Programmentwurf

einer Todo-Liste in der

Programmiersprache Microsoft C#

Fach: Advanced Software Engineering

vorgelegt von

Michaela Fleig und Mohammad Mehjazi

Matrikel-Nr. HIER und HIER

Inhaltsverzeichnis

[Bearbeitung der Aufgabenstellung (Praxisprojekt) 3](#_Toc69157778)

[Auswahl geeigneter Methoden 3](#_Toc69157779)

[Formulierung der verwendeten Algorithmen in einer Programmiersprache 3](#_Toc69157780)

[Testen und Überprüfen der Ergebnisse 3](#_Toc69157781)

[Programmdokumentation 3](#_Toc69157782)

[Unit Test (mind. 10) 3](#_Toc69157783)

[Programming Principles 3](#_Toc69157784)

[Refactoring (mind. 2) 3](#_Toc69157785)

[Clean Architecture (mind. 2 Schichten) -> 3](#_Toc69157786)

[Entwurfsmuster (mind. 1) -> Presentation Model (setzt auf MVC auf) nach Gernot Starke 3](#_Toc69157787)

# 

# Bearbeitung der Aufgabenstellung (Programmentwurf)

## Auswahl geeigneter Methoden (Wahl Programmiersprache, OS)

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 aktuellsten 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 die selbe 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.

# Programmdokumentation

## Unit Test (mind. 10) -> ein paar (Hashes, beschreiben der Funktionalitäten, AAA, Code Coverage, Mock)

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 ihr 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.

HIER VERGLEICHE EINFÜGEN

## Programming Principles -> SOLID-Principles (SRP, OCP nicht möglich, LSP möglich?, ISP fehlt noch, DIP fehlt noch)

Programming Principles stellen die als Design Prinzipien eine gute Basis für sauberen Code in Software Systemen unterschiedlicher Größe. 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 Software Systemen 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 zueinander stehen 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.]

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.

HIER VERGLEICH EINFÜGEN – SRP [Beispiele: Versehentliche Duplikation, Zusammenfügen von unterschiedlichen Methoden die für dieselbe Personengruppe arbeiten, Facade Pattern kann Instanzen und Delegierungen von vorhandenen Klassen mit Funktionen übernehmen]

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.]

Das Open Closed Principle (OCP) ist ein erstrebenswerter Ansatz der Software Programmierung, kann in diesem nicht realisiert werden. Bei jeder Veränderung am bisherigen Code durch beispielsweise dem Hinzufügen einer Eingabefläche muss alter Code mindestens dahingehend verändert werden, dass die Eingabefläche nach Inhalt der Speicher-Datei initial leer oder befüllt sein muss.

HIER VERGLEICH EINFÜGEN [Direktionskontrolle, Informationen verstecken]

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 Komponenten. 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.]

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

HIER VERGLEICH EINFÜGEN

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.]

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.]

HIER VERGLEICH EINFÜGEN

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.

HIER VERGLEICH EINFÜGEN [Bezug auf abstrakte Klassen, Vererbung aus abstrakten Klassen, keine konkreten Funktionen überschreiben, kein konkreter Name nennen]

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 (mind. 2) -> Bad Smells (Duplizierter Code unten, Lange Funktionen fehlt noch, mysteriöser Name fehlt noch teilweise, Code-“Klumpen” über Kommentare fehlt noch -> Understand, Shotgun Operation Gefahr mit aktueller Realisierung -> resultiert in Feature Neid)

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 reoganisieren 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 Software Version 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.]

HIER VERGLEICHE EINFÜGEN

## Clean Architecture (mind. 2 Schichten) -> Verwaltung Tabs, IoHandler & User & Date, DataStorage & GUIs; (auch Presentation Model, nach Gernot Starke)

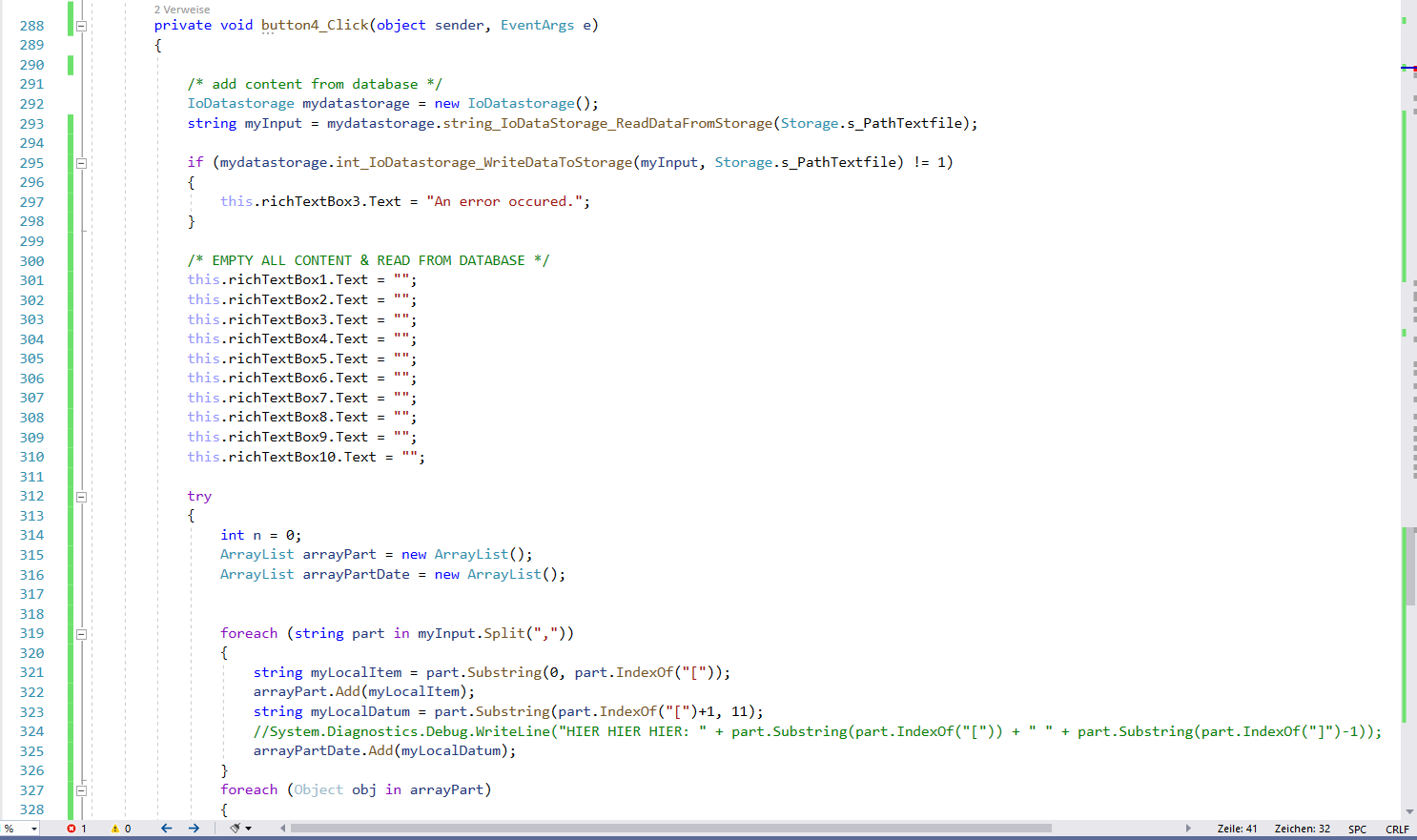
## Entwurfsmuster (mind. 1) -> Observer

Refactoring

Von Stand: xxx



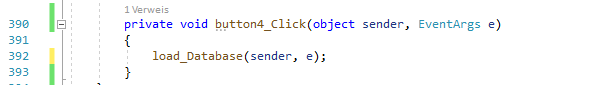
Ist nicht lesbar, nicht voll funktionsfähig, nach Martin C. Fowler ideal zum Refactoring. (Lange Funktion)



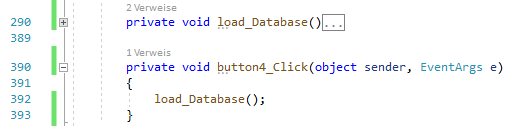
Funktion soll an anderer Stelle ebenfalls verwendet werden. Selbe Funktion, nicht verändert.



Funktion extrahiert. (Refactoring: Extract Method)



Aufruf der Funktion, als ein Simulieren des Buttons.



Funktion als eigenständige Funktion aufgerufen.

\*\*\*\*\*\*\*\*\*\*\*\*\*

Refactoring: Extract Method (Lange Funktion über viele ifs gefunden, doppelte Verwendung der selben Funktionalität) durch Decomposition

Ausgangsversion: f5d1184

* Viele if-Bedingungen, um jede Textbox auf ihren aktuellen Inhalt zu überprüfen und zu beschreiben. Danach Inhalte (Termin-Name und –Datum) formatieren und in Datenbank schreiben. Überprüfung, ob Termin-Datum mit heutigem Datum übereinstimmt. Falls ja, in die nächste freie Textbox von Gruppe „Upcoming Events!“ schreiben.
* Sehr viele ähnliche Bedingungen, wenige inhaltliche Unterschiede. Extract Method.

Schritt-Version: 9c11979

* Extrahierte if-Bedingungen, über einen Switch-Case mit dem Count gelöst.
* Count noch immer in Bedingung. Weitere if-Bedingungen noch offen.

Schritt-Version: e076a78

* Code für Count aus if gelöscht, Debug-Anweisung gelöscht.
* Viel doppelter Code. Wird aber so an keiner weiteren Stelle aktuell gelöscht (nicht umsonst Code für mögliche zukünftige Fälle schreiben), daher aktuell beste Lösung.

Schritt-Version: 59c9d9f

* Löschen von offensichtlich nicht benötigtem Code.
* Noch immer Bedingungen mit ähnlichen Abfragen. Extract Method mit Übergabeparameter.

Schritt-Version: a6c65ed

* Extrahierte if-Bedingungen.
* Methode wird direkt aufgerufen (Remove Variable).

Schritt-Version: eacb19

* Remove Variable, nur noch in extrahierter Methode bekannt.
* Funktion zum Kalender ist noch vorhanden. Remove Variable.

Schritt-Version: 883b515

* Remove Variable, nur noch direkt beim Aufruf verwendet.

Schritt-Version: a724813

* Remove Variable, nur noch direkt beim Aufruf verwendet.

Schritt-Version: c95225c

* Verändere Reihenfolge von if-Statement.

Schritt-Version: f41042f

* Extract Method.

Schritt-Version: be62f25

* Kommentare schreiben.

Finale Version: a881f0c

* Veränderter Name (Rename Method als Teil von Simplifying Method Calls)

Refactoring 2:

Start-Version: 1b20c72

Schritt-Version: ce14b3b

Schritt-Version: 2e6a092 (extrahiert Methode und Variable)

Schritt-Version: 54c15af (extrahiert Methoden und Variablen)

Schritt-Version: 42f9436 (if Bedingung in catch-Block umgewandelt)

Schritt-Version: 544bc03 (kommentierten Code-Block gelöscht)

Refactoring 3:

Start-Version: 25241b4

Schritt-Version: 11b1f1e

Schritt-Version: b91977e

Clean Architecture:

Schritt-Version: 96c49e6 (umbenennen der Ordner, erste Abhängigkeiten geändert, von main zu gui)