TECHNISCHE HOCHSCHULE MITTELHESSEN
CAMPUS FRIEDBERG
MND
Mathematik, Naturwissenschaften und Datenverarbeitung

Seminararbeit

im Studiengang Wirtschaftsinformatik

am Fachbereich Mathematik, Naturwissenschaften und Datenverarbeitung
der Technischen Hochschule Mittelhessen

Clean Code

vorgelegt von:
Cong Chanh Vinzenz Nguyen
Matrikelnummer: 5291018
E-Mail: cong.chanh.vinzenz.nguyen@mnd.thm.de

Referent: Armin Wiechmann, M.Sc.

Sommersemester 2022

Abstract

Programmcode wird mit der Zeit immer größer und komplexer. Schlechte Architektur und starke Abhängigkeiten erschweren die Wartung und Erweiterung bestehender Programme. Lösungsansätze für das Problem werden unter anderem in Form von Clean Code Prinzipien und Techniken entwickelt. Mithilfe von ausgewählter Literatur und eigenen Beispielen werden in der vorliegenden Arbeit einige Clean Code Techniken untersucht, die Vorteile und Nachteile herausgearbeitet, sowie abgewogen, inwiefern Clean Code die Codequalität steigert. Obwohl viele der Techniken helfen können besseren Code zu schreiben, sollte dies jedoch nur mit Bedacht eingesetzt werden und weitere Ansätze wie funktionale Programmierung beachtet werden.

Inhaltsverzeichnis

iv. Abbildungsverzeichnis iv

v. Code Verzeichnis v

1 Einleitung 1

1.1 Problemstellung 1

1.2 Zielsetzung und Erkenntnisinteresse 1

1.3 Aufbau der Arbeit 2

2 Moderne Clean Code Techniken 2

2.1 Grundlagen und der Begriff Clean Code 2

2.2 SOLID Principles 3

2.2.1 Allgemeines zu Clean Code 3

2.2.2 Single Responsibility Principle 3

2.2.3 Open Closed Principle 4

2.2.4 Liskov Substitution Principle 4

2.2.5 Interface Segregation Principle 5

2.2.6 Dependency Inversion Principle 5

2.2.7 DI Container 9

2.3 Softwarearchitektur 10

2.4 Nachteile von Clean Code 12

2.4.1 Erweiterte Programmierkenntnisse 12

2.4.2 Lesbarkeit 13

2.4.3 Performance 13

2.4.4 Programmierparadigmen 14

3 Fazit 17

vi. Literaturverzeichnis vi

iv. Abbildungsverzeichnis

Abbildung 1: Vererbungshierachie von Mock Objekt zu einer konkreten Implementierung 4

Abbildung 2: Polymorpher Aufruf über eine Schnittstelle 5

Abbildung 3: 3-Schichten Architektur 11

Abbildung 4: Neu entstandene Onion Architecture [12] 12

Abbildung 5: Benchmark Ergebnisse mit logarithmischer Skalierung [12] 14

v. Code Verzeichnis

Code 1: Copy Modul als Konsolenanwendung 6

Code 2: Copy Modul mit Auswahl für Konsole und Textdatei 6

Code 3: Interfaces IReader und IWriter 7

Code 4: Konkrete Klassen für IWriter und IReader 8

Code 5: Composition Root mittels Poor Man's DI 9

Code 6: AutoFac Container 9

Code 7: Eine Copy Instanz wird erstellt und ausgeführt 10

Code 8: IFilter Interface mit einer Execute Methode 14

Code 9: Konkrete Filterklasse 15

Code 10: Composition Root für ein Filter Beispiel 16

Code 11: Generische Filter Methode mit einer Funktion als Übergabeparameter 16

Code 12: Main Methode für einen Filter mit funktionaler Programmierung 16

1 Einleitung

1.1 Problemstellung

Mit der Einführung von Computern in die Arbeitswelt wuchs der Bedarf entsprechender Software und deren Anforderungen. Daraus resultierten weitere Anforderungen an die Programme, weshalb deren entsprechender Quellcode größer wurde.

Somit wuchs auch die Komplexität des Quellcodes. Da viele dieser Codes zum Teil unsauber geschrieben worden sind, wurde es schwieriger, diesen Code anzupassen und zu pflegen. Dies ist ein normaler Vorgang und wird unter Entwicklern als „gewachsene Struktur“ oder „Legacy Code” bezeichnet. Wenn neue Entwickler in ein bestehendes Projekt eingearbeitet werden müssen, kann dies dazu führen, dass sie den Code nicht auf Anhieb verstehen.

Ein weiteres mögliches Problem ist, dass die Architektur so starke Abhängigkeiten aufweisen kann, sodass eine minimale Änderung eines Moduls dafür sorgen kann, dass das Programm aufgrund von Seiteneffekten ein anderes und unerwünschtes Verhalten darlegt.

Dies sorgt dafür, dass ganze Projekte von Grund auf neu geschrieben werden müssen oder sich die Kosten für die Wartung vervielfachen. Im schlimmsten Falle könnte dies zu einer Firmeninsolvenz führen. Auch im Zuge von agilen Vorgehensmodellen, welche in der aktuellen Softwareentwicklung immer beliebter werden, ist flexibler Code sehr wichtig geworden, da sich die Anforderungen permanent ändern können.

Um dieses Problem zu lösen, wurde eine Reihe von Techniken und Konzepten entwickelt, welche unter dem Begriff „Clean Code“ gefasst werden. Robert C. Martin, der auch Uncle Bob genannt wird, hat in seinem Buch „Clean Code“ [1] diese Konzepte thematisiert.

Es folgten weitere Publikationen von Robert C. Martin, Martin Fowler, Gamma et al. und anderen Autoren. Diese Werke sind zum Teil populärwissenschaftlich und innerhalb der Softwarebranche hoch angesehen.

1.2 Zielsetzung und Erkenntnisinteresse

Das Ziel dieser Hausarbeit ist es dem Leser einen Überblick über Clean Code Techniken zu geben und diese vorzustellen. Das Ziel von Clean Code Techniken ist es im Allgemeinen besser lesbaren und wartbaren Code zu entwerfen, um es anderen Entwicklern zu ermöglichen, schnell mit dem bereits erstellen Code zurechtkommen zu können und diesen zu erweitern.

Darüber hinaus werden Beispiele zu den einzelnen vorgestellten Techniken gezeigt.

1.3 Aufbau der Arbeit

Es handelt sich bei dieser Arbeit um eine Literaturarbeit im Rahmen eines Seminars im vierten Fachsemester für den Studiengang Wirtschaftsinformatik. Hierzu wurde eine Auswahl an Literatur getroffen. Die Erkenntnisse werden anhand von Beispielen in der Programmiersprache C# demonstriert.

Da diese Arbeit sich an ein Publikum des vierten Fachsemester Wirtschaftsinformatik richtet werden grundlegende Programmierkenntnisse in einer objektorientierten Programmiersprache wie Java oder C# vorausgesetzt.

Um das Thema verständlich zu machen, wird zunächst auf die wichtigsten Grundlagen eingegangen, Begriffe definiert und die Entstehung der Problematik behandelt, weswegen Clean Code erst notwendig wird.

Anschließend werden die „Solid Principes of Object Oriented Design“, kurz SOLID, vorgestellt und anhand von Beispielen in C# demonstriert. Insbesondere das Dependency Inversion Principle und die daraus resultierenden Veränderungen und Notwendigkeiten wie Dependency Injection werden in dieser Arbeit genauer behandelt. Als Dependency Injection Framework wird AutoFac verwendet.

Das Thema Code Conventions, welches ebenfalls zu Clean Code gehört, kann im Rahmen einer solchen Seminararbeit nicht behandelt werden. Dieses wird bereits in den meisten Dokumentationen wie zum Beispiel dem .Net Framework genauer erläutert.

Zum Schluss werden die wichtigsten Erkenntnisse zusammengefasst und mit den Auswirkungen von Clean Code auf die Qualität des Codes evaluiert. Ferner wird ein Ausblick gewagt auf die zukünftige Entwicklung von Clean Code und Trends in Hinblick auf Programmierparadigmen und Prinzipien.

2 Moderne Clean Code Techniken

2.1 Grundlagen und der Begriff Clean Code

Bevor darauf eingegangen wird, welche Prinzipien es gibt und welche Probleme dadurch gelöst werden muss, zunächst definiert werden, was Code zu sauberem Code, also Clean Code macht. Unsauberer Code wird oftmals als „Code Smell“ oder auch nur „Smell“ genannt [2, p. 63].

Robert C. Martin sieht das Ziel der Softwarearchitektur darin, die Menge der benötigten menschlichen Ressourcen zu minimieren [3, p. 26].

Daraus folgt, dass Code für andere Entwickler verständlich sein sollte, um eine lange Einarbeitung zu verhindern. Außerdem sollte guter Code flexibel sein. Flexibilität meint, dass dieser leicht änderbar und erweiterbar ist. Dadurch soll schnell auf neue Anforderungen oder Bugs reagiert werden können. Dies impliziert auch, dass guter Code wiederverwendbar sein sollte. Ein weiteres Merkmal nach Robert C. Martin für guten Code ist die Testbarkeit in Form von isolierten Tests. [1, p. 40]. Das können zum Beispiel Unit Tests sein. Ferner sollen keine Seiteneffekte auftreten können [1, p. 75].

Zwei wichtige Metriken, um dies zu messen und zu bewerten sind Kopplung und Kohäsion. [1, p. 23] Viele der Prinzipien, die unter den Begriff Clean Code fallen, zielen auf eine lose Kopplung sowie eine starke Kohäsion ab. Kopplung bezeichnet die Abhängigkeit zwischen Softwaremodulen. Sollten keine direkten Abhängigkeiten vorhanden sein, so nennt man dies eine lose Kopplung. Kohäsion beschreibt, wie stark ein Modul seine Interna nutzt. Eine schwache Kohäsion entsteht dann, wenn viel modulinterner Code nicht benötigt wird oder nur selten benutzt wird.

2.2 SOLID Principles

2.2.1 Allgemeines zu Clean Code

Um Code sauber schreiben und entkoppeln zu können, bedarf es verschiedener Techniken. Einige der wichtigsten Techniken fasste Robert C. Martin als First Five Priciples [4] zusammen. Später wurde diese, von Michael Feathers das Akronym SOLID zugeschrieben [3, p. 63]. Neben diesen gibt es noch weitere Prinzipien, welche im Rahmen dieser Seminararbeit nicht behandelt werden, können. Zu den wichtigsten Prinzipien neben SOLID zählen Don't Repeat Yourself (DRY), You Ain't Gonna Need It (YAGNI) und Keep It Simple Stupid (KISS).

SOLID steht dabei für folgende Prinzipien [3, p. 63], die folgend noch genauer erläutert werden:

- Single Responsibilty Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Die wichtigste Technik für sauberen Code davon ist das Dependency Inversion Principle, da dies die Grundlage für Entkopplung darstellt. Robert C. Martin schrieb in einem Artikel dazu Folgendes:

"If the OCP states the goal of OO architecture, the DIP states the primary mechanism." [3, p. 65]

Auf diesen Prinzipen basieren viele weitere Überlegungen und Techniken. Beispiele wären Techniken wie Dependency Injection, einige Design Patterns von Gamma et al. [5] oder auch ganze Architekturmodelle wie Ports and Adapter von Martin Fowler. Diese werden im Kapitel Dependency Inversion Principle genauer erläutert.

2.2.2 Single Responsibility Principle

Das Single Responsibility Principle wurde von Robert C. Martin eingeführt.

"A module should be responsible to one and only one, actor." [3, p. 65]

Jedes Objekt sollte genau eine eindeutige Aufgabe haben und diese Aufgabe sollte von der entsprechenden Klasse komplett gekapselt sein.

In der Praxis sorgt das Single Responsibility Principle dafür, dass Klassen eine hohe Kohäsion erreichen. [1, p. 171] Das heißt, dass die Interna stärker genutzt werden und dadurch weniger Code geschrieben wird, welcher nur selten oder gar nicht ausgeführt wird. Die Klassen werden durch Einhaltung des Single Responsibility Principle kleiner und es wird vermieden, dass sich verschiedene Anliegen vermischen. [1, p. 172] Im Idealfall löst eine Klasse genau ein Problem. Als Folge des Single Responsibility Principles steigt die Gesamtzahl der Klassen, die leichter wiederzuverwenden sind und ein flexibleres Design ermöglichen.

2.2.3 Open Closed Principle

Das Open Closed Principle wurde von Betrand Meyer wie folgt beschrieben:

"A software artifact should be open for extension but closed for modification." [3, p. 70]

Code, der nach dem Open Closed Principle geschrieben ist, muss sich so verhalten, dass zu einem späteren Zeitpunkt neue Funktionalitäten hinzugefügt werden können, ohne bestehenden Code zu verändern. Dies betrifft jedoch nicht den Composition Root im Sinne von Dependency Injection, welches im Kapitel Dependency Inversion Principle genauer beschrieben wird. Umgesetzt wird das Open Closed Principle oftmals durch polymorphe Aufrufe mit Hilfe des Dependency Inversion Principle.

2.2.4 Liskov Substitution Principle

Das Liskov Substitution Principle wurde von Barbara Liskov entwickelt und bezieht sich auf die Vererbung. [3, p. 75]

Das Liskov Substitution Principle sagt aus, dass Objekte von Typ T durch Objekte vom Typ S ersetzt werden können, wobei S eine Subklasse von T ist. Dann müssen alle semantischen Einschränkungen von T auch für S gelten. [3, p. 75]

Verwendung findet dieses Prinzip unter anderem für die Erstellung von Mock Objekten beim Unit Testing. Statt konkreter Klassen werden Abstraktionen benötigt, von denen das Mock Objekt und die konkrete Implementierung ableiten. Beide leiten von der Abstraktion ab. Aufgrund dessen können diese beliebig untereinander ausgetauscht werden.

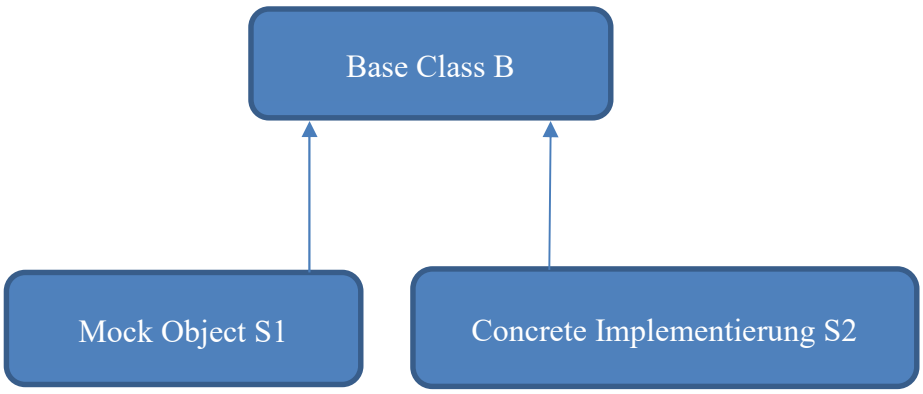

Abbildung 1: Vererbungshierachie von Mock Objekt zu einer konkreten Implementierung

Da ein Mock Objekt von der Klasse S1 ist und die Klasse S1 eine Subklasse der Basisklasse B ist, gelten alle semantischen Einschränkungen von B auch für S1. Die gilt ebenfalls für die konkrete Implementierung S2, die ebenfalls eine Subklasse der Basisklasse B ist. Somit können Objekte der Klasse S1 als auch S2 an jeder Stelle für Objekte der Klasse B verwendet werden. Dadurch wird ermöglicht, Mock Objekte zu einer konkreten Klasse erstellen zu können. Dies ist eine Form von Polymorphie.

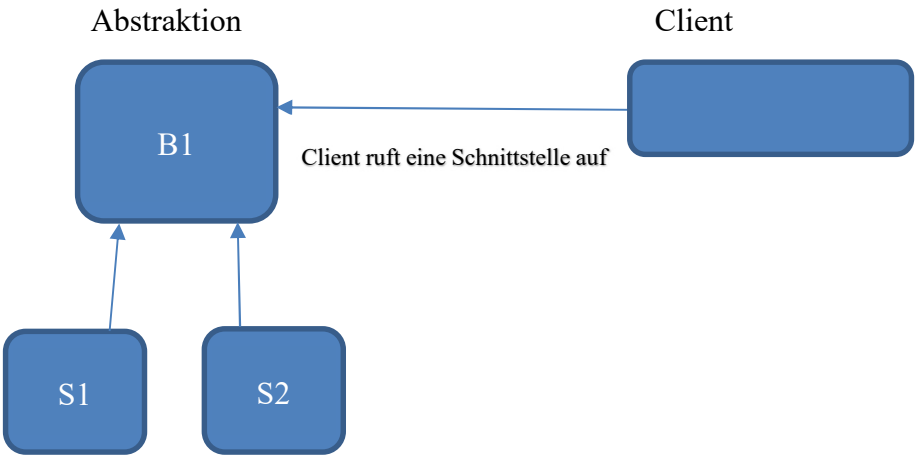

Abbildung 2: Polymorpher Aufruf über eine Schnittstelle

Eine Basisklasse kann verwendet werdet, um den Subklassen eine Standardimplementierung anzubieten und somit Code wiederzuverwenden. Wie am Beispiel gezeigt, kann Vererbung auch genutzt werden, um polymorphe Aufrufe zu ermöglichen.

2.2.5 Interface Segregation Principle

Das Interface Segregation Principle besagt, dass viele kleine Interfaces statt ein großes erstellt werden sollen. [6, p. 14] In der Praxis bedeutet das, dass Abstraktionen klein gehalten werden sollten und falls nötig in mehrere Abstraktionen aufgeteilt werden sollte. [3, p. 75] Dies führt dazu, dass viele kleine Abstraktionen entstehen anstatt wenige sehr große Abstraktionen. Dadurch wird die Geschwindigkeit bei der Softwareentwicklung erhöht, da Klassen schneller eine Abstraktion implementieren können und somit schneller Erweiterungen eingebaut werden können. Ferner erhöht es die Kohäsion.

2.2.6 Dependency Inversion Principle

Das Dependency Inversion Principle ist ein wichtiger Mechanismus für Entkopplung von Code. Eng miteinander verbunden sind Inversion of Control sowie Dependency Injection. Im weiteren Verlauf wird auch das Framework AutoFac vorgestellt, welches diese Prinzipien anwendet.

A. “High-level modules should not depend upon low-level modules. Both should depend upon abstractions.” [7, p. 6]

B. “Abstractions should not depend on details. Details should depend upon abstractions.” [7, p. 6]

Bei dem Dependecy Inversion Principle geht es um die Umkehrung und Auflösung von direkten Abhängigkeiten. Eine direkte Abhängigkeit ist dann gegeben, wenn ein Modul

direkt auf ein anderes Modul oder eine Funktionalität zugreift. High Level Modules sind solche Module, welche eine oder mehrere Low Level Modules zu einer neuen Funktionalität orchestrieren oder diese aufrufen. [7, p. 6] Das Dependency Inversion Principle soll direkten Zugriffe verhindern und somit auch direkte Abhängigkeiten. Dazu soll gegen Abstraktionen statt konkreter Klassen programmiert werden. Eine Abstraktion in C# oder Java kann eine abstrakte Basisklasse oder ein Interface sein. Details sind in diesem Kontext konkrete Implementierungen.

Um das Dependency Inversion Principle zu verdeutlichen, wird im Beispiel ein Copy Programm so verändert, dass es austauschbar wird.

```
public class Copy
{
   public void ReadAndWrite()
   {
      var input= Console.ReadLine();
      Console.WriteLine(input);
   }
}
```

Code 1: Copy Modul als Konsolenanwendung

Eine Betrachtung dieses Copy Moduls im Code 1 zeigt, dass dieses auf eine reine Konsolenanwendung beschränkt ist. Ändern sich die Anforderungen und das Modul soll auch Textdateien lesen und schreiben können, würde eine Lösung wie folgt aussehen.

```
public void ReadAndWrite(string inputTarget, string outputTarget)
{
   string input="";
   if (inputTarget == "console")
   {
      input= Console.ReadLine();
   }
   if (inputTarget == "txt")
   {
      input = File.ReadAllText("test.txt");
   }
   if (outputTarget == "console")
   {
      Console.WriteLine(input);
   }
   if (outputTarget == "txt")
   {
      File.WriteAllText("test.txt",input);
   }
}
```

Code 2: Copy Modul mit Auswahl für Konsole und Textdatei

Eine Erweiterung ist zwar möglich, jedoch mit vielen Nachteilen einhergehend. Der Code ist unnötig aufgeblasen, da mit jeder neuen Option eine neue If-Abfrage erstellt werden muss. Falls eine Option entfernt werden sollte, müsste das auch aus dem Code und aus der dazugehörigen Businesslogik gelöscht werden. Auch müssen redundante Übergabeparameter hinzugefügt werden. Falls die Businesslogik oder die Auswahl größer werden sollte, bläht sich der Code auf. Letztendlich führt dies dazu, dass eine große Klasse entsteht, die über hunderte oder gar tausende Zeilen geht und die Entwickler schneller die Übersicht verlieren. Die Kohäsion ist niedrig, die Kopplung ist stark und der Code ist wenig flexibel. Das widersprich mehrfach den Zielen von Clean Code. Es widerspricht auch dem Depedency Inversion Principle. Die Low Level Module sind die Konsolen Zugriffe oder die Zugriffe auf die Textdatei. Sobald die Low Level Module sich ändern, muss das High Level Modul sich dem anpassen. Dies zeugt von einer Abhängigkeit. Diese direkten Abhängigkeiten sollen umgekehrt werden, indem die Module von Abstraktionen abhängig gemacht werden. Dies wird erreicht, indem Interfaces implementiert werden, auf welche zugegriffen wird. Die Interfaces IReader und IWriter werden dem Code hinzugefügt. Die konkreten Klassen ConsoleReader und ConsoleWriter sowie die jeweiligen Interfaces werden implementiert und aus dem Copy Modul gelöscht. Für TxtReader und TxtWriter gilt dasselbe.

```
public interface IReader
{
    string Read();
}

public interface IWriter
{
    void Write(string text);
}
```

Code 3: Interfaces IReader und IWriter

```
public class ConsoleReader : IReader
{
    public string Read()
    {
        return Console.ReadLine();
    }
}

public class TxtReader : IReader
{
    public string Read()
    {
        return File.ReadAllText("test.txt");
    }
}

public class ConsoleWriter : IWriter
{
    public void Write(string text)
    {
        Console.WriteLine(text);
    }
}

public class TxtWriter : IWriter
{
    public void Write(string text)
    {
        File.WriteAllText("test.txt",text);
    }
}
```

Code 4: Konkrete Klassen für IWriter und IReader

Die Low Level Module und deren Logik werden aus dem High Level Modul entfernt. Ein Problem entsteht beim Copy Modul, da dieses nicht mehr auf die vorher implementierten Funktionalitäten direkt zugreifen kann. Eine Lösung dafür bietet das Liskov Substitution Principle.

Das Copy Modul kann Objekte vom Interface IReader und IWriter beinhalten. Diese können über einen Konstruktor übergeben werden. Dieses Prinzip wird Constructor Injection [8, p. 14] genannt und ist eine Art der Dependency Injection. Weitere Möglichkeiten Werte von außen zu übergeben wären Method und Property Injection. Zusätzlich wird Copy gegen ein ICopy Interface implementiert, um auch für das Copy Modul das Dependency Inversion Principle anzuwenden, um eine höhere Wiederverwendbarkeit zu erreichen.

Es wird jeweils ein Objekt, welches IReader bzw. IWriter implementiert, deklariert. Diese werden anschließend in der ReadAndWrite Methode wieder aufgerufen und

verwendet. Dem Copy Modul ist dadurch egal, welche konkrete Implementierung hinter den Interfaces steckt.

Um Objekte über Constructor Injection übergeben zu können müssen diese zunächst erzeugt werden. Genauso muss der Konstruktoraufruf für Copy aufgerufen werden.

```
//Main & Composition Root
var writer = new ConsoleWriter();
var reader = new TxtReader();
var copy = new Copy(reader, writer);
copy.ReadAndWrite();
```

Code 5: Composition Root mittels Poor Man's DI

Diese Stelle wird Composition Root genannt. An dieser Stelle werden die Abhängigkeiten in einer Anwendung aufgelöst [8, p. 47].

Das gezeigte Beispiel Code 5 zeigt, wie ein Composition Root [8, p. 14] ohne Framework funktioniert. Die benötigten Objekte werden nacheinander erstellt und als Übergabeparameter übergeben. Dies wird „Poor Man's DI“ genannt [8, p. 7]. Bei größeren Projekten kann es unübersichtlich und das Konfigurieren komplexer werden. Hierzu gibt es einige Frameworks, welche die Orchestrierung vereinfachen. Diese werden DI Container genannt. In einem weiteren Beispiel wird das Framework AutoFac vorgestellt.

2.2.7 DI Container

Das Framework wird mittels NuGet dem Projekt hinzugefügt.

```
public static IContainer Container { get; set; }
static void Main(string[] args)
{
   var builder = new ContainerBuilder();

   builder.RegisterType<FileWriter>()
      .Named<IWriter>("TxtWriter")
      .As<IWriter>();

   builder.RegisterType<ConsoleReader>()
      .Named<IReader>("ConsoleReader")
      .As<IReader>();

   builder.RegisterType<Copy>()
      .As<ICopy>()
      .Named<ICopy>("CopyModule");

   Container = builder.Build();
   WriteSampleText();
}
```

Code 6: AutoFac Container

Über einen Builder werden die verschiedenen Module als konkrete Implementierung der jeweiligen Interfaces registriert. Ferner wird den Modulen ein Name gegeben, wodurch sie eindeutig zugeordnet werden und im weiteren Programmverlauf danach aufgelöst werden kann. Dadurch werden die konkreten Klassen für das Interface genutzt, sollte über AutoFac eine Instanz abgerufen werden. Dies geschieht in der Methode WriteSampleText.

```
public static void WriteSampleText()
{
    using (var scope = Container.BeginLifetimeScope())
    {
        var rd = scope.Resolve<IReader>();
        var wr = scope.Resolve<IWriter>();
        var copyProcess = scope.Resolve<ICopy>();
        copyProcess.CopyInputToOutput();
    }
}
```

Code 7: Eine Copy Instanz wird erstellt und ausgeführt

In dieser Methode wird ein Scope erstellt, worin ein ConsoleReader und ein TxtWriter zu einem Copy Modul orchestriert werden. Jedoch geschieht dies ohne, dass etwas Konkretes darin geschrieben wird. Stattdessen wird die bereits eingestellte Konfiguration aus der Main Methode genutzt. Der Composition Root wurde in einen separaten Scope ausgelagert und innerhalb der Methoden findet keine Businesslogik statt.

Dies hilft je nach gewähltem Architekturmodell eine saubere Schichtentrennung einzuhalten und sorgt für eine höhere Wartbarkeit und flexibleren Code.

2.3 Softwarearchitektur

Das Dependency Inversion Principle kann nicht nur auf die Programmierung, sondern auch auf Architektur und höhere Ebenen angewendet werden. Wie im vorherigen Beispiel bereits gezeigt ist es möglich die Architektur mittels Dependency Inversion zu verbessern. Als Ausgangspunkt für eine Architektur, welche gegen das Dependency Inversion Principle verstößt, wird die 3-Schichten Architektur verwendet. [9]

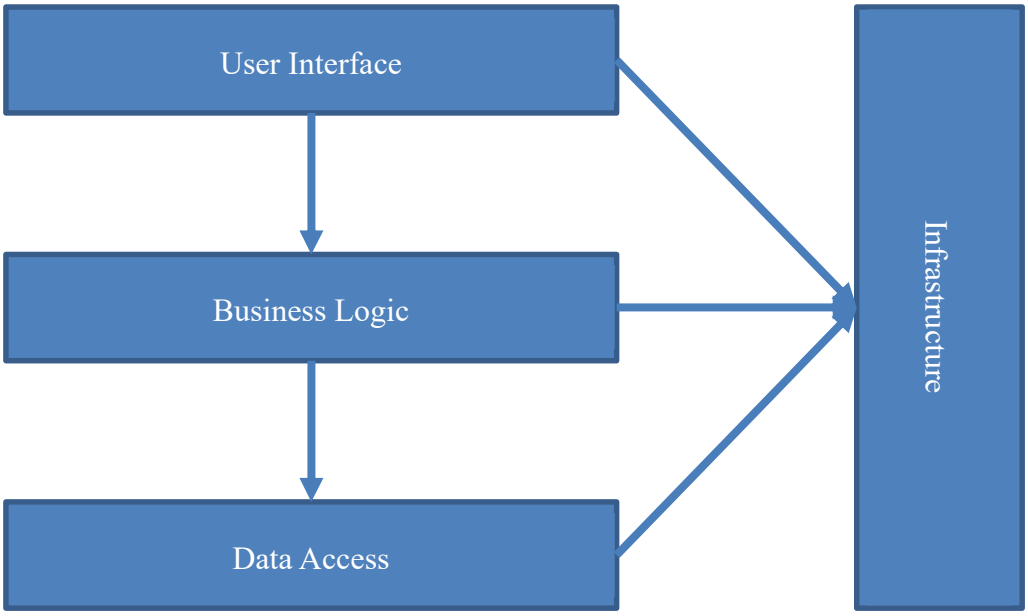

Abbildung 3: 3-Schichten Architektur

Die UI Schicht enthält Views und UI bezogene Logik.

Die Infrastructure Schicht enthält Library und Framework Code wie zum Beispiel das .Net Framework, AutoFac, Entity Framework und NUnit. Außerdem verfügt die Infrastructure Schicht über die Fähigkeit auf externe Systeme zuzugreifen wie etwa File System oder Netzwerkprotokolle.

Die Data Access Schicht enthält Code, der sich direkt auf die Persistenz bezieht wie DML Commands und Stored Procedure.

Die Business Logic Schicht enthält die Programmlogik, welche für eine fachliche Anwendung benötigt wird. Oftmals wird diese Schicht auch als Core bezeichnet, da diese den Kern einer Anwendung darstellt. Insbesondere wird dies deutlich bei der Onion Architecture oder bei der hexagonalen Architektur.

Diese 3-Schicht Architektur ist nicht konform mit dem Dependency Inversion Principle, weil bei diesen direkten Abhängigkeiten entstehen. Die UI Schicht greift direkt auf die Business Logic zu. Diese wiederum ist abhängig von der Data Access Schicht. Dies bedeutet, dass die Data Access Schicht die wichtigste Schicht ist und alles andere davon abhängig ist. Dies würde dem Dependency Inversion Principle widersprechen, da High Level Module von Low Level Modulen abhängig wären. Außerdem würde man sich von konkreten Implementierungen abhängig machen anstatt von Abstraktionen. Ohne die Data Access Schicht würden die UI und Business Logic Schicht nicht funktionieren. Dies würde auch Techniken wie Unit Testing verhindern und schadet somit der Code Qualität. Die Problematik liegt in der Abhängigkeit von der Data Access Schicht zur Business Logic Schicht. Hierzu müssen die Abhängigkeiten umgekehrt werden.

Da die Business Logic Schicht nicht mehr auf die Data Access Schicht zugreifen kann, müssen in der Business Logic Schicht Interfaces implementiert werden, welche auf eine konkrete Klasse der Data Access Schicht zugreifen. Dies kann zum Beispiel ein IRepository und ein konkretes SqlRepository sein. Zusätzlich können der Business Logic Schicht weitere Schichten über Schnittstellen angebunden werden, welche analog zur

Data Access Schicht dazu implementiert werden. Ein Beispiel wäre ein Unit Testing Modul, welches über Interfaces Mock Objekte zur Business Logic Klassen implementieren kann. Eine genauere Betrachtung zeigt, dass die neu entstandene Konstellation eine andere Architektur zur bisherigen 3-Schicht Architektur darstellt.

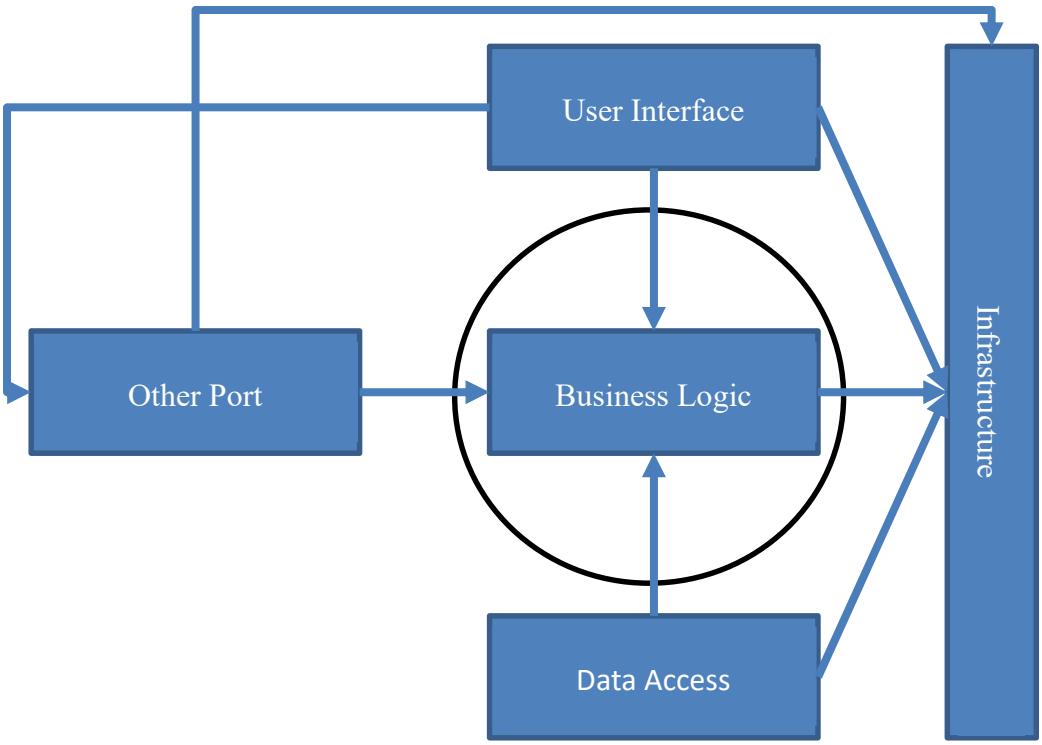

Abbildung 4: Neu entstandene Onion Architecture [12]

Die Business Logic Schicht ist zum Kern der Anwendung geworden. Danach folgen je nach Abhängigkeit weitere Schichten. Diese Architektur wird von Alistair Cockburn als hexagonale Architektur bezeichnet [10]. Jeffrey Palermo nennt dies eine Onion Architecture [11].

2.4 Nachteile von Clean Code

Die vorgestellten Techniken SOLID, DI Container und die Softwarearchitekturmodelle haben einige Nachteile. Diese werden in diesem Kapitel genauer erläutert.

2.4.1 Erweiterte Programmierkenntnisse

Es gibt neben technischen Nachteilen auch einige allgemeine Nachteile. Zum einen sind viele der Prinzipien nicht intuitiv. Bei einem Projekt, an dem mehrere Entwickler mit unterschiedlichem Hintergrund zusammenarbeiten, kann es passieren, dass einige Personen diese Prinzipien und Techniken nicht kennen. Insbesondere für Entwickler, die mit älteren Technologien arbeiten oder aus der hardwarenahen Programmierung kommen, kann dies eine Herausforderung sein. Es bedarf einer längeren Einarbeitungszeit, bis die Entwickler sinnvoll eingesetzt werden können. Nicht immer ist es möglich nur mit Entwicklern zusammenzuarbeiten, die dieses Wissen bereits haben.

2.4.2 Lesbarkeit

Ein weiterer Nachteil wäre, dass das Debugging oftmals schwererer sein kann als bei anderen Ansätzen. Dies ist dem geschuldet, dass sich zum einen die Anzahl der Klassen erhöht und es dadurch unübersichtlicher werden kann, da oft zwischen verschiedenen Klassen gesprungen werden muss. Auch ist nicht immer direkt ersichtlich, wo ein Aufruf stattfindet und welche Werte übergeben werden.

2.4.3 Performance

Neben allgemeinen Nachteilen gibt es noch technische Nachteile. Ein Nachteil ist die Performance. Guter Code wurde bisher nur an Wartbarkeit, Testbarkeit, Erweiterbarkeit und Lesbarkeit gemessen.

Es gibt Fälle, wo Performance eine wichtige Rolle spielt. Dies kann bei zeitkritischen Anwendungen wie Big Data, Spiele oder Machine Learning der Fall sein. [12, pp. 959-960]. Außerdem sollte beachtet werden, dass Anwendungen auch skalieren können und dort die Performance zu einem Problem werden könnte. Das Problem der Clean Code Techniken und Prinzipien ist, dass diese zu einem Teil dem performanten Code widersprechen und zum anderen Teil nicht beachtet werden. Relevant ist ebenfalls, dass Code nicht immer gut wird durch die Einhaltung der Prinzipien.

Das bereits vorgestellte Framework AutoFac und andere DI Container können bezüglich der Performance ein Problem sein. Diese arbeiten intern über Reflection [13], welches sehr rechenintensiv ist und sich somit negativ auf die Performance auswirken kann.

Auch die Lesbarkeit kann im Zielkonflikt mit performantem Code stehen. Zum einen kann performanter Code schwer zu lesen sein, da zum Beispiel mit Pointern gearbeitet werden kann [12, p. 952]. Zum anderen sollte mit Beachtung des Garbage Collectors programmiert werden. Dies schließt ein, dass bestimmte Techniken wie Thread Erstellung, Exceptions oder I/O Operations vermieden oder nur mit Bedacht eingesetzt werden sollen, da diese rechenintensiv sind. Nach Clean Code Prinzipien stellen diese kein Problem dar. Für die Performance hingegen ist das sehr wichtig. Insbesondere eine klare Trennung zwischen In-Memory Operations und I/O Zugriffe ist für Performance essenziell. Dazu wurde ein Benchmark [12, 14] durchgeführt, um dies zu verdeutlichen.

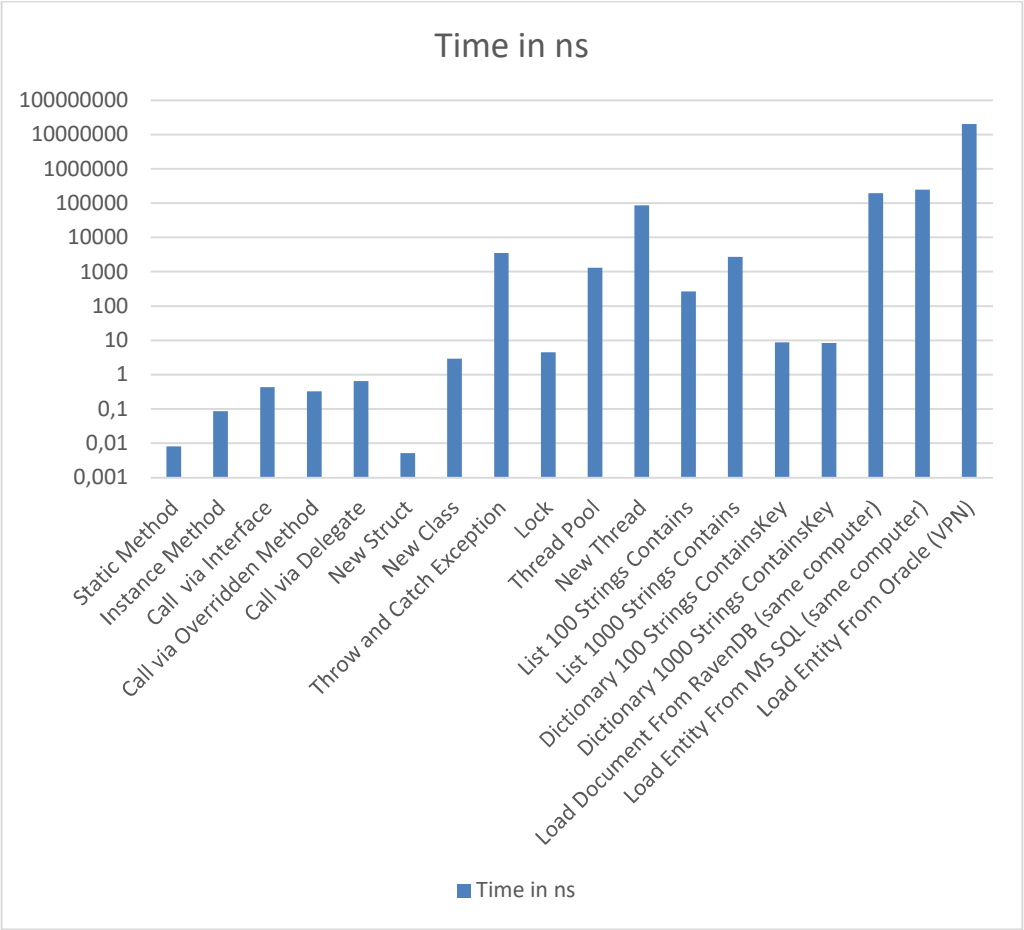

Abbildung 5: Benchmark Ergebnisse mit logarithmischer Skalierung [12]

2.4.4 Programmierparadigmen

Ein weiterer Aspekt ist, dass sich viele der Clean Code Prinzipien stark auf die Objektorientierung beziehen. Es gibt Fälle, welche mit einem anderen Paradigma besser lösbar sind als mit objektorientierter Programmierung. C# bietet neben objektorientierten Ansätzen auch Möglichkeiten in anderen Paradigmen wie der funktionalen Programmierung zu entwickeln.

Ein Beispiel für einen solchen Fall wäre eine Filterfunktion, welche alle geraden oder ungeraden Zahlen aussortiert [15]. Eine Möglichkeit wäre gegen Clean Code eine große Filterklasse zu schreiben, welche über Switch-Case oder If-Abfragen die gewünschten Filter auswählt. Ein nach SOLID programmierter Code würde ein IFilter Interface anbieten, welches mehrere konkrete Filter Klassen implementieren.

```
public interface IFilter
{
    IEnumerable<int> Execute(IEnumerable<int> myArray);
}
```

Code 8: IFilter Interface mit einer Execute Methode

```
public class FilterEven:IFilter
{
   public IEnumerable<int> Execute(IEnumerable<int> myArray)
   {
      foreach (var i in myArray)
      {
         if (i % 2 == 0)
         {
            yield return i;
         }
      }
   }
}

public class FilterOdd:IFilter
{
   public IEnumerable<int> Execute(IEnumerable<int> myArray)
   {
      foreach (var i in myArray)
      {
         if (i % 2 != 0)
         {
            yield return i;
         }
      }
   }
}
```

Code 9: Konkrete Filterklasse

Es muss für jede Filter Variation eine eigene Klasse erstellt werden. Die Anzahl der Klassen würde dadurch steigen. Ebenso würde auch die Menge an Code steigen. Dies würde sich schlecht auf die Wartbarkeit auswirken. Die benötigte Entwicklungszeit würde steigen und das Debugging und Testing würde schwerer fallen, da es eine Vielzahl von Klassen gibt, welche eine Fehlerquelle sein könnten. Es würde auch gegen das Don't Repeat Yourself Prinzip verstoßen, da es zu viel Code Duplikation kommt.

```
public static void Main(string [] args)
{
   var myArray= new []{1,2,3,4,5,6,7,8,9};
   IFilter filter = new FilterOdd();
   var filteredArray= filter.Execute(myArray);
   foreach (var i in filteredArray)
   {
      Console.WriteLine(i);
   }
}
```

Code 10: Composition Root für ein Filter Beispiel

Zu einer besseren Lösung würde hier ein funktionaler Ansatz führen. Statt viele Klassen und Objekte zu programmieren, würde hier eine generische Methode reichen, welche eine Funktion als Übergabeparameter erhält.

```
private static IEnumerable<T> Filter<T>(
   IEnumerable<T> source,
   Func<T,bool> filterCondition
   )
{
   foreach (var i in source)
   {
      if (filterCondition(i))
      {
         yield return i;
      }
   }
}
```

Code 11: Generische Filter Methode mit einer Funktion als Übergabeparameter

Ferner ermöglicht diese Art der Programmierung, dass nicht gegen einen konkreten Typ programmiert werden muss. Stattdessen kann gegen einen generischen Typ T programmiert werden. Die führt zu besserem Code, da nicht für jeden Datentypen eine

```
public static void Main(string [] args)
{
   var myArray= new []{1,2,3,4,5,6,7,8,9};
   foreach (var i in Filter(myArray, i => i%2==0 ))
   {
      Console.WriteLine(i);
   }
}
```

Code 12: Main Methode für einen Filter mit funktionaler Programmierung

eigene Filterfunktion geschrieben werden muss. Auch die Lesbarkeit in der Main Methode wird dadurch verbessert. Außerdem ist keine Erstellung von Objekten mehr nötig, um zu filtern.

Die Firma Microsoft, die die Sprache C# erfunden hat und immer noch weiter einwickelt, hat diese Möglichkeit der funktionalen Programmierung in C# bereits eingebaut. LINQ ist ein solches Sprachfeature und bietet die Möglichkeit mittels funktionaler Programmierung Abfragen für Objekte von IEnumerable zu erstellen [16].

3 Fazit

In dieser Seminararbeit wurde untersucht, inwiefern durch Clean Code Techniken die Codequalität verbessert werden kann. Dies wurde anhand von Beispielen in C# demonstriert.

Obwohl die Beispiele zeigen, dass Clean Code die Codequalität verbessern kann, lässt sich keine generelle Empfehlung ableiten, da Clean Code nicht immer geeignet ist oder nicht alle Probleme löst. Vielmehr sollten die Techniken eine von vielen Werkzeugen für einen Softwareentwickler sein, um Probleme zu lösen. Ein erfahrener Entwickler sollte verschiedene Techniken beherrschen und bei einem meist nicht trivialen Problem genau die Techniken nutzen, die eine möglichst einfache Lösung bieten.

Insbesondere das Dependency Inversion Principle ändert die Art und Weise der Strukturierung von Code deutlich. Dadurch wird es möglich, Code zu entkoppeln, um eine lose Kopplung und starke Kohäsion zu erreichen. Die Flexibilität des Codes steigert sich, ebenso wie die Wiederverwendbarkeit. Die Lesbarkeit wird nicht immer durch Clean Code Techniken verbessert. Die Testbarkeit wird durch Clean Code verbessert und eröffnet somit die Möglichkeit für weitere Techniken wie Test Driven Development.

Die Frage, inwiefern Clean Code Auswirkungen auf andere Programmiersprachen und Paradigmen hat, könnte Ansatz für weitere Forschung sein.

vi. Literaturverzeichnis

References

[1] R. C. Martin, *Clean code: A handbook of agile software craftsmanship*. Upper Saddle River NJ: Prentice Hall, 2009.

[2] M. Fowler, *Refactoring: Improving the design of existing code / Martin Fowler*. Reading, MA: Addison-Wesley, 1999.

[3] R. C. Martin and K. Henney, *Clean Architecture: A craftsman's guide to software structure and design*. Boston, Columbus, Indianapolis, New York, San Francisco, Amsterdam, Cape Town, Dubai, London, Madrid, Milan, Munich, Paris, Montreal, Toronto, Delhi, Mexico City, São Paulo, Sydney, Hong Kong, Seoul, Singapore, Taipei, Tokyo: Prentice Hall, 2018.

[4] *ArticleS.UncleBob.PrinciplesOfOod.* [Online]. Available: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (accessed: Aug. 30 2022).

[5] Gamma et al., *Design patterns: Elements of reusable object-oriented software,* 39th ed. Boston, Mass., Munich: Addison-Wesley, 2011.

[6] Robert C. Martin, *Design Principles and Design Patterns.* [Online]. Available: http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf

[7] M. Noback, "The Dependency Inversion Principle," in pp. 67–104. [Online]. Available: https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf

[8] M. Seemann, *Dependency injection in .NET*. Greenwich, Conn.: Manning; London : Pearson Education [distributor], 2011.

[9] Rick-Anderson, *Creating a Business Logic Layer (C#).* [Online]. Available: https://docs.microsoft.com/en-us/aspnet/web-forms/overview/data-access/introduction/creating-a-business-logic-layer-cs (accessed: Aug. 30 2022).

[10] Alistair Cockburn, *Hexagonal architecture.* [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/ (accessed: Aug. 30 2022).

[11] J. Palermo, "The Onion Architecture : part 1," *Programming with Palermo*, 29 Jul., 2008. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ (accessed: Aug. 30 2022).

[12] K. Kokosa, *Pro .NET Memory Management: For Better Code, Performance, and Scalability*. Berkeley, CA: Apress; Imprint,Apress, 2018.

[13] *Registration Concepts — Autofac 6.0.0 documentation.* [Online]. Available: https://autofac.readthedocs.io/en/latest/register/registration.html (accessed: Aug. 30 2022).

[14] *GitHub - feO2x/DndgGraz.DesignPrinciplesVsPerformance: Slides and example code for my Design Principles vs. Performance talk at the .NET Developer Group Graz in March 2021.* [Online]. Available: https://github.com/feO2x/DndgGraz.DesignPrinciplesVsPerformance (accessed: Aug. 31 2022).

[15] *Besserer C#-Code durch funktionale Programmierung | Rainer Stropek - YouTube.* [Online]. Available: https://www.youtube.com/watch?v=8vQ15jddRBM&t=3679s (accessed: Aug. 31 2022).

[16] BillWagner, *Language-Integrated Query (LINQ) (C#)*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (accessed: Aug. 31 2022).