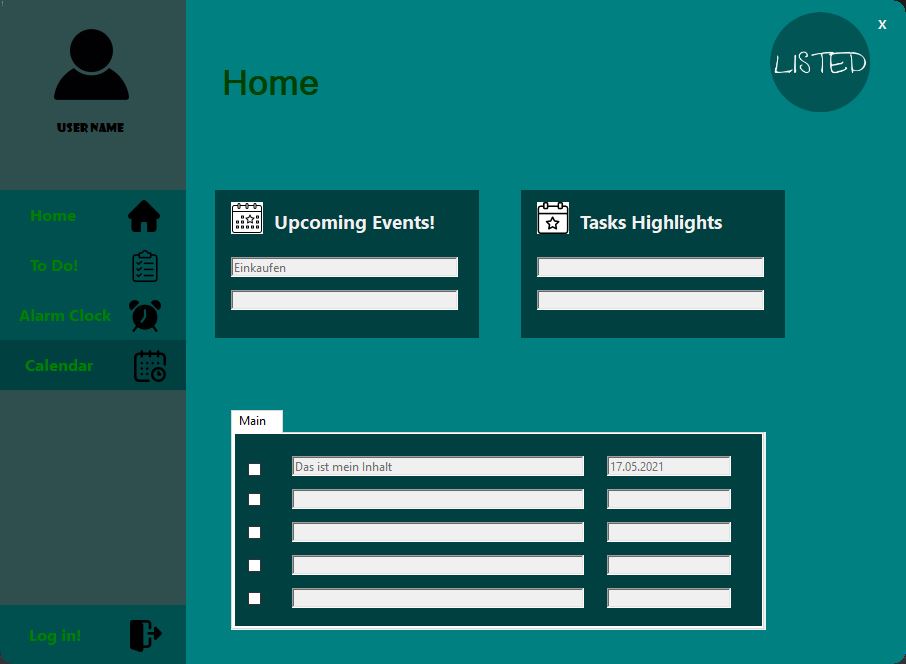


Abbildung 10: Objekt Referenzen der Instanz myGui1 von Gui1

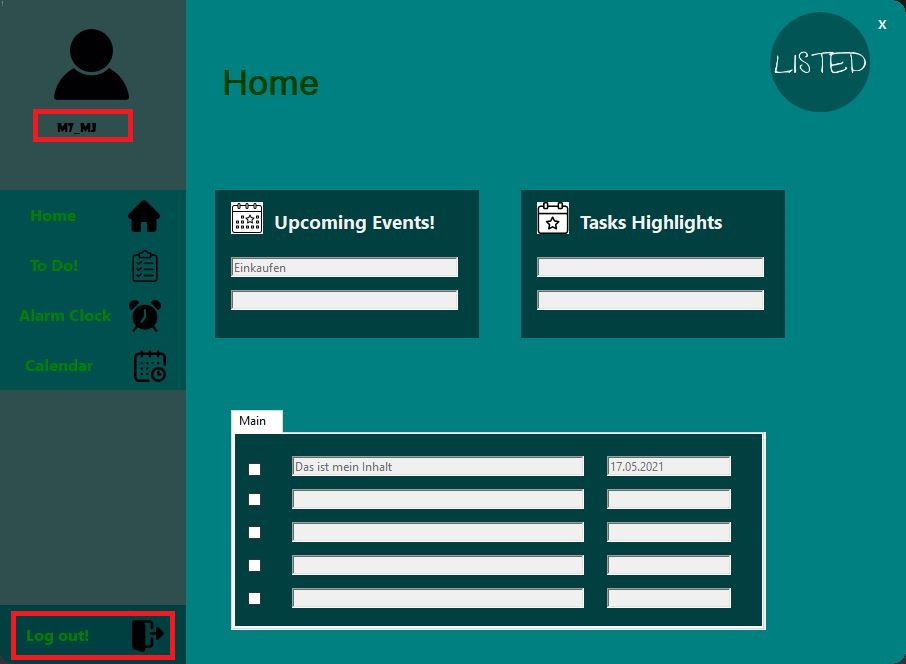
Aus Abbildung 10 wird ersichtlich, dass ControlGui1 und Gui2 direkt auf die Instanz von Gui1 zugreifen. Damit wird das zuvor beschriebene Entwurfsmuster ermöglicht und implementiert.

**LISTED Demo**

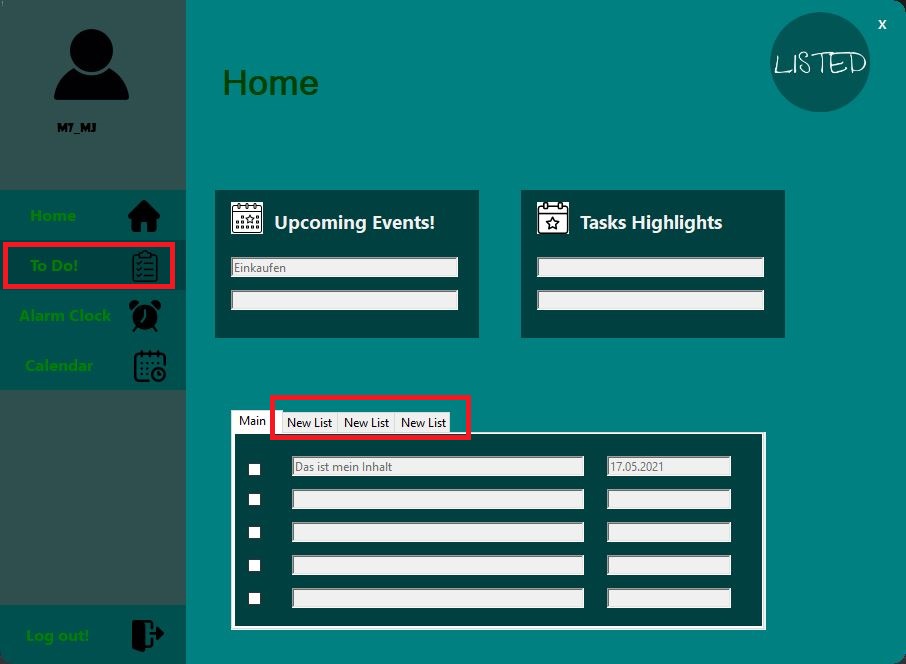
**Home:**

****

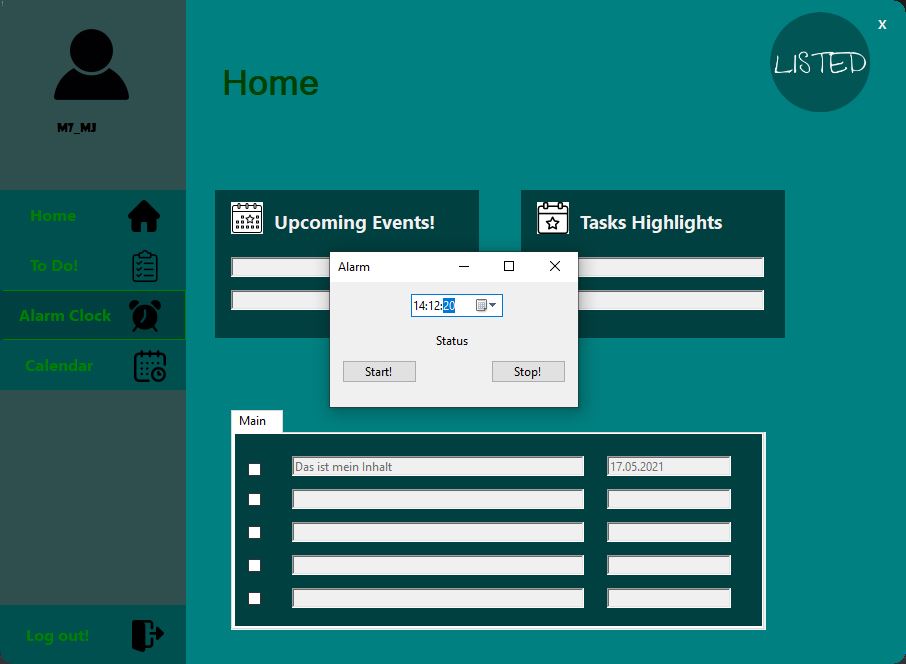
**Login:** momentan nur mit Windows Benutzer ID (Die Erweiterung um mehrere Benutzer mit Benutzerverwaltung ist möglich und teilweise implementiert)



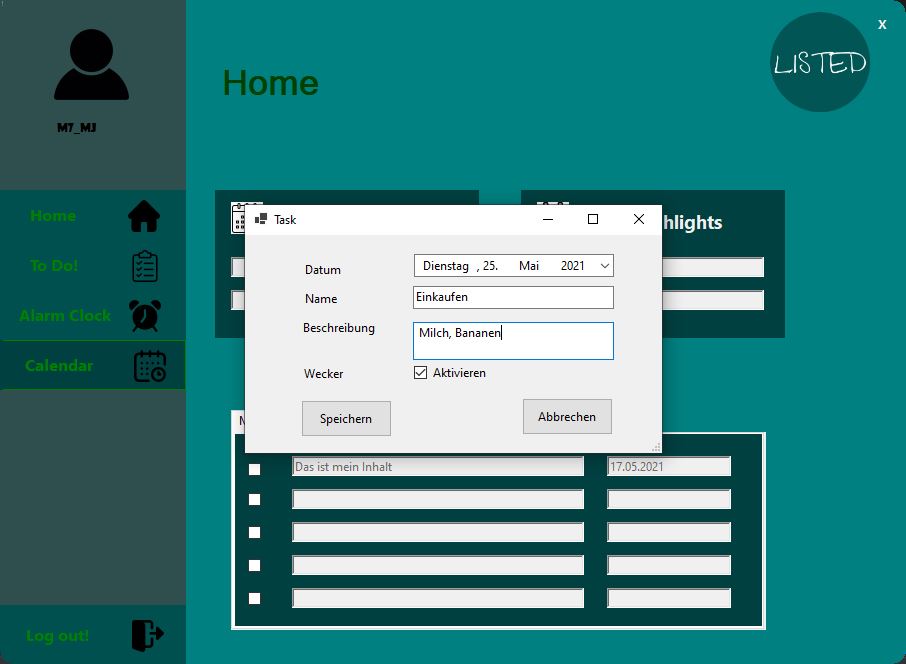
**Erstellung von neuen Listen:**

****

**Alarm Clock:**

****

**Kalender:**

****