FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

Objektbasierte 1:n-Navigation: Ein Vergleich zwischen externer Iteration und LINQ

Abschlussarbeit im Studiengang Bachelor of Science im Fach Informatik

vorgelegt am 21. Oktober 2009 von

Jens Duczmal
Konrad-Adenauer-Allee 18
28329 Bremen
Matrikelnummer: 6580076
Duczmal@gmx.de

betreut durch
Prof. Dr. Friedrich Steimann

Inhaltsangabe

In der objektorientierten Programmierung werden Objekte häufig in Auflistungen unterschiedlichen Typs verwaltet. Der Zugriff auf diese Objekte geschieht mit Iterationen in Form von Schleifen, durch Indizes oder über Enumeratoren.

Microsoft® stellt im Rahmen des .NET-Framework 3.5 eine Alternative zur Verfügung: Die **L**anguage **IN**tegrated **Q**uery. Im Rahmen dieser Arbeit werden die Möglichkeiten der objektbasierten 1:n-Navigation durch die sprachintegrierte Abfrage dem klassisch iterativen Modell gegenübergestellt, sowie Vor- und Nachteile beider Techniken diskutiert.

Inhaltsverzeichnis

Inhaltsangabe I

Inhaltsverzeichnis II

1 Einleitung 1
- 1.1 Motivation ... 1
- 1.2 Verwendete Programme ... 1
- 1.3 Aufbau der Arbeit ... 2

2 LINQ – Language INtegrated Query 3
- 2.1 Implementierungen ... 3
- 2.2 Sprachkonzepte als Grundlage für LINQ ... 4

3 LINQ to Objects 6
- 3.1 Syntax ... 7
  - 3.1.1 Abfragesyntax ... 7
  - 3.1.2 Erweiterungsmethodensyntax ... 7
  - 3.1.3 Typinferenz und anonyme Typen ... 8
- 3.2 Verzögerte Ausführung am Beispiel von `Where` ... 9
- 3.3 Standardabfrageoperatoren ... 10
- 3.4 `SelectMany`-Operator ... 11
- 3.5 `ToList`-Operator ... 12
- 3.6 `Cast`-Operator ... 14
- 3.7 `Sum`- / `Count`-Operator ... 14
- 3.8 Eigene Erweiterungsmethoden ... 14
- 3.9 Abfragesyntax vs. Methodensyntax ... 16

4 Fallstudie: TheaterMaxx 17
- 4.1 Klassenmodelle ... 18
  - 4.1.1 Klassenmodell `stamm` ... 18
  - 4.1.2 Klassenmodell `inszenierung` ... 18
  - 4.1.3 Klassenmodell `stueck` ... 19
  - 4.1.4 Klassenmodell `termin` ... 20
- 4.2 Metriken ... 21
- 4.3 Verwendungsmuster in der Fallstudie ... 23
  - 4.3.1 Direkte Iteration ... 24
  - 4.3.2 Gefilterte Iteration ... 24
  - 4.3.3 Abfrage des ersten Elementes ... 25
  - 4.3.4 Summen bilden und/oder Anzahl von Elementen ermitteln ... 26
  - 4.3.5 Geschachtelte Iteration ... 28
  - 4.3.6 Vereinigungsmengen ... 31
  - 4.3.7 Vorzeitiger Iterationsabbruch ... 32
  - 4.3.8 Aufeinanderfolgende Elemente vergleichen ... 33
  - 4.3.9 Algorithmus „Stabile Heirat“ ... 34

4.4 Weitere Verwendungsmuster 35
4.4.1 Gruppierungen 35
4.4.2 Sortierte Gruppierungen 36
4.4.3 Maximum ermitteln 37
4.4.4 Element mit Maximum ermitteln 38
4.5 Bidirektionale Navigation 40
4.6 Messergebnisse: Gesamtübersicht 43

5 Ergebnisse der Fallstudie 45
5.1 Zusammenfassung und Interpretation der Ergebnisse 45
5.2 Erfahrungen und Hinweise 47

6 Schlussbetrachtungen und Ausblick 49
6.1 Zusammenfassung 49
6.2 Ausblick 49

A Quelltext Fallstudie TheaterMaxx 50
A.1. Klasse: Inszenierung 50
A.2. Klasse: StabileHeirat 54
A.3. Klasse: Ensemble 56
A.4. Klasse: Person 57
A.5. Klasse: Darsteller 57
A.6. Klasse: Rollenprofil 57
A.7. Klasse: Stück 58
A.8. Klasse: Akt 59
A.9. Klasse: Szene 60
A.10. Klasse: Terminplan 61
A.11. Klasse: Probentermin 63
A.12. Klasse: Spieltermin 63

B Inhalt der beiliegenden CD 64

Quelltextverzeichnis IV

Tabellenverzeichnis V

Abbildungsverzeichnis V

Literaturverzeichnis VI

Eidesstattliche Erklärung VII

1 Einleitung

1.1 Motivation

In der objektorientierten Programmierung werden Objekte häufig in Auflistungen unterschiedlichen Typs verwaltet. Der Zugriff auf diese Objekte geschieht mit Iterationen in Form von Schleifen, durch Indizes oder über Enumeratoren. Sollen Objekte mit bestimmten Merkmalen herausgesucht und ebenfalls als Auflistung zurückgegeben werden, führt dies oft zu unschönem „Boilerplate-Code“[1]. Ebenso können Schleifen, in denen die Objekte nach verschiedenen Kriterien gefiltert oder unterschiedlich behandelt werden, verhältnismäßig komplex werden, vor allem wenn die Objekte wiederum Auflistungen weiterer Objekte beinhalten.

Microsoft® stellt seit dem .NET-Framework 3.5 für die Programmiersprachen C# und VB.NET eine Komponente zum Abfragen von Datenquellen zur Verfügung: die **L**anguage **IN**tegrated **Q**uery (kurz: LINQ). Die sprachintegrierte Abfrage ermöglicht die Formulierung von Abfragen als Quellcode, im Gegensatz zu Datenbankabfragen, bei denen diese als Zeichenfolge hinterlegt sind und nicht vom Compiler überprüft werden können.

Gegenstand dieser Arbeit ist es, die klassische iterative Vorgehensweise der **L**anguage **IN**tegrated **Q**uery gegenüberzustellen und zu ermitteln, inwieweit unnötiger „Boilerplate-Code“ vermieden werden kann. Beide Techniken werden anhand verschiedener Verwendungsmuster auf Vor- und Nachteile in den Bereichen Iterationen, Geschwindigkeit, Wartbarkeit und Anwendbarkeit miteinander verglichen und die Ergebnisse ausführlich diskutiert. Zudem wird auf die Frage der Unterstützung bidirektionaler Navigation durch LINQ eingegangen.

1.2 Verwendete Programme

- Microsoft® Visual Studio Team System 2008
  Eine Entwicklungsumgebung für das .NET-Framework

- .NET-Reflector® Version 5.1.6.0[2]
  Eine Software zum Dekompilieren und Analysieren von .NET-Assemblies, um den Quellcode in der C#-Syntax zu rekonstruieren.

- NClass Version 2.02[3]
  Eine Open-Source-Software zur Erstellung von Klassendiagrammen

[1] Unter „Boilerplate-Code“ versteht man Codefragmente, die an vielen Stellen in mehr oder weniger unveränderter Form benötigt werden.
[2] Vorstellung und Download unter http://www.red-gate.com/products/reflector/
[3] Vorstellung und Download unter http://nclass.sourceforge.net/

1.3 Aufbau der Arbeit

Zunächst stellt Kapitel 2 die verschiedenen LINQ-Implementierungen kurz vor und erläutert die Sprachkonzepte, auf denen die Language Integrated Query basiert.

Kapitel 3 beschäftigt sich mit LINQ to Objects, der Implementierung die Hauptgegenstand dieser Arbeit ist. Es wird zunächst die allgemeine Syntax erklärt und auf das Merkmal der „verzögerten Ausführung“ genauer eingegangen. Vervollständigt wird das Kapitel mit einer Auflistung aller Standardabfrageoperatoren die LINQ derzeit anbietet, von denen einige exemplarisch näher erläutert werden.

In Kapitel 4 folgt eine Fallstudie, basierend auf der Software „TheaterMaxx“ die im Rahmen des Programmierpraktikums an der FernUniversität in Hagen entwickelt wurde. Kernstück der Fallstudie ist die Transformation von Methoden mit klassischer Iteration in LINQ-Abfragen. Die Abfragen werden nach Verwendungsmustern gegliedert und es wird mit verschiedenen Metriken versucht, die Vor- und Nachteile beider Vorgehensweisen zu ermitteln. Zudem wird auf weitere Verwendungsmuster eingegangen die nicht Bestandteil der TheaterMaxx-Software sind, jedoch häufiger vorkommen können und ebenso wird geprüft, inwiefern bidirektionale Navigation von LINQ unterstützt wird.

Die Auswertung der Ergebnisse dieser Fallstudie wird im Kapitel 5 besprochen. Kapitel 6 schließt die Arbeit mit einer kurzen Zusammenfassung und einem Ausblick ab.

2 LINQ – Language INtegrated Query

Zunächst werden die verschiedenen Implementierungen der sprachintegrierten Abfrage aufgeführt. Im anschließenden Abschnitt wird detaillierter auf die Sprachkonzepte eingegangen, denen LINQ zugrunde liegt.

2.1 Implementierungen

Die Abfragesprache LINQ umfasst einen Satz von Erweiterungsmethoden, der die sprachintegrierte Abfrage auf Daten ermöglicht. Sie wurde mit dem .NET-Framework 3.5 bzw. der Entwicklungsumgebung „Microsoft Visual Studio 2008“ eingeführt.

LINQ ermöglicht den standardisierten Zugriff auf nahezu beliebige Arten von Daten wie z.B. Datenbanktabellen, XML-Dateien, speicherresidente Datenstrukturen, dem Dateisystem, Textdateien etc. Diese Art des Zugriffes steht allen .NET-konformen Sprachen wie z.B. C# und VB.NET zur Verfügung.

Für die verschiedenen Datenquellen existieren mehrere LINQ-Implementierungen, die jeweils im .NET-Framework realisiert sind:

LINQ to Objects bildet die Basis aller LINQ-Abfragen und ist im Namensraum `System.Linq` verankert. LINQ to Objects ermöglicht die Navigation auf allen internen oder benutzerdefinierten Datenstrukturen, welche die Schnittstelle `IEnumerable` oder die generische Variante `IEnumerable<T>` implementieren.

LINQ to XML bietet eine Programmierschnittstelle für XML im Arbeitsspeicher, die das LINQ-Framework nutzt.

LINQ to SQL ist ein Provider von Microsoft®, mit dem LINQ-Abfragen auf Microsoft® SQL-Server-Datenbanken ausgeführt werden können.

LINQ to ADO.NET ermöglicht Abfragen auf ADO.NET-Objekte[4] und unterteilt sich wiederum in drei Komponenten: LINQ to DataSet, LINQ to SQL und LINQ to Entities. Über diese Schnittstelle ist es theoretisch möglich auf beliebige Datenbanken zuzugreifen, vorausgesetzt es ist ein geeigneter Provider verfügbar.

Gegenstand dieser Arbeit ist ausschließlich LINQ to Objects, welches das Fundament für weitere LINQ-Techniken bietet. Wann immer auf den folgenden Seiten von LINQ gesprochen wird, ist LINQ to Objects gemeint, sofern es nicht anders erwähnt wird.

[4] ADO.NET ist eine Sammlung von Klassen im .NET-Framework um den Zugriff auf relationale Datenbanken zu gewährleisten.

2.2 Sprachkonzepte als Grundlage für LINQ

Language INtegrated Query baut auf vielen neuen Sprachkonzepten auf, welche mit .NET 2.0 bzw. NET 3.0 eingeführt wurden und einen kleinen Schritt in Richtung funktionaler Programmierung darstellen.

Um die Syntax und die Verwendung von LINQ besser verstehen zu können, soll zunächst kurz auf die neuen Konzepte eingegangen werden. Diese gliedern sich in fünf verschiedene Bereiche:

Implizit typisierte, lokale Variablen. Dieses Konzept wird als Typinferenz bezeichnet und ermöglicht die Deklaration von lokalen Variablen ohne einen expliziten Typ dafür angeben zu müssen. Der Compiler ermittelt den korrekten Typ dann automatisch. Die Variable verhält sich exakt so, als wäre er explizit deklariert worden.
Beispiel: `var name = „Duczmal“` wird hier als `String` erkannt und kompiliert.

Vereinfachte Objektinstanziierung ermöglicht die Instanziierung neuer Objekte durch Übergabe von Startwerten im Standardkonstruktor, ohne dass dieser für die Parameterübergabe überladen werden muss.
Beispiel: `var person = new Person () {Name = „Duczmal“}`

Anonyme Typen sind Typen, die automatisch aus der vereinfachten Objektinstanziierung erstellt und abgeleitet werden. So kann ein neues Objekt mit einem anonymen Typ, z.B. durch

```
var anonymePerson=new {Name=„Duczmal”,Ort=„Bremen}
```

erstellt werden. Dem Objekt werden die Parameter `Name` und `Ort` automatisch zugeordnet. Weitere Objekte mit denselben Parametern und derselben Reihenfolge sind vom gleichen anonymen Typ.

Erweiterungsmethoden. Bis zur Einführung der Erweiterungsmethoden galt in objektorientierten Programmiersprachen die Regel, dass neue Methoden einer Klasse auch in dieser definiert werden müssen. Erweiterungsmethoden umgehen diese Beschränkung.

Angenommen, es sollen alle Klassen um eine Methode „Ausgeben“ erweitert werden, um z.B. den Inhalt des Objektes auf der Konsole auszugeben. Anstatt sämtliche Klassen zu überarbeiten, reicht es aus, die Methode in einer statischen Klasse zu deklarieren:

```
static void Ausgeben(this Object obj)
```

Das Schlüsselwort `this` kennzeichnet, dass es sich hier um eine Erweiterungsmethode auf dem Typ `Object` handelt.

Da `Object` den Stamm der Typenhierarchie darstellt, ist die Methode auf allen Variablen und Objekten verfügbar – somit kann durchaus von einer Möglichkeit der generellen Syntaxerweiterung von C# gesprochen werden, wobei diese Technik allen .NET-konformen Sprachen zur Verfügung steht.

Erweiterungsmethoden können beliebige Typen erweitern, sogar solche für die der Quellcode nicht vorliegt, wobei dieser Fall sehr vorsichtig genutzt werden sollte.

LINQ bietet eine Reihe von Standardabfrageoperatoren, die auf `IEnumerable` deklariert sind und daher sämtliche Auflistungen, die von diesem Typ ableiten, um zusätzliche Befehle erweitern. [5]

Lambda-Ausdrücke.[6] Die Bezeichnung ist an dem Lambda-Kalkül angelehnt, welcher die theoretische Grundlage vieler funktionaler Programmiersprachen bildet.[7] Ein Lambda-Ausdruck besteht aus Eingabeparametern, dem Zeichen => und einem Ausdruck bzw. Anweisungsblock. Er ersetzt den Aufruf eines Delegaten mit einer anonymen Methode. Syntaxbeispiel:

```
Func<int, int, long> addition = (x, y) => x + y;
long result = addition(4, 5);
```

Die Variable `addition` ist ein im Namenraum `System.Core` definierter `Delegate`. Er erwartet zwei Eingabeparameter vom Typ Integer (`x,y`) und einen Rückgabewert vom Typ Long (`x+y`).

Weitergehende Informationen finden sich in [MIC09], [MAR08] und [KUE08].

Zusammenfassend bietet die Language INtegrated Query folgende Vorteile:

- Vermeidung von stets gleichem Quellcode („Boilerplate-Code“)
- einheitliche Sprachsyntax, trotz Verwendung verschiedener Datenquellen
- Umstellung von speicherresidenten Auflistungen auf Datenbanken, ohne umfangreiche Änderungen im Quellcode
- „Sprachunabhängigkeit“[8]

[5] Siehe dazu Abschnitt 3.3
[6] Mehr Informationen zu Lambda-Ausdrücken in C# in [MIC09].
[7] Beim Lambda-Kalkül wird der Ausdruck als auswertbare Funktion betrachtet, so dass Funktionen als Parameter übergeben werden können.
[8] Die Sprachenunabhängigkeit bezieht sich nur auf Sprachen die vom .NET-Framework unterstützt werden.

3 LINQ to Objects

Wie bereits im Kapitel 2 erwähnt, bildet LINQ to Objects die Grundlage für die sprachintegrierte Abfrage und ist seit C# 3.0 fester Bestandteil des Microsoft® .NET-Framework.

LINQ-Abfragen ähneln der SQL-Syntax und ermöglichen die Formulierung selbiger in reinem Code, anstatt mit Zeichenketten. Die Vorteile dabei sind, dass der Code vom Compiler geprüft und optimiert werden kann und zudem von der automatischen Codevervollständigung „IntelliSense“[9] unterstützt wird.

Diese Abfragen können auf alle Auflistungen angewandt werden, welche die Schnittstelle `IEnumerable` implementieren. Da `ICollection` wiederum `IEnumerable` implementiert, können LINQ-Abfragen auf nahezu allen Datenstrukturen ausgeführt werden. Darunter fallen z.B. `String`, `Array`, `Queue`, `Stack`, `Dictionary`, `HashSet`, `List`, `LinkedList` etc. sowie ihre generischen Ableitungen.

Um dem Leser ein möglichst großes Verständnis für die im weiteren Verlauf getätigte Untersuchung zu ermöglichen, werden im Folgenden die wichtigsten Konzepte von LINQ eingeführt:

- Abschnitt 3.1 verdeutlicht die allgemeine Syntax von LINQ-Abfragen.
- Abschnitt 3.2 erklärt die Systematik der „verzögerten Ausführung“ von Abfragen anhand der Operatoren `Select` und `Where`.
- Abschnitt 3.3 zeigt eine Auflistung aller verfügbaren Standardabfrageoperatoren. In den Abschnitten 3.4 bis 3.8 wird auf einige näher eingegangen wird.

[9] Weitere Informationen zu IntelliSense in [MIC07].

3.1 Syntax

3.1.1 Abfragesyntax

Wenn man davon ausgeht, dass eine Auflistung `Personen`[10] vorliegt und die Namen aller Personen ausgegeben werden sollen, die älter als 30 Jahre sind, lässt sich dieses mit folgender LINQ-**Abfragesyntax** ausdrücken:

```
1   IEnumerable<String> query =
2                               from p in Personen
3                               where p.Alter > 30
4                               select p.Name;

6   foreach (String s in query)
7       Console.WriteLine(s);
```

Quelltext 3-1: Beispiel – Abfragesyntax

Zunächst wird in Zeile 1 eine neue Auflistung `query` deklariert, um die Ergebnisse der Abfrage aufnehmen zu können. In diesem Fall ist sie vom Typ `IEnumerable<String>`, da lediglich die Namen zurückgegeben werden sollen.

In Zeile 2 wird mit `from p in Personen` bestimmt, dass die Auflistung `Personen` durchlaufen und die einzelnen Elemente der Bereichsvariablen `p` zugeordnet werden sollen. .NET erkennt hier bereits automatisch, dass es sich bei der Quelle um eine Auflistung von Personen(-Typen) handelt. Entsprechend ist `p` vom Typ `Person`. Auf `p` kann nun in weiteren Operatoren immer wieder Bezug genommen werden.

Mit Hilfe des Schlüsselwortes `where` wird in Zeile 3 festgelegt, dass nur Personen mit einem Alter größer als 30 gesucht werden. Wegen der Typinferenz aus Zeile 2 bietet die Codevervollständigung „IntelliSense“ während der Eingabe automatisch die Eigenschaften aus der Klasse `Person` an.

`Select` in Zeile 4 ist ein sog. Projektionsoperator. Er speichert die Ergebnisse der Abfrage in einer Auflistung vom Typ `IEnumerable<T>`. Der `Select`-Operator projiziert in dieser Abfrage `p.Name`, daher ist `T` vom Typ `String`.

3.1.2 Erweiterungsmethodensyntax

In den Sprachkonzepten, im Abschnitt 2.2, wurde erklärt, dass Lambda-Ausdrücke eine Grundlage für LINQ bilden. Dieselbe Abfrage lässt sich auch in der **Erweiterungsmethodensyntax** formulieren, die im Folgenden nur noch als Methodensyntax bezeichnet wird:

```
1   IEnumerable<String> query =
2                               Personen
3                               .Where(p => p.Alter > 30)
4                               .Select(p => p.Name);
```

Quelltext 3-2: Beispiel – Methodensyntax

[10] Hier vom Typ `List<Person>`.

In dieser Syntax wird viel deutlicher, dass es sich bei den Operatoren `Where` und `Select` tatsächlich um Erweiterungsmethoden handelt, denen ein Lambda-Ausdruck übergeben wird. `Personen` ist vom Typ `List<Person>` und implementiert damit `IEnumerable`. Deshalb sind beide Erweiterungsmethoden auf der Auflistung `Personen` verfügbar.

In Zeile 3 wird erneut der Filter definiert, jedoch als Lambda-Ausdruck. Zur Wiederholung: `(Eingabeparameter) => Ausdruck oder Anweisungsblock`
Der Eingabeparameter ist hier die Bereichsvariable `p`, welche automatisch als `Person` erkannt wird. Der Ausdruck `p.Alter > 30` legt die Bedingung fest, die erfüllt werden soll, damit die Elemente selektiert werden können.

Der Ausdruck in Zeile 4 bestimmt, dass nur die Namen der Personen zurückgegeben werden.

3.1.3 Typinferenz und anonyme Typen

Mit dem `Select`-Operator können nicht nur einzelne Eigenschaften eines Typs, wie z.B. der Name, zurückgegeben werden, sondern auch der gesamte Typ (durch `p => p`) oder mehrere Eigenschaften. Soll die Abfrage nicht nur die Namen der Personen, sondern zusätzlich ihren Wohnort enthalten, lässt sich dies wie folgt ausdrücken:

```
1   var query = Personen
2               .Where(p => p.Alter > 30)
3               .Select(p => new {Nachname = p.Nachname, Wohnort = p.Ort});

5   foreach (var p in query)
6       Console.WriteLine(p.Nachname + " " + p.Wohnort);
```

Quelltext 3-3: Beispiel – Select mit anonymem Typ

In Zeile 3 wird ein neuer, anonymer Typ als Rückgabewert für die Auflistung festgelegt. Aus diesem Grund ist die `query` auch mit dem Schlüsselwort `var` deklariert worden.

Zusätzlich gibt es die Möglichkeit, einzelne Eigenschaften umzubenennen, damit ggfs. Namenskonflikte vermieden werden. In diesem Beispiel wird die Eigenschaft `Ort`, welche auf dem Typ `Person` definiert ist, innerhalb der Elemente von `query` in `Wohnort` umbenannt, aus `Name` wird `Nachname`. Die Laufvariable `p` in Zeile 6 wird nun über die Eigenschaften, die in der `Select`-Anweisung in Zeile 3 festgelegt wurden, angesprochen.

3.2 Verzögerte Ausführung am Beispiel von `Where`

Eine mit LINQ definierte Abfrage wird nicht sofort bei der Deklaration ausgeführt, sondern erst bei ihrer Verwendung. Dies geschieht unter anderem durch:

- die Iteration mit `foreach`
- die Iteration über `GetEnumerator()` und `MoveNext()`
- eine Umwandlung per `ToList()`
- Aggregatsionsfunktionen wie `Sum()` oder `Count()`
- die Sortierung mit `OrderBy()`

Verantwortlich für die „verzögerte Ausführung“ ist das `Yield`-Schlüsselwort – genauer gesagt, die `yield return`-Anweisung. `Yield` wird innerhalb eines Iterator-Blocks eingesetzt und liefert entweder einen Wert zurück oder signalisiert das Ende der Iteration. Die Implementierung von `Where` im .NET-Framework, nach [HEI07], verdeutlicht dies:

```
1    Namespace System.Linq {
2        using System;
3        using System.Collections.Generic;
4        public static class Enumerable {
5            public static IEnumerable<T> Where<T>(
                     this IEnumerable<T> source,
                     Func<T, bool> predicate) {

6                foreach (T item in source)
7                    if (predicate(item))
8                        yield return item;
9            }
10       }
11   }
```

Quelltext 3-4: Die Erweiterungsmethode `Where`

`Where` liefert als Rückgabewert eine Auflistung vom Typ `IEnumerable<T>` und ist als generische Erweiterungsmethode von `IEnumerable<T>` deklariert, dargestellt durch das Schlüsselwort `this` im ersten Parameter. Der zweite Parameter, `Func<T, bool>`, stellt einen generischen Delegaten dar. Dieser kann auf eine Methode zeigen, die einen generischen Typen `T` entgegennimmt und einen booleschen Rückgabewert liefert, wie im Unterabschnitt 3.1.2 vorgestellt:

```
Func<T, bool> predicate = p => p.Alter > 30
```

Die Erweiterungsmethode iteriert per `foreach` über die `IEnumerable`-Auflistung `source`. Für jedes Element wird geprüft, ob der generische Delegat `predicate` `true` zurückgibt. Ist dies der Fall, wird das Element mit `yield return` zurückgegeben.

Interessant an `yield return` ist, dass die Ausführung unmittelbar an die aufrufende Methode zurückgegeben wird. Erst beim nächsten Aufruf der Erweiterungsmethode durch die externe Iteration, wird `Where` exakt an der unterbrochenen Position weiter

ausgeführt, d.h. erst dann läuft die Iteration auf `source` weiter. Dies gilt auch für verkettete Aufrufe. Weitergehende Beispiele und Erläuterungen befinden sich online unter [MIC09a].

Abschließend wird der Begriff der „externen Iteration“ noch einmal verdeutlicht: Die `foreach`-Schleife innerhalb der o.a. Erweiterungsmethode `Where` stellt eine innere Iteration dar. Unter der externen Iteration versteht man die Schleife, welche über die von `Where`, bzw. anderen Erweiterungsmethoden, zurückgegebene `Query` iteriert.

3.3 Standardabfrageoperatoren

Neben dem bereits erwähnten Operator `Where` stellt LINQ weitere Abfrageoperatoren zur Verfügung. Sie sind im Namensraum `System.Linq` hinterlegt und als Erweiterungsmethoden von `IEnumerable` definiert.[11] Die Operatoren können daher auf sämtliche Auflistungen, die auf `IEnumerable` oder der generischen Variante basieren, angewandt werden. Eine Gesamtübersicht der Operatoren findet sich in folgender Tabelle:

Art	Operatoren
Aggregation	`Aggregate, Average, Count, LongCount, Min, Max, Sum`
Element	`DefaultIfEmpty, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault`
Filtern	`Where, OfType`
Generierung	`DefaultIfEmpty, Empty, Range, Repeat`
Gleichheit	`SequenceEqual`
Gruppierung	`GroupBy, ToLookup`
Konvertierung	`AsEnumerable, AsQueryable, Cast, OfType, ToArray, ToDictionary, ToList, ToLookup`
Mengen	`Distinct, Except, Intersect, Union`
Partitionieren	`Skip, SkipWhile, Take, TakeWhile`
Projektion	`Select, SelectMany`
Quantifizierung	`All, Any, Contains`
Sortieren	`OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse`
Verkettung	`Concat`
Verknüpfung	`Join, GroupJoin`

Tabelle 3-1: LINQ-Standardabfrageoperatoren

In den nächsten Abschnitten werden einige ausgewählte Operatoren genauer erklärt. Die Auswahl ist nur beispielhaft und erklärt sich durch eine häufige Verwendung dieser Operatoren in der Fallstudie „TheaterMaxx“.[12]

[11] Weitergehende Informationen in [MIC07a].
[12] Quelltexte der Fallstudie finden sich im Anhang A.

3.4 SelectMany-Operator

Enthalten die Elemente der Quellauflistung Aufzählungen, welche wiederum Elemente enthalten, an denen ein Interesse besteht, erfordert die Abfrage der Unterelemente eine weitere Iteration.

Gegeben sei als Beispiel ein Theater, welches über mehrere Ensembles verfügt. Ein Ensemble ist eine Zusammenstellung von Mitgliedern, z.B. Schauspielern (Darsteller), Statisten, Helfern, Technikern etc. Der folgende Quellcode iteriert über alle Ensemblemitglieder und gibt deren Namen auf der Konsole aus:

```
1   foreach (var ensemble in Ensembles)
2       foreach (var mitglied in ensemble.Ensemblemitglieder)
3           Console.WriteLine(mitglied.ToString());
```

Quelltext 3-5: Iteration über alle Ensemblemitglieder

LINQ ermöglicht die Ausgabe der Mitgliedsnamen, in einer einzigen (externen) `foreach`-Schleife:

```
1   var query = from ensemble in Ensembles
2               from mitglied in ensemble. Ensemblemitglieder
3               select mitglied;

4   foreach (var mitglied in query)
5       Console.WriteLine(mitglied.ToString());
```

Quelltext 3-6: Iteration über alle Ensemblemitglieder (Abfragesyntax)

In der ersten Zeile wird die Auflistung `Ensembles` durchlaufen und jedes Ensemble in der gleichnamigen Bereichsvariable abgelegt.

Zeile 2 greift wiederum über diese Bereichsvariable auf die Auflistung der `Ensemblemitglieder` des jeweiligen Ensembles zu und erstellt eine weitere Bereichsvariable.

Die einzelnen Mitglieder werden in Zeile 3 projiziert. Es wird eine Auflistung vom Typ `IEnumerable<Ensemblemitglied>` zurückgegeben.

Die Abfragesyntax unterstützt den Operator `SelectMany` nicht und erfordert zwei `from`-Abschnitte. Das Resultat ist gleichwertig. Zur besseren Verdeutlichung des Unterschiedes zwischen `Select` und `SelectMany` wird zunächst wiederholt auf den Operator `Select` eingegangen:

Der `Select`-Operator dient dem Zweck, für jedes Element der Auflistung einen Wert zurückzugeben. Dieser Wert kann aus mehreren Teilwerten, z.B. Name und Ort, zusammengesetzt sein, wird jedoch zu einem neuen, anonymen Typ, und damit zu einem einzigen Wert, zusammengefasst. Die Abfrage enthält anschließend dieselbe Anzahl an Elementen wie die Quelle.

Enthalten die Elemente der Quellauflistung, wiederum eine (Unter-)Auflistung, werden die einzelnen Unterelemente durch `SelectMany` automatisch miteinander verkettet und ein einziges Gesamtergebnis zurückgegeben. In diesem Beispiel besteht das Gesamtergebnis aus der Auflistung der Ensemblemitglieder aller Ensembles.

Die Lösung mit der LINQ-Methodensyntax gestaltet sich wie folgt:

```
1   var query = Ensembles
2               .SelectMany(ensemble => ensemble.Ensemblemitglieder)

3   foreach (var mitglied in query)
4       Console.WriteLine(mitglied.ToString());
```

Quelltext 3-7: Iteration über alle Ensemblemitglieder per `SelectMany`

3.5 `ToList`-Operator

In der Fallstudie befinden sich viele Stellen, in denen eine Teilauflistung erzeugt wird. Beispielsweise soll eine Funktion eine Auflistung aller Personen aus einem beliebigen Ort liefern.

Eine klassische Vorgehensweise ist, dass zunächst eine neue Auflistung für das Rückgabeergebnis erstellt wird. Anschließend wird über alle Personen iteriert und der Ort mit dem Gewünschten verglichen. Nur Personen, auf die das Kriterium zutrifft, werden an die Rückgabeliste angehängt.

```
1  private List<Person> PersonenAusOrt(String Ort) {
2      List<Person> result = new List<Person>();

3      foreach (Person p in Personen)
4          if (p.Ort == Ort)
5              result.Add(p);
6      return result;
7  }
```

Quelltext 3-8: Rückgabe einer Teilauflistung von Personen (Iterativ)

Nachteilig bei dieser Lösung ist der unerwünschte „Boilerplate-Code“, da in allen ähnlichen Codefragmenten immer eine temporäre Liste erzeugt und die Elemente einzeln hinzugefügt werden müssen.

Die Möglichkeit einer Filterung mit LINQ wurde bereits im Abschnitt 3.1 vorgestellt. Die `query` muss lediglich in eine Liste konvertiert und kann dann direkt zurückgegeben werden.

```
1  private List<Person> PersonenAusOrt(String Ort) {
2      var query = from p in Personen
                   where p.Ort == Ort
                   select p;

3      return query.ToList();
4  }
```

Quelltext 3-9: Rückgabe einer Teilauflistung von Personen (Abfragesyntax)

Der Operator `ToList` besitzt kein Äquivalent in der Abfragsyntax, so dass in diesem Fall die Methodensyntax auf das `Query`-Objekt angewandt wird.

`ToList` wird im Übrigen nicht verzögert, sondern direkt ausgeführt. Die Umwandlung hat daher zur Folge, dass auch die Abfrage sofort ausgeführt wird. Soll die aufrufende Methode die verzögerte Ausführung nutzen können, muss der Rückgabetyp der Funktion als `IEnumerable<Person>` definiert und `query` direkt per `return` übergeben werden.

Wird die Liste direkt im `return`-Statement produziert, wird störender „Boilerplate-Code“ größtenteils vermieden, wie nachfolgendes Beispiel in der Methodensyntax zeigt:

```
1  private List<Person> PersonenAusOrt(String Ort) {
2      return Personen
               .Where(p => p.Ort == Ort)
               .ToList();
3  }
```

Quelltext 3-10: Rückgabe einer Teilauflistung von Personen (Methodensyntax)

Abschließend bleibt die genaue Funktionsweise von `ToList` zu klären. In der Erweiterungsmethode wird aus der `IEnumerable`-Auflistung `source` ein neues `List`-Objekt erzeugt und im Konstruktor übergeben, wie folgender Code zeigt:

```
   public static List<TSource> ToList<TSource>(
                                  this IEnumerable<TSource> source){
2      if (source == null) {
3          throw Error.ArgumentNull("source");
4      }
5      return new List<TSource>(source);
6  }
```

Quelltext 3-11: Implementierung von `ToList` auf `IEnumerable`

Die Implementierung von List wäre zu lang um sie abzudrucken, jedoch kann sie leicht mit der Software .NET-Reflector[13] eingesehen werden. Im Wesentlichen geschieht im Konstruktor von `List` Folgendes:

- es wird eine neue, generische `ICollection` erzeugt und versucht, die Quellauflistung mit dem `as`-Operator in diese zu konvertieren.
- gelingt dies, werden die Elemente mit `CopyTo` in die Liste übertragen.
- andernfalls wird ein leeres Array erzeugt und per `GetEnumerator()` und `MoveNext()` durch die Quellauflistung iteriert und die Elemente einzeln an die Liste angefügt.

[13] Informationen auf http://www.red-gate.com/products/reflector/. Die Software ist ebenfalls auf der beiliegenden CD zu finden, siehe Anhang B.

3.6 `Cast`-Operator

LINQ kann auf nicht generische Auflistungen keine Standardabfrageoperatoren außer `Cast<T>` verwenden. Der Operator konvertiert jedes Element der Auflistung in den angegebenen Datentyp. Im folgenden Beispiel soll eine Abfrage auf einer `ArrayList` ausgeführt werden. Diese Auflistungsart ist nicht generisch. Erst durch `Cast` kann der Operator `Select` verwendet werden.

```
1   ArrayList testliste = new ArrayList();
2   testliste.Cast<String>()
3            .Select(s => s);
```

Quelltext 3-12: Operator `Cast`

Ein weiteres Einsatzgebiet ist die Transformation vererbter Objekte. Eine Auflistung mit `Posten`-Objekten soll in eine Auflistung mit Darstellern konvertiert werden. `Darsteller` erbt in der Fallstudie von `Posten`:

```
IEnumerable<Darsteller> darsteller = _posten.Cast<Darsteller>();
```

3.7 `Sum-` / `Count`-Operator

Häufige Verwendung findet in der Fallstudie der Aggregationsoperator `Sum`. Er bildet eine Summe über alle numerischen Elemente einer Auflistung, oder über eine numerische Eigenschaft der Elemente. Ein weiterer, oft genutzter, Operator ist `Count`. Er gibt die Anzahl der Elemente in einer Auflistung zurück.

`Count` wird ähnlich wie `ToList` verarbeitet. Kann die Quellauflistung mit `as` in eine `ICollection` konvertiert werden, wird die Eigenschaft `Count` direkt auf dieser aufgerufen. Andernfalls wird über die Auflistung iteriert und die Anzahl der Elemente gezählt. `Sum` iteriert grundsätzlich immer und addiert die einzelnen Werte.

3.8 Eigene Erweiterungsmethoden

Die Standardabfageoperatoren sind als Erweiterungsmethoden implementiert. Daher besteht die Möglichkeit LINQ durch eigene Methoden auf ähnliche Art und Weise zu erweitern. Im Folgenden wird ein Beispiel hierfür vorgestellt:

Betrachtet man folgenden Quellcode, in dem die Namen aller Personen über 30 Jahren auf der Konsole ausgegeben werden,

```
1  var query = Personen
2                .Where(p => p.Alter > 30)
3                .Select(p => p.Name);

5  foreach (String s in query)
6      Console.WriteLine(s);
```

Quelltext 3-13: Ausgabe der Namen aller Personen über 30 Jahren (Methodensyntax)

findet sich auch trotz der neuen Abfragetechnik noch immer eine klassische `foreach`-Schleife.

LINQ bietet von sich aus keine Möglichkeit an, einer Abfrage mitzuteilen, wie mit den projizierten Elementen weiter verfahren werden soll. Besonders bei der strukturierten Methodensyntax kann die zusätzliche Schleife störend wirken.

Dies lässt sich über einen eigenen `ForEach`-Operator erledigen. Selbsterstellte Erweiterungsmethoden müssen in einer statischen Klasse definiert werden und die Methode muss ebenfalls statisch sein:

```
1  static class Extensions
2  {
3      public static void ForEach<T>(this IEnumerable<T> source,
                                      Action<T> func){
4          foreach (var item in source)
5              func(item);
6      }
7  }
```

Quelltext 3-14: Erweiterungsmethode `ForEach`

Die (immer noch erforderliche) `foreach`-Schleife wird so in die Erweiterungsmethode ausgelagert. Es wird dadurch keine zusätzliche Funktionalität geschaffen, so dass man dieses Konstrukt als „Syntaktischen Zucker“[14] bezeichnen könnte – allerdings lagern auch die anderen LINQ-Operatoren den eigentlichen Code nur an andere Stellen aus.

Nach dieser Logik könnte man sämtliche LINQ-Konstrukte als syntaktischen Zucker auffassen, da die Funktionalität nicht erweitert wird. Einfachere Les- und Wartbarkeit ist jedoch positiv aufzufassen, so dass man die Erweiterungsmethoden durchaus als nützliches Werkzeug betrachten kann.

Diese Erweiterungsmethode kann auf die Abfrage, bzw. jede `IEnumerable`-Auflistung angewandt werden und gliedert sich in die Methodensyntax ein:

```
1  Personen
2  .Where(p => p.Alter > 30)
3  .Select(p => p.Name)
4  .ForEach(s => Console.WriteLine(s));
```

Quelltext 3-15: Ausgabe aller Personen über 30 Jahren (ForEach-Operator)

Durch anonyme Methoden lässt sich so auch mehrzeiliger Quellcode ausführen, ohne diesen in eine zusätzliche Methode auslagern zu müssen:

```
1  Personen
2  .Where(p => p.Alter > 30)
3  .ForEach(s => {
4      Console.WriteLine(p.Name);
5      Console.WriteLine(p.Ort);
6      p.DoSomething();
7  });
```

Quelltext 3-16: `ForEach`-Operator mit mehrzeiligem Anweisungsblock

[14] Der Begriff „Syntaktischer Zucker“ wurde von dem britischen Informatiker Peter J. Landin geprägt und bezeichnet Erweiterungen der Syntax, die lediglich der Vereinfachung dienen, jedoch nicht die Funktionalität einer Programmiersprache erhöhen.

3.9 Abfragesyntax vs. Methodensyntax

Beide Schreibweisen sind gleichwertig und können nach Belieben genutzt werden, auch innerhalb einer Methode. Die Abfragesyntax kann immer direkt in die Methodensyntax überführt werden.

Jedoch gibt es einige Operatoren für die in der Abfragesyntax kein Gegenstück existiert, wie z.B. `Sum` und `Count`. Diese müssen dann entweder nachträglich auf die Query angewandt werden oder die Abfrage wird direkt in der Methodensyntax formuliert.

Aus Gründen der Einheitlichkeit wird daher im Folgenden ausschließlich die Methodensyntax genutzt.

4 Fallstudie: TheaterMaxx

Nachdem die Grundlagen von LINQ erläutert wurden, soll der praktische Nutzen anhand einer Fallstudie evaluiert werden. Insbesondere soll dabei geprüft werden,

- mit welchem Aufwand vorhandene Anwendungen von klassischer Iteration nach LINQ umgestellt werden können,
- wie sich die Performance und die Les- und Wartbarkeit des Projektes verändert,
- ob vorhandene Schnittstellen von der Umstellung beeinflusst werden,
- ob eine konsequente und ausschließliche Verwendung von LINQ zielführend ist,
- und ob sich hieraus Ableitungen für andere Projekte bilden lassen.

Grundlage der Fallstudie bildet die Software „TheaterMaxx“ der FernUniversität in Hagen, welche dort im Rahmen eines Programmierpraktikums entwickelt worden ist. Die Software wurde vom Verfasser dieser Arbeit von Java nach C# übersetzt.

TheaterMaxx dient der Organisation von Theaterinszenierungen – angefangen von der Verwaltung der Darsteller, der weiteren Mitarbeiter und der Rollen über den textuellen Aufbau der aufzuführenden Stücke bis hin zu einer rudimentären Terminplanung.

Bei der Transformation der dort verwendeten klassischen Iterationen in die LINQ-Methodensyntax wurde darauf geachtet, die Typen der verwendeten Auflistungen unverändert zu lassen. Sollten sich daraus Vor- oder Nachteile für die eine oder andere Programmiertechnik ergeben, wird in den entsprechenden Abschnitten explizit darauf hingewiesen.

Abschnitt 4.1 enthält die kommentierten Klassenmodelle der vier wichtigsten Namensräume der TheaterMaxx-Fallstudie. Die für den Vergleich verwendeten Metriken erläutert Abschnitt 4.2. Eine Gliederung der LINQ-Abfragen nach Verwendungsmustern befindet sich in den Unterabschnitten zu 4.3. Hier wird die iterative Vorgehensweise einer Lösung in LINQ gegenübergestellt und anhand der Metriken ausgewertet. Weitere Verwendungsmuster, die nicht Bestandteil von „TheaterMaxx“ sind, werden im Abschnitt 4.4 vorgestellt. Abschnitt 4.5 beschäftigt sich mit der Problematik der bidirektionalen Navigation zwischen Auflistungen und der Frage, ob und wie diese in LINQ unterstützt wird. Eine tabellarische Zusammenfassung aller Messergebnisse zeigt Abschnitt 4.6.

4.1 Klassenmodelle

Die Einstiegsklasse ist Theatermaxx. Über diese kann auf mehrere Stücke, Räume, Ensembles, Inszenierungen oder Personen zugegriffen werden. In den folgenden Unterabschnitten werden die Klassendiagramme der Fallstudie dargestellt und kurz erläutert. Alle Modelle unterliegen dem Namensraum `theatermaxx.modell`, der nachfolgend nicht erneut genannt wird.[15]

4.1.1 Klassenmodell `stamm`

In Abbildung 4-1 ist das Klassenmodell `stamm` dargestellt. Unter einer Inszenierung versteht man die individuelle, praktische Umsetzung eines (geschriebenen) Stückes. Dabei kann es natürlich mehrere Inszenierungen geben, die dasselbe Stück umsetzen. Einer Inszenierung sind Räume zugeordnet, in denen die Personen proben oder das Stück aufführen. Ensembles bestehen aus mehreren Ensemblemitgliedern, welche Personen sind.

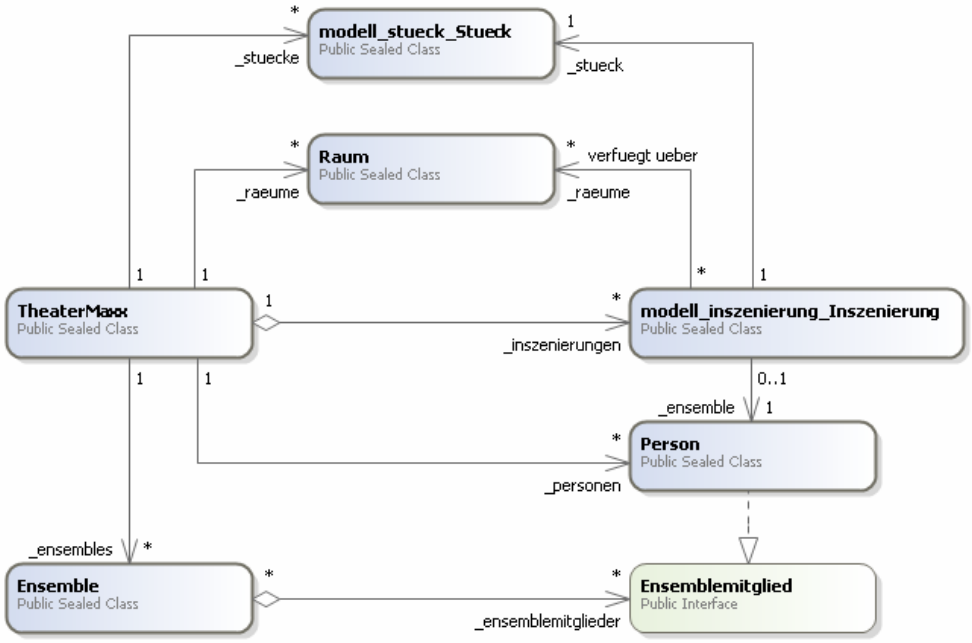

Abbildung 4-1: Klassenmodell `stamm`

4.1.2 Klassenmodell `inszenierung`

Abbildung 4-2 zeigt das Klassenmodel des Namensraumes `inszenierung`. Eine Inszenierung umfasst immer genau ein einzelnes Stück, wobei dieses in mehreren Inszenierungen umgesetzt sein kann. Ebenso bedient sich die Inszenierung eines Ensembles an Personen, welche die Ensemblemitglieder darstellen.

Die Inszenierung ist die praktische Umsetzung des Stückes anhand von Posten, welche durch Darsteller (z.B. Schauspieler, Statisten) und Nichtdarsteller (z.B. für Regie, Licht,

[15] Die Klassendiagramme wurden mit der Software NClass erstellt. Diese erlaubt es in der aktuellen Version leider nicht, Punkte in den Klassennamen zu verwenden. Aus diesem Grund wurden die Namensräume der Klassen, welche sich nicht in dem abgebildeten Namensraum befinden, mit einem Unterstrich dargestellt.

Kostüme etc.) besetzt werden. Lediglich Darstellerposten übernehmen eine Rolle (aus dem Stück). Die einzelnen Posten sind Ensemblemitgliedern zugeordnet.

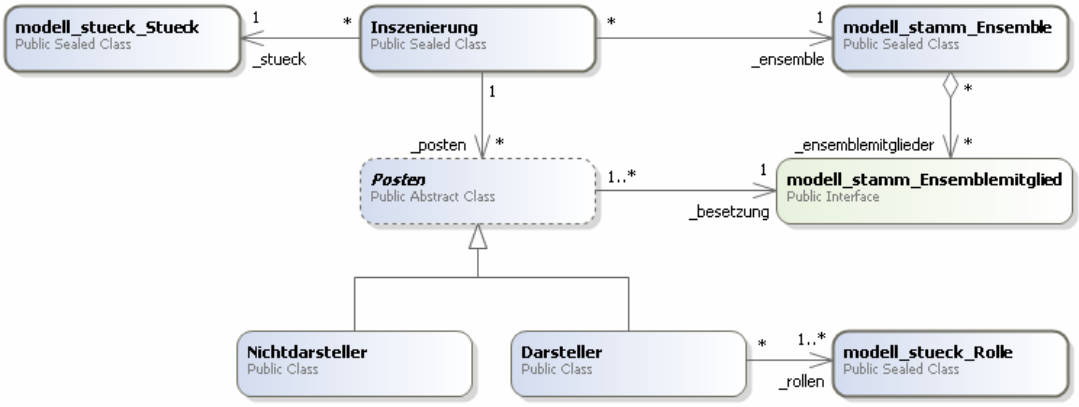

Abbildung 4-2: Klassenmodell `inszenierung`

4.1.3 Klassenmodell `stueck`

Wie Abbildung 4-3: Klassenmodell `stueck` veranschaulicht, besteht ein (geschriebenes) Stück besteht aus Akten. Diese unterteilen sich wiederum in Szenen, wobei jede Szene aus mehreren (textuellen) Absätzen besteht. Ein Absatz kann entweder die gesprochene Rede des Schauspielers oder eine Anweisung des Regisseurs sein. Beide wiederum bestehen aus Text. Die gesprochene Rede hält der Schauspieler im Rahmen seiner Rolle.

Im Gegensatz zum Akt, wird ein Stück vollständig aufgeführt und stellt damit eine Aufführungseinheit dar. Akte werden nicht einzeln vor Publikum aufgeführt, sondern lediglich geprobt. Beide stellen jedoch Spieleinheiten dar.

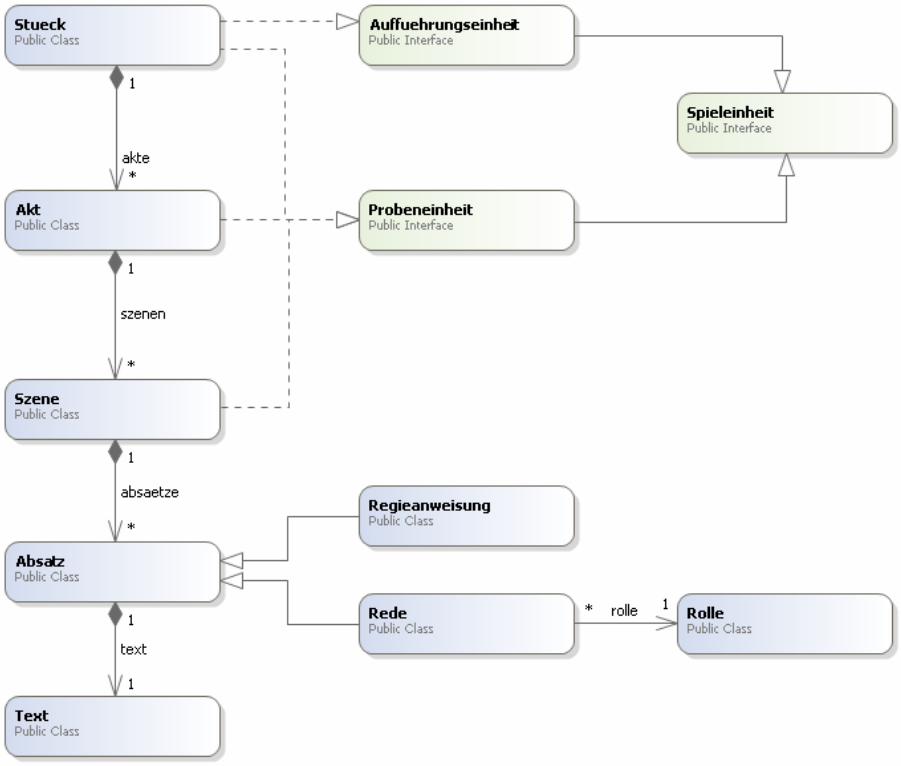

Abbildung 4-3: Klassenmodell `stueck`

4.1.4 Klassenmodell `termin`

Aus Abbildung 4-4: Klassenmodell `termin` geht hervor, dass für eine Inszenierung ein Theaterterminplan existiert, in dem Termine verwaltet werden. Zudem kann jeder Raum, der im Rahmen der Inszenierung genutzt wird, einen (anderen) Terminplan haben in dem z.B. die Nutzungszeiten eingetragen sind, damit andere Inszenierungen sich nicht terminlich überschneiden.

Für Personen bzw. Ensemblemitglieder können ebenfalls Termine eingeplant werden, z.B. bei Abwesenheit durch Urlaub, Krankheit etc.

Theatertermine wiederum unterteilen sich in Spieltermine für Spieleinheiten, Probentermine für die Probeneinheiten (welche Akte geprobt werden) und Aufführungstermine für die Aufführungseinheiten (d.h. es wird ein ganzes Stück aufgeführt).

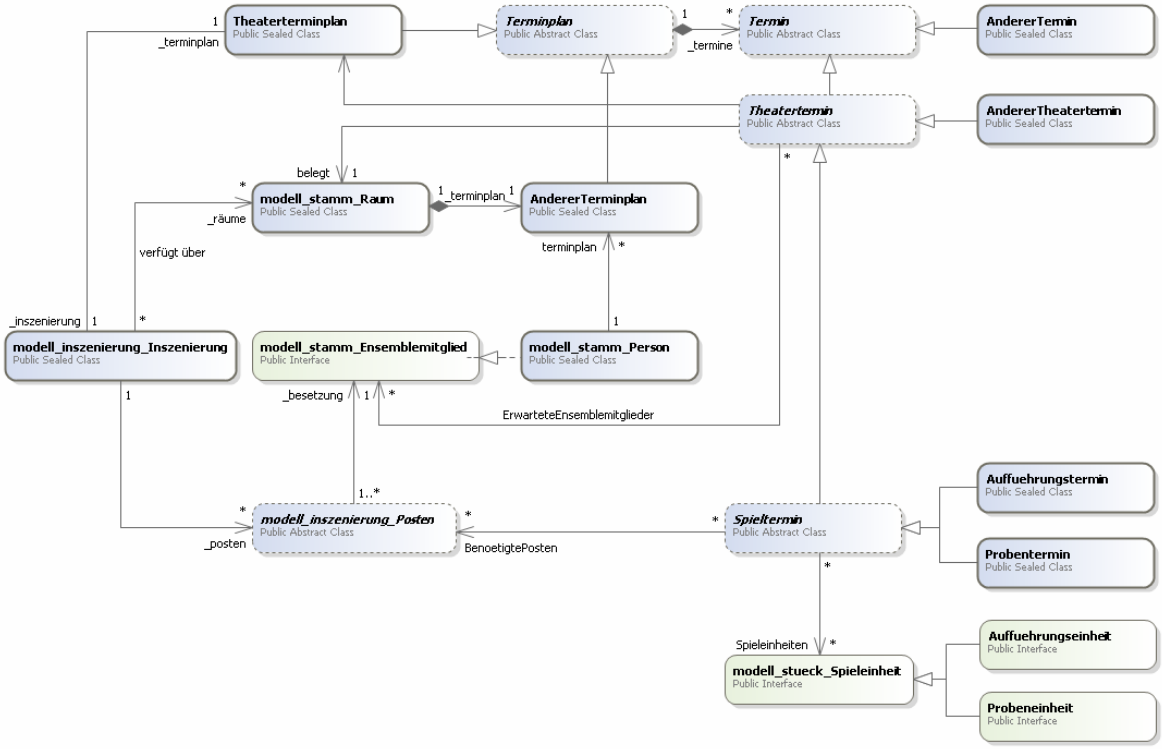

Abbildung 4-4: Klassenmodell `termin`

4.2 Metriken

Beide Syntaxen[16] werden mit verschiedenen Metriken gemessen um daraus Hinweise abzuleiten, welche Vorgehensweisen in den verschiedenen Fällen die geeignetsten sind. Die Metriken definieren sich wie folgt:

Iterationen — Die Anzahl der vollständigen Iterationen durch die Auflistung, um die Problemstellung zu lösen. Dies ist in erster Linie die Anzahl der `foreach`-Schleifen. Bei Erweiterungsmethoden sind diese allerdings nicht im Quellcode sichtbar. Diese Metrik dient der Erkennung solch versteckter Iterationen. Im Falle von Besonderheiten wird bei dem jeweiligen Verwendungsmuster darauf hingewiesen.

Wartbarkeitsindex — Dieser Wert wurde mit Hilfe des „Microsoft Visual Studio Team System 2008“ ermittelt. Laut Microsoft® ist dies ein relativer Indexwert zwischen 0 und 100. Je höher der Wert, um so wartbarer der Code.[17] Der Index setzt sich aus mehreren Faktoren zusammen:

- die Halstead-Metrik ist ein statisches, analysierendes Verfahren zur Messung der textuellen Komplexität von Software. Mit Hilfe eines Algorithmus, der auf der Anzahl der (verschiedenen) Operatoren und Operanden basiert, wird eine Kennzahl berechnet.[18]
- die zyklomatische Komplexität ist ein Index für die strukturelle Komplexität des Quellcodes. Es wird die Anzahl der einzelnen Codepfade, wie Schleifen, `Case`-Blöcke, `If`-Anweisungen etc. gezählt.
- die Anzahl der Codezeilen. Es werden die Zeilen im IL-Code[19], gezählt. Dabei werden alle Leerzeichen, geschweifte Klammern, Kommentare sowie die Deklaration von Namensräumen, Typen und Membern von der Berechnung ausgeschlossen.

Es ist anzumerken, das der Index leicht differiert, je nachdem ob das Projekt im Release- oder Debugmodus überprüft wird. Es scheint, als würden im Releasemodus weniger Codezeilen gezählt zu werden. Die Messungen wurden im Debugmodus durchgeführt, da in diesem auch gearbeitet wird.

Performance (Geschwindigkeit) — Für die Verwendungsmuster wurde die benötigte Zeit gemessen, um den Quellcode auszuführen – jeweils für die klassische Iteration und die LINQ-Methodensyntax bzw. für Alternativen oder optimierten Code[20]. Die Verwendungsmuster und Messklassen befinden sich im C#-Projekt, welches der Arbeit beiliegt, und ermöglichen den Vergleich auf anderen Testsystemen.

[16] Klassische Iteration und Methodensyntax.
[17] Quelle: [MIC07b].
[18] Siehe auch [HAL77].
[19] IL-Code wird vom Compiler aus dem Quelltext erzeugt. Dieser wird dann von einem JIT-Compiler (Just-In-Time) zur Laufzeit in nativen Code übersetzt und ausgeführt. Weitergehende Informationen finden sich online unter http://msdn.microsoft.com/de-de/library/bb979570.aspx.
[20] Testsystem: AMD Athlon 3700+ Single Core mit 2 GB DDR-Ram.

Es wurden je zwei Messungen durchgeführt: Einmal mit 46 Personen, die auf die Rollen verteilt sind, und einmal mit 10.046 Personen. Jedes Verwendungsmuster wurde in einer Schleife 1.000 Mal aufgerufen und die benötigte Gesamtzeit in Millisekunden gemessen.

Wird die Geschwindigkeit nicht durch die Anzahl an Personen beeinflusst, so wird lediglich der erste Messwert in der Fallstudie aufgeführt.

Zu beachten ist hier, dass es trotz der tausendfachen Iteration bei den Messungen reproduzierbar zu Schwankungen kommt. Zeitangaben, die verhältnismäßig nahe beisammen liegen, sollten daher vom Leser vorsichtig bewertet werden.

Usability (Anwendbarkeit) — Durch feste Metriken kann man reproduzierbare Eigenschaften des Codes bestimmen. Bei nicht zeitkritischen Anwendungen ist anzunehmen, dass auch die Vorlieben des jeweiligen Entwicklers eine Rolle dabei spielen, welche Technik eingesetzt wird. Da dies immer eine subjektive Bewertung ist, wird hier vom Verfasser eine Zuordnung zu den bekannten Schulnoten 1 bis 6 gewählt. Es folgt eine Kommentierung in *Kursivschrift*.

4.3 Verwendungsmuster in der Fallstudie

Viele der Iterationen bzw. LINQ-Abfragen lassen sich kategorisieren, z.B. nach Aggregation oder dem Ermitteln einer Teilmenge von Elementen. Die Aufteilung in solche Verwendungsmuster erleichtert den Vergleich beider Programmiertechniken, da der Quellcode für die Messungen auf das Wesentliche reduziert werden kann.

Jedes Verwendungsmuster, sowie die Anzahl der Vorkommen in der Fallstudie, werden in den folgenden Abschnitten zunächst kurz erläutert und der klassische iterative Code einer Lösung in LINQ tabellarisch gegenübergestellt.

Dieser Quellcode enthält nur die wichtigsten Zeilen, um den Verwendungszweck zu demonstrieren. Auf Anwendungsblöcke, die projektspezifischen Code enthalten, wurde weitestgehend verzichtet, um die verwendeten Metriken möglichst vergleichbar zu halten.

Die Tabellen basieren jedoch immer auf der TheaterMaxx-Fallstudie, d.h. es werden dieselben Klassen mit ihrer Bezeichnung und Hierarchie eingesetzt. Die Vorstellung der Verwendungsmuster ist wie folgt strukturiert:

- Zunächst wird das Verwendungsmuster kurz beschrieben.
- Falls angemessen, wird näher auf die iterative Implementierung und der Lösung in LINQ eingegangen.
- Darunter folgt eine Gegenüberstellung des Quellcodes. Die linke Spalte beinhaltet die iterative Vorgehensweise, die Rechte eine Transformation in die LINQ-Methodensyntax.
- Die Anzahl der Vorkommen innerhalb der Fallstudie wird aufgeführt. Hier ist zu beachten, dass nur die Muster gezählt wurden, die für die jeweilige Methode ausschlaggebend sind oder besonders häufig genutzt wurden. Beispielsweise wird die direkte Iteration überall eingesetzt, aber häufig in Kombination mit Filtern, so dass diese Stellen nur als gefilterte Iteration gewertet wurden.
- Anschließend werden die Messergebnisse tabellarisch dargestellt. Das jeweils bessere Ergebnis ist in **Fettschrift** dargestellt, bei Gleichstand ist keines der Ergebnisse hervorgehoben.
- Die subjektive Begründung der Metrik „Usability“ schließt den Abschnitt in *Kursivschrift* ab.

Die TheaterMaxx-Anwendung liegt dieser Arbeit auf CD bei und ist, durch Verwendung bedingter Kompilierung, zwischen beiden Varianten umschaltbar.[21]

[21] Siehe Anhang B.

4.3.1 Direkte Iteration

Bei diesem Verwendungsmuster wird direkt auf der Auflistung iteriert, um z.B. auf jedem Element eine Methode aufzurufen, Eigenschaften abzufragen oder es einer anderen Verwendung zuzuführen. Dargestellt wird dies durch den Aufruf der fiktiven Methode `DoSomething()`.

An dieser Stelle sei angemerkt, das im Folgenden bei der Methodensyntax grundsätzlich die im Abschnitt 3.8 vorgestellte Erweiterungsmethode `ForEach` verwendet wird, falls über eine Auflistung iteriert werden muss – wie in diesem Beispiel:

Iteration
```
foreach (var p in tm.Personen)
    p.DoSomething();
```

LINQ (Methodensyntax)
```
tm.Personen
   .ForEach(p => p.DoSomething());
```

Quelltext 4-1: Direkte Iteration

Anzahl der (ausschließlichen) Vorkommen in der Fallstudie: 26

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	1	81	**6 / 328**	**1**
LINQ	1	81	8 / 589	2

Tabelle 4-1: Messergebnisse – Direkte Iteration

Durch die Verwendung der oben beschriebenen `ForEach`-Erweiterungsmethode, ist die Methodensyntax vergleichbar mit der klassischen Iteration. Besondere Vorteile in der Verwendung einer LINQ-Abfrage sind nicht zu erkennen.

4.3.2 Gefilterte Iteration

Das Verwendungsmuster der gefilterten Iteration wird eingesetzt, wenn nur für eine Teilmenge von Elementen einer Auflistung Methoden aufgerufen, Eigenschaften abgefragt oder anderer Quellcode ausgeführt werden soll.

In der klassischen Variante wird zu diesem Zweck eine `If`-Anweisung eingefügt, in der eine oder mehrere Bedingungen überprüft werden. LINQ greift zu diesem Zweck auf die Erweiterungsmethode `Where` zurück, und übergibt ihr den Bedingungsblock.

Iteration
```
foreach (var p in tm.Personen)
    if (p.Alter > 30)
        p.DoSomething();
```

LINQ (Methodensyntax)
```
tm.Personen
   .Where(p => p.Alter > 30)
   .ForEach(p => p.DoSomething());
```

Quelltext 4-2: Gefilterte Iteration

Anzahl der Vorkommen in der Fallstudie: 14

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	1	**76**	**6 / 336**	1
LINQ	1	74	8 / 1407	1

Tabelle 4-2: Messergebnisse – Gefilterte Iteration

Interessant ist, dass der Schleifenaufruf und die Bedingungsabfrage die Positionen tauschen. Bei der Lesbarkeit ist kein Unterschied festzustellen und so kommt es zu einem Gleichstand in der Usability.

4.3.3 Abfrage des ersten Elementes

Es soll das erste Element in einer Auflistung ermittelt und in einer Objektvariablen referenziert werden – bei Verwendung innerhalb einer Funktion ist auch eine direkte Rückgabe per `return`-Anweisung denkbar.

Im ersten Beispiel handelt es sich bei der Personenauflistung um eine `List<Person>`. Diese Auflistungsart ermöglicht den Zugriff auf einzelne Elemente, wie bei einem Array, über die Indexnummer. Das erste Element kann direkt abgefragt werden. LINQ bietet zu diesem Zweck den Operator `First` an. Eine gleichnamige Funktion findet sich für die klassische Iteration in der Auflistung `LinkedList`, welche hier jedoch nicht weiter diskutiert wird.

Iteration	**LINQ (Methodensyntax)**
`Person person = tm.Personen[0];`	`Person person = tm.Personen.First();`

Quelltext 4-3: Abfrage des ersten Elementes (`List`)

Anders verhält es sich, wenn z.B. ein `HashSet` als Auflistung eingesetzt wird. In diesem Fall kann nicht über einen Index auf die Elemente zugegriffen werden, weil aufgrund der Hashfunktion keine feste Reihenfolge der Elemente definiert ist. Hier muss über einen Enumerator durch die Auflistung bzw. zum ersten Element navigiert werden.

`HashSet` implementiert jedoch `ICollection`, welche wiederum sich von `IEnumerable` ableitet. Die LINQ-Erweiterungsmethode kann daher auch in einem solchen Fall genutzt werden.

.

Iteration	**LINQ (Methodensyntax)**

`var enu = tm.Personen.GetEnumerator();`
`enu.MoveNext();`
`Person person = enu.Current;`
	`Person person = tm.Personen.First();`

Quelltext 4-4 : Abfrage des ersten Elementes (`HashSet`)

Anzahl der Vorkommen in der Fallstudie: 3

	Iterationen	Wartbarkeit	Performance	Usability
Iteration List	0	**90**	5 / 5	**1**
Iteration HashSet	0	77	5 / 5	3
LINQ	0	**90**	5 / 5	**1**

Tabelle 4-3: Messergebnisse – Abfrage des ersten Elementes

Die Abfragesprache profitiert von dem Konzept der Erweiterungsmethoden. Der Entwickler ist bei gleicher Syntax flexibler in der Auswahl der verwendbaren Auflistungen ohne Quellcode ändern zu müssen. Ist dies eine einfache oder verkettete Liste, endet der Vergleich der Usability pari.

4.3.4 Summen bilden und/oder Anzahl von Elementen ermitteln

Bilden einer Summe

Es soll eine Summe über eine numerische Eigenschaft aller Elemente in einer Auflistung berechnet werden. Dies kann iterativ am Kürzesten mit dem +=-Operator gelöst werden. LINQ erledigt diese Aufgabe intern über den mehrfach überladenen Operator `Sum`. Er kann ohne Parameter oder mit einem Delegaten aufgerufen werden, wie hier mit einem Lambda-Ausdruck, der die numerische Eigenschaft `Alter` projiziert.

Iteration	**LINQ (Methodensyntax)**

`long sum = 0;`
`foreach (var p in tm.Personen)`
`  sum += p.Alter;`
	`long sum = tm.Personen`
	`  .Sum(p => p.Alter);`

Quelltext 4-5: Bilden einer Summe über eine numerische Eigenschaft

Folgendes Beispiel demonstriert die Summenbildung über eine numerische Auflistung:

Iteration	**LINQ (Methodensyntax)**

`long sum = 0;`
`foreach (var i in Integers)`
`  sum += i;`
	`long sum = Integers.Sum();`

Quelltext 4-6: Bilden einer Summe über numerische Elemente

Anzahl der Vorkommen in der Fallstudie: 5

	Iterationen	Wartbarkeit	Performance	Usability
Iteration Person	1	76	**6 / 329**	1
LINQ Person	1	**80**	9 / 1283	1
Iteration Integer	1	78	**12**	2
LINQ Integer	1	**92**	124	**1**

Tabelle 4-4: Messergebnisse – Bilden einer Summe über numerische Elemente

Die klassische Iteration ist leicht verständlich, jedoch muss zunächst die Summenvariable mit einem Wert initialisiert werden, für den Fall, dass die Auflistung keine Elemente beinhaltet. Die LINQ-Abfrage gibt grundsätzlich immer einen Zahlenwert (Default 0) zurück. Zudem ist die Erweiterungsmethode interessant, wenn Summen über z.B. Integerauflistungen gebildet werden sollen.

Zählen der Elemente einer Auflistung

Die Anzahl aller Elemente in einer Auflistung soll gezählt werden. Im Falle der Verwendung einer Liste oder eines HashSets kann dies klassisch über die Eigenschaft `Count` geschehen. Bei konsequenter Verwendung der Abfragesprache wird auf die Erweiterungsmethode `Count` zugegriffen, die über `IEnumerable` zur Verfügung steht.

Iteration	**LINQ (Methodensyntax)**
`int counter = tm.Personen.Count;`	`int counter = tm.Personen.Count();`

Quelltext 4-7: Zählen der Elemente einer Auflistung

Anzahl der Vorkommen in der Fallstudie: 10

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	**0**	**91**	**4 / 4**	1
LINQ	0-1	90	8 / 5	1

Tabelle 4-5: Messergebnisse – Zählen der Elemente einer Auflistung

Syntaktisch ist kein Unterschied festzustellen, weshalb die Usability identisch bewertet wird. Dabei sollte nicht vergessen werden, dass die Erweiterungsmethode über alle Elemente iteriert, falls die Auflistung nicht `ICollection` implementiert, wie im Abschnitt 3.7 erklärt wurde. Bei der klassischen Iteration ist dies ebenfalls so, allerdings ist sich der Entwickler dessen bewusster, da es im Quellcode ersichtlich wird.

Zählen einer Teilmenge von Elementen

Nicht alle Elemente einer Auflistung werden gezählt, sondern nur diese, auf die eine spezifische Bedingung zutrifft. In beiden Varianten muss immer über alle Elemente iteriert werden, unabhängig vom Typ der Auflistung. Bei der LINQ-Abfrage ist dies dadurch begründet, dass `Where` zum einen eine Auflistung vom Typ `IEnumerable` zurückgibt und zum anderen verzögert ausgeführt wird – es bleibt für `Count` keine andere Wahl als die Elemente einzeln zu zählen.

Iteration

```
int counter = 0;
foreach (var p in tm.Personen)
    if (p.Alter > 30)
        counter++;
```

LINQ (Methodensyntax)

```
int counter = tm.Personen
              .Where(p =>p.Alter > 30)
              .Count();
```

Quelltext 4-8: Zählen einer Teilmenge von Elementen

Anzahl der Vorkommen in der Fallstudie: 1

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	1	72	**6 / 328**	1
LINQ	1	**79**	8 / 787	1

Tabelle 4-6: Messergebnisse – Zählen einer Teilmenge von Elementen

Die Abfragesprache bietet den Vorteil der Einsparung einer expliziten `foreach`-Schleife, da innerhalb der Erweiterungsmethode iteriert wird. Zudem muss die Zählvariable nicht initialisiert werden. Subjektiv und ohne Begründung wird die Usability dennoch gleich bewertet.

Durchschnittswert berechnen

Die Fallstudie beinhaltet eine Stelle, in der gleichzeitig gezählt und addiert wird – die Funktion `GueteDerBesetzung()` in der Klasse `Inszenierung`. Hier wird für jeden Darstellerposten ein Wert ermittelt, wie gut die Besetzung zu ihm passt, und im Anschluss diese Summe durch die Anzahl der Posten dividiert, um einen Durchschnittswert zu erhalten.

Für die Metriken wurde diese Funktion nur leicht abgewandelt. Die Vorgehensweise ist unverändert geblieben. Es soll das Durchschnittsalter aller Personen über 30 Jahren berechnet werden.

Die klassische Iteration bildet im Anweisungsblock der `If`-Anweisung die Summe und inkrementiert den Zähler. LINQ verfügt hingegen direkt über einen entsprechenden Aggregationsoperator `Average`, der intern identisch arbeitet.

Iteration

```
int counter = 0;
long sum = 0;

foreach (var p in tm.Personen)
    if (p.Alter > 30) {
        sum += p.Alter;
        counter++;
    }
float schnitt = sum / counter;
```

LINQ (Methodensyntax)

```
var query = tm.Personen
            .Where(p => p.Alter > 30)
            .Select(p => p.Alter);

float schnitt= (float)query.Average();
```

Quelltext 4-9: Durchschnittswert berechnen

Anzahl der Vorkommen in der Fallstudie: 1

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	**1**	66	**5 / 404**	2
LINQ	2	**70**	7 / 1049	**1**

Tabelle 4-7: Messergebnisse – Durchschnittswert berechnen

Die Quelltexte erscheinen ungefähr gleich (un-)übersichtlich. LINQ ist durch die geringe Anzahl an Einrückungen einfach zu lesen, und der `Operator` Average übernimmt sämtliche Berechnungen.

4.3.5 Geschachtelte Iteration

Unter geschachtelter Iteration versteht man die Iteration mit einer äußeren, sowie einer oder mehreren inneren Schleifen. In der Fallstudie finden sich verschiedene Unterarten dieses Verwendungsmusters:

- Iteration einer Auflistung mit Elementen, die Unterauflistungen enthalten
- geschachtelte Iteration über ein komplexes Dictionary
- geschachtelte Iteration über zwei Auflistungen

Elemente, die Unterauflistungen enthalten

Die Elemente der Auflistung beinhalten weitere Unterauflistungen, wie es in der Fallstudie z.B. in der Klasse `Akt` der Fall ist. Ein Stück besteht aus mehreren Akten. Jeder Akt verweist auf eine Menge von Szenen und es soll für jede einzelne Szene Code ausgeführt werden.

Ohne Verwendung von LINQ werden zwei `foreach`-Schleifen benötigt, um über beide Auflistungen zu iterieren und die einzelnen Unterelemente anzusprechen. Die Methodensyntax ermöglicht die Iteration mit nur einer Schleife durch Einsatz des in Abschnitt 3.4 beschriebenen Operators `SelectMany`. Dieser verkettet die einzelnen Szenen zu einer einzigen Auflistung, auf der dann die `ForEach`-Erweiterungsmethode arbeitet.

Iteration

```
foreach (var a in tm.Stueck.Akte)
    foreach (var s in a.Szenen)
        DoSomething(s);
```

LINQ (Methodensyntax)

```
tm.Stueck.Akte
    .SelectMany(a => a.Szenen)
    .ForEach(s => DoSomething(s));
```

Quelltext 4-10: Geschachtelte Iteration über Auflistungen mit Unterauflistungen

Anzahl der Vorkommen in der Fallstudie: 1 (Nur `SelectMany`)

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	2	74	**1311**	1
LINQ	2	**75**	1348	1

Tabelle 4-8: Messergebnisse – Geschachtelte Iteration über Auflist. mit Unterauflist.

Beide Möglichkeiten sind ausreichend kurz und prägnant formuliert. Eine identische Bewertung der Usability erscheint berechtigt.

Iteration komplexer Dictionaries

Eine Ableitung des o.a. Verwendungsmusters wird in der Klasse `Rollenprofil` der Fallstudie eingesetzt. Der Unterschied besteht darin, dass im Anweisungsblock, zusätzlich zu den Unterelementen, auch Code für die Hauptelemente ausgeführt werden soll – in diesem Fall werden die Namen der Akte sowie die Namen aller Szenen einem `StringBuilder` übergeben. Diese triviale Änderung wirkt sich auf LINQ mehr aus, als man zunächst annehmen möchte.

Aus Sicht der Abfragesprache ist das Beispiel ist allerdings in einem weiteren Punkt interessant: Bei der Quellauflistung handelt es sich um einen nicht generischen Typ. Als erster Operator muss deshalb `Cast` eingesetzt werden.[22]

Die Hauptauflistung `_akteUndSzenen` ist ein `HashTable`. Dieser Auflistungstyp implementiert `IDictionary` und besteht aus Schlüssel-Wert-Paaren, wobei es sich sowohl bei dem Schlüssel als auch bei dem Wert um ein Objekt vom Typ `Object` handelt. In diesem Fall bestehen der Schlüssel aus einem Akt und der Wert aus einem generischen `HashSet` vom Typ `Szene`, welches alle Szenen des Aktes beinhaltet.

Zunächst wird mit dem Operator `Cast` definiert, dass es sich um Elemente vom Typ `DictionaryEntry` handelt. `SelectMany` kann hier nicht eingesetzt werden, da zusätzlich der Name des Aktes abgefragt wird. `Select` und `SelectMany` schließen sich zwar nicht gegenseitig aus, jedoch müssen beide Operatoren abschließend eine Projektion zurückgeben.[23] Eine kombinierte Projektion der Elemente zweier Auflistungen ist mit den Standardabfrageoperatoren nicht möglich.

Mit dem `Select`-Operator wird zunächst ein neuer anonymer Typ definiert, um die Klassen des Schlüssel-Wert-Paares zu bestimmen. Anschließend folgen zwei Iterationen mit Hilfe der eigenen Erweiterungsmethode `ForEach`, ähnlich wie bei der klassischen Variante.

Iteration

```
var antwort = new StringBuilder();

foreach (Akt akt in _akteUndSzenen.Keys){
   antwort.Append(akt.Name);
   foreach (Szene szene in
        (HashSet<Szene>)_akteUndSzenen[akt])
        antwort.Append(szene.Name);
}
```

LINQ (Methodensyntax)

```
var antwort = new StringBuilder();

_akteUndSzenen.Cast<DictionaryEntry>()
.Select(a => new {
        akt = (Akt)a.Key,
        szenen = (HashSet<Szene>)a.Value})
.ForEach(entry => {
    antwort.Append(entry.akt.Name);
    entry.szenen.ForEach(szene =>
        antwort.Append(szene.Name));
});
```

Quelltext 4-11: Iteration komplexer Dictionaries

[22] Siehe auch Abschnitt 3.6.

[23] Bezüglich paralleler Verwendung von `Select` und `SelectMany` siehe Abschnitt 5.2.

Anzahl der Vorkommen in der Fallstudie: 1

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	2	**64**	**184**	**1**
LINQ	2	59	226	3

Tabelle 4-9: Messergebnisse – Iteration komplexer Dictionaries

Die Tatsache, dass Elemente der Hauptauflistung angesprochen werden, reicht aus, um die Methodensyntax deutlich komplexer zu gestalten. Der Quellcode ist schlechter verständlich und damit offensichtlich anfälliger für Fehler.

Geschaltete Iteration zweier Auflistungen

Dieses Verwendungsmuster kommt in der Klasse `Terminplan`, in der Funktion `DurchschnittMit`, zur Anwendung. Die Funktion prüft für einen übergebenen `Terminplan tp` und dem Objekt, auf dem sie definiert ist, ob sich Termine gegenseitig überschneiden und gibt einen neuen Terminplan zurück, der die überlappenden Termine enthält. Die geschachtelte Iteration findet über zwei getrennte Listen von Terminen statt.

Es muss angemerkt werden, dass die Funktion in der Fallstudie genutzt wird, um die freien Termine zweier Personen miteinander abzugleichen. Die Terminpläne enthalten deshalb nur die Termine, an der die Person noch verfügbar ist, mit dem Zweck, einen Terminplan zu generieren, an der beide Personen Zeit haben.

Die klassische Iteration basiert auf zwei `foreach`-Schleifen. Für jedes Element der Auflistung A wird jedes Element der Auflistung B durchlaufen und der Algorithmus ausgeführt. LINQ ruft die Erweiterungsmethode `ForEach` direkt auf der ersten Auflistung auf. Für jedes Element a wird die Auflistung B zunächst gefiltert das Ergebnis in einen freien Termin projiziert. Diese werden in einer weiteren `ForEach`-Methode an die Rückgabeauflistung angehängt.

Iteration

```
var termineA= new List<Termin>(_termine);
var termineB= new List<Termin>(tp._termine);
Terminplan e = new FreierTerminplan();
termineA.Sort();
termineB.Sort();

foreach (Termin a in termineA){
   foreach (Termin b in termineB){
       if (a.KollidiertMit(b))
          e.Eintragen(a.UeberlappungMit(b));
    }
}
```

LINQ (Methodensyntax)

```
var termineA=_termine.ToList();
var termineB= tp._termine.ToList();
Terminplan e = new FreierTerminplan();
termineA.Sort();
termineB.Sort();

termineA.ForEach(a => {
    termineB
    .Where(b => a.KollidiertMit(b))
    .Select(b => a.UeberlappungMit(b))
    .ForEach(t => e.Eintragen(t));
});
```

Quelltext 4-12: Geschachtelte Iteration zweier Auflistungen

Anzahl der Vorkommen in der Fallstudie: 1

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	2	60	**158**	1
LINQ	2	60	241	1

Tabelle 4-10: Messergebnisse – Geschachtelte Iteration zweier Auflistungen

Beide Techniken erhalten dieselbe Wertung. Die Vorgehensweise der klassischen Iteration ist vertraut und übersichtlich, die Methodensyntax besticht durch eine geringere Anzahl von Einrückungsebenen und trägt damit zur Übersichtlichkeit bei.

4.3.6 Vereinigungsmengen

Mehrere Auflistungen, mit Elementen gleichen Typs, werden in diesem Verwendungsmuster zu einer einzigen Auflistung, ohne Duplikate der Elemente, zusammengeführt. Die Klassen `Stueck`, `Akt` und `Szene` der Fallstudie geben jeweils eine Auflistung mit Rollen zurück, wobei jede Klasse wiederum eine gleichnamige Funktion auf der untergeordneten Klasse aufruft. In jeder dieser Funktionen wird die Vereinigungsmenge der Rollen gebildet.

Rollen werden in der Fallstudie in Auflistungen des Typs `List<Rolle>` verwaltet. Listen erlauben jedoch Duplikate, so dass dieser Typ ungeeignet ist, um Vereinigungsmengen zu generieren. Das `HashSet` hingegen, ermöglicht Zugriff auf die Methode `UnionWith`. Sie vereinigt die Menge, auf der sie aufgerufen wird, mit einer als Parameter übergebenen Menge. Abschließend wird eine neue Liste durch Übergabe des `HashSets` im Konstruktor erstellt und zurückgegeben.

In der Methodensyntax steht eine ähnliche Methode namens `Union` zur Verfügung. `Union` wird auf einer Auflistung aufgerufen und vereinigt diese mit einer weiteren Auflistung mit gleichen Elementen, die als Parameter übergeben wird. Dies kann innerhalb der Erweiterungsmethode `ForEach` geschehen. Als Problematisch bzw. unerwünscht hat sich in diesem Fall die verzögerte Ausführung von `Union` herausgestellt. Wird die Ausführung der Abfrage erst bei der `Return`-Anweisung durch `rollen.ToList()` erzwungen, wird eine Auflistung ohne Elemente zurückgegeben. Es ist zwingend erforderlich, direkt nach dem Operator `Union` die Abfrage ausführen zu lassen. Dies ist der Grund, weshalb die Auflistung `rollen` als `List` deklariert ist und nicht, wie in der klassischen Variante, als `HashSet`.

Iteration

```
var rollen = new HashSet<Rolle>();
foreach (Akt akte in _akte)
    rollen.UnionWith(akte.Rollen);
return new List<Rolle>(rollen);
```

LINQ (Methodensyntax)

```
var rollen = new List<Rolle>();
_akte.ForEach(akt => {
    rollen = rollen.Union(akt.Rollen)
                   .ToList();
});
return rollen;
```

Quelltext 4-13: Vereinigungsmengen

Anmerkung: Die Funktion hätte auch mit einem `HashSet` als Rückgabetyp implementiert werden können. Dann müsste LINQ stattdessen die Rückgabeauflistung erst konvertieren. Die gewählte Vorgehensweise soll keine Wertung darstellen.

Eine interessante Möglichkeit bietet in diesem Fall der Operator `SelectMany`. Es wurde bereits erklärt, dass dieser zum Verketten von Elementen mehrer Unterauflistungen dient, um geschachtelte Iterationen zu ersetzen.[24] Der Operator projiziert jedoch auch Duplikate. Durch Kombination mit dem Operator `Distinct`, der vielleicht aus der SQL-Syntax bekannt ist, werden diese Duplikate nachträglich herausgefiltert und der Operator lässt sich damit für Vereinigungsmengen zweckentfremden.

Intern gibt `Distinct` einen `DistinctIterator` zurück der wiederum `IEnumerator` implementiert. `Distinct` wird aus diesem Grund ebenfalls verzögert ausgeführt. Da in der Fallstudie eine Liste zurückgegeben werden muss und der Operator `ToList` die Ausführung veranlasst, ist dies kein Problem.

[24] Siehe Abschnitt 3.4.

Alternative : *Verwendung von* `SelectMany` *und* `Distinct`

```
1  return _akte
2         .SelectMany(akt => akt.Rollen)
3         .Distinct().ToList();
```

Quelltext 4-14: Vereinigungsmengen mit Distinct

Anzahl der Vorkommen in der Fallstudie: 2

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	2	70	556	2
LINQ Union	2	70	616	2
LINQ Distinct	2	**75**	**423**	**1**
LINQ Demeter[25]	*N/A*	*64*	*349*	**–**

Tabelle 4-11: Messergebnisse – Vereinigungsmengen

Unter Verwendung von `UnionWith` *und* `Union` *ist kein nennenswerter Unterschied zwischen Methodensyntax und klassischer Iteration zu erkennen. Aus Sicht der Usability ist die Kombination aus* `SelectMany` *und* `Distinct` *vorteilig zu sehen, da der Quellcode kürzer ist, keine temporäre Auflistung erzeugt werden muss und die Abfrage direkt an die* `Return`*-Anweisung gehängt werden kann.*

4.3.7 Vorzeitiger Iterationsabbruch

Vorzeitiges Verlassen einer Schleife ist ein in der Praxis häufig auftretendes Verwendungsmuster, beispielsweise um zu ermitteln, ob mindestens ein Element eine bestimmte Bedingung erfüllt. Angelehnt an die Funktion `IstFreiAn()`, welche sich mehrfach in der Fallstudie befindet, soll ermittelt werden, ob mindestens eine Person genau 38 Jahre alt ist.

Innerhalb einer klassischen Schleife ist es problemlos möglich, diese mit `break` abzubrechen oder über `return` die gesamte Funktion zu beenden. LINQ-Abfragen, können nicht abgebrochen werden, und die Auflistung würde immer vollständig durchlaufen werden. Die Methodensyntax bietet hingegen mit `Any` einen Operator an, der einen booleschen Wert zurückgibt, je nachdem, ob eine Sequenz Elemente enthält, für die eine Bedingung zutrifft. Alternativ kann `Any` auch ohne Parameter aufgerufen und zum Filtern der Operator `Where` eingesetzt werden.

Iteration

```
foreach (var person in tm.Personen)
    if (person.Alter == 38)
        return true;
return false;
```

LINQ (Methodensyntax)

```
return tm.Personen
       .Any(person => person.Alter == 38);
```

Quelltext 4-15: Vorzeitiger Iterationsabbruch

Anzahl der Vorkommen in der Fallstudie: 2

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	1	71	**6 / 321**	2
LINQ	1	**76**	9 / 586	**1**

Tabelle 4-12: Messergebnisse – Vorzeitiger Iterationsabbruch

[25] Messung einer LINQ-Abfrage die wichtige Gesetze der Programmierung verletzt. Ausführliche Diskussion in Abschnitt 5.2.

Durch das Merkmal der verzögerten Ausführung einer etwaig verwendeten `Where`-Methode bzw. der im Vergleich zur klassischen Iteration identischen Arbeitsweise des parametrisierten Operators `Any`, entstehen durch LINQ keinerlei Nachteile. Die Syntax ist deutlich kürzer und damit zu bevorzugen.

4.3.8 Aufeinanderfolgende Elemente vergleichen

Die Fallstudie verwendet dieses Muster in der Klasse `Terminplan`, um zu ermitteln, ob zwei Termine direkt aufeinander folgen.[26] Diese Termine sind in einem `HashSet` verwaltet und müssen erst in eine Liste konvertiert und anschließend sortiert werden. Ein `TreeSet`, wie in Java, ist im .NET-Framework nicht vorhanden. Die klassische Iteration bedient sich hier eines `Enumerator` um durch die Auflistung zu navigieren. Da immer zwei aufeinanderfolgende Elemente miteinander verglichen werden sollen, ist dies die einfachste Möglichkeit. Es wird zunächst das erste Element in der Variable `a` abgelegt. Anschließend iteriert man über den Rest der Auflistung, speichert das nächste Element in der Variablen `b` und führt den Vergleich durch. Die Finesse ist hier, die „Laufvariable“ zu überschreiben.[27]

Die Methodensyntax beginnt zunächst mit ähnlichen Aufrufen. Mit `FirstOrDefault` lässt sich das erste Element abrufen und `Skip(1)` gibt die Auflistung ab dem zweiten Element zurück.[28] Der Operator `Skip` zeichnet in diesem Fall dafür verantwortlich, dass `query` nicht als List<Termin> deklariert wurde, denn der Compiler würde einen Fehler mangels impliziter Konvertierungsmöglichkeit ausgeben.

Ab dieser Stelle kommt man mit LINQ nicht mehr weiter. Es wäre zwar möglich, in die selbsterstellte `ForEach`-Methode einen Anweisungsblock einzufügen, dieser lässt sich zum einen nicht abbrechen, zum anderen würden alle Folgeelemente immer mit dem ersten Element `a` verglichen. Es wurde keine andere Möglichkeit gefunden, als zu einer klassischen Schleife zu greifen oder eine weitere Erweiterungsmethode zu schreiben, welche dieselbe Funktionalität implementiert – angesichts der seltenen Aufgabenstellung erscheint dies nicht zweckdienlich.

Iteration

```
var temp = new List<Termin>(_termine);
temp.Sort();
IEnumerator termine = temp.GetEnumerator();
if (termine.MoveNext())       {
    Termin a = (Termin)termine.Current;
    while (termine.MoveNext()){
        Termin b = (Termin)termine.Current;
        if (a.AufeinanderfolgendMit(b))
            return true;
        a = b;
    }
}
return false;
```

LINQ (Methodensyntax)

```
IEnumerable<Termin> query =_termine
                           .OrderBy(t => t)
                           .ToList();
Termin a = query.FirstOrDefault();
if (a != null) {
    query = query.Skip(1);
    foreach (Termin b in query) {
        if (a.AufeinanderfolgendMit(b))
            return true;
        a = b;
    }
}
return false;
```

Quelltext 4-16: Aufeinanderfolgende Elemente vergleichen

Abweichend von der generellen Vorgehensweise, wurden in diesem Verwendungsmuster die Metriken nicht auf dem o.a. Quellcode durchgeführt. Stattdessen wurde mit Personen gearbeitet und ermittelt, ob je zwei aufeinander folgende Personen im selben

[26] Genauer gesagt, in der Funktion `EnthaeltAnschliessendeTermine`.
[27] Innerhalb einer foreach-Schleife dürfen Laufvariablen nicht überschrieben werden. Mit Enumeratoren ist dies jedoch möglich.
[28] Es handelt sich hier um einen nullbasierten Index.

Alter sind, unabhängig vom Bedeutungsgehalt in der Praxis. Grund für diese Änderung ist, das Verwendungsmuster mit einer großen Anzahl von Elementen testen zu können – mehrere tausend Termine erschienen unpraktisch. Der Quellcode für die Metrik befindet sich ebenfalls im Projekt der Fallstudie, im Namensraum `Metriken`.

Anzahl der Vorkommen in der Fallstudie: 1

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	**1**	**62**	**5 / 5**	**1**
LINQ	N	61	17 / 1759	4

Tabelle 4-13: Messergebnisse – Aufeinanderfolgende Elemente vergleichen

Dieser Sonderfall zeigt die Grenzen von LINQ auf. Der konsequente Verzicht auf klassische Vorgehensweisen ist nicht immer möglich. In diesem Fall sollte gänzlich auf LINQ verzichtet werden, da die verwendeten Operatoren in dieser Kombination eher unglücklich erscheinen und keinen Mehrwert bringen, weder funktional noch textuell.

4.3.9 Algorithmus „Stabile Heirat“

Zur Messung von sehr komplexen Aufgaben mit vielen verschiedenen Iterationen wurde der Algorithmus der „Stabilen Heirat“ – auch als Heirats- oder Sekretärinnenproblem bekannt – mit aufgenommen.

Der Algorithmus wurde im Rahmen des Programmierpraktikums an der FernUniversität in Hagen umgesetzt und versucht, eine optimale Besetzung der Darstellerposten mit Ensemblemitgliedern zu finden. Aufgerufen wird der Algorithmus über die Methode `StabilVerheiraten` in der Klasse `Inszenierung` und innerhalb der Klasse `StabileHeirat` weitergeführt. Quelltexte dieser Klassen befinden sich im Anhang und werden wegen ihres Umfanges an dieser Stelle nicht wiederholt. Weiterführende Informationen zu der genauen Vorgehensweise bei der Lösung des Heiratsproblems enthalten [WIR98] und [BRU04].

Bezüglich der Geschwindigkeitsmessung ist anzumerken, dass diese vermutlich noch weiter optimiert werden kann, zumindest bezogen auf die Methodensyntax. Eine Optimierung der Performance ist jedoch nicht das Hauptanliegen dieser Arbeit, so dass lediglich der Vollständigkeit halber eine Messung dieses Verwendungsmusters aufgenommen wurde. Zudem zeichnet sich durch die bisherigen Messungen ab, dass die Methodensyntax etwas langsamer ist. Durch kumulative Nutzung von Abfragen erscheint die verhältnismäßig hohe Differenz bei der Performancemessung nicht sehr überraschend.

Anzahl der Vorkommen in der Fallstudie: 1

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	N/A	43	**15579**	N/A
LINQ	N/A	**40**	27691	N/A

Tabelle 4-14: Messergebnisse – Algorithmus „Stabile Heirat“

4.4 Weitere Verwendungsmuster

Außerhalb der Fallstudie existieren zahlreiche weitere Verwendungsmuster für Erweiterungsmethoden der Language INtegrated Query. Während der Entwicklung der Testklassen für die Metriken, und der Frage nach Optimierungen der Performance, sind einige ausgesuchte Verwendungsmuster entstanden.[29] In den nachfolgenden Abschnitten werden diese erläutert und der iterativen Lösung gegenübergestellt.

4.4.1 Gruppierungen

Gegeben sei eine Liste von Personen, die aus verschiedenen Orten stammen. Die Aufgabenstellung ist, auf der Konsole den Namen und die Anzahl der unterschiedlichen Orte auszugeben. Duplikate dürfen nicht auftreten.

Ohne LINQ lässt sich das Problem mit einem `Dictionary<String, int>` lösen, wobei als Key jeweils der Name eines Ortes abgelegt wird und der Integer-Wert als Zählvariable dient. Die Auflistung wird durchlaufen und für jede `Person` geprüft, ob das `Dictionary` bereits einen `Ort` mit der Bezeichnung beinhaltet. Ist dies nicht der Fall, wird ein neuer Eintrag hinzugefügt und die Zählvariable mit dem Wert 1 initialisiert. Andernfalls wird nur der Zähler inkrementiert. Abschließend wird über das `Dictionary` iteriert und auf der Konsole ausgegeben.

Mit der Methodensyntax empfiehlt sich für die Hauptaufgabe, der Gruppierung der Personen nach Ortsnamen, der Operator `GroupBy`. Dieser ist mehrfach überladen und ermöglicht eine flexible Bestimmung der Eigenschaften, nach denen gruppiert werden soll. Der Rückgabewert ist hier vom Typ `IGrouping<String,Person>`. Im nachfolgenden Operator `Select` wird das `IGrouping`-Objekt in einen neuen anonymen Typen projiziert, der als Key den Namen des Ortes enthält und als zweite Eigenschaft die Anzahl der Elemente, nämlich die Personen aus dem jeweiligen Ort, mit `Count` berechnet.[30] Die Ausgabe auf der Konsole erfolgt, wie gewohnt, über eine `ForEach`-Methode.

Iteration

```
var result = new Dictionary<String, int>();

foreach (var p in tm.Personen) {
    if (!result.ContainsKey(p.Ort))
        result.Add(p.Ort, 1);
    else
        result[p.Ort]++;
}
foreach (var p in result)
    Console.WriteLine(
     p.Key + " " + p.Value.ToString());
```

LINQ (Methodensyntax)

```
tm.Personen
    .GroupBy(p => p.Ort)
    .Select(g => new { Name = g.Key,
                       Count = g.Count()})
    .ForEach(p => Console.WriteLine (
      p.Name + " " + p.Count.ToString()));
```

Quelltext 4-17: Gruppierungen

[29] Geschwindigkeitsoptimierungen sind nicht das Hauptanliegen dieser Arbeit. Bei auffälligen Unterschieden in der Performance wurde jedoch versucht, diese zu verbessern.
[30] Bei der Gelegenheit wurden Eigenschaften des anonymen Typs mit neuen Bezeichnungen versehen.

Wie die klassische Iteration an dieser Stelle bezüglich der Ausführungsgeschwindigkeit weiter optimiert werden kann, zeigt nachfolgender Quelltext:

```
1   var result = new Dictionary<String, int>();

2   foreach (var p in tm.Personen) {
3       int value;
4       if (result.TryGetValue(p.Ort, out value))
5           value++;
6       else
7           value = 1;
8       result[p.Ort] = value;
9   }

10  foreach (var s in result)
11      Console.WriteLine(s.Key + " " + s.Value.ToString());
```

Quelltext 4-18: Gruppierungen (Optimiert)

Einerseits wird `TryGetValue` schneller ausgeführt als `ContainsKey`, falls häufiger auf Schlüsselwerte zugegriffen wird, die noch nicht in der Auflistung vorhanden sind.[31] Andererseits benötigt `result[s.Ort]++` eigentlich zwei Nachschlageoperationen, da dieser Ausdruck auch als `result[s.Ort] = result[s.Ort] + 1` gewertet wird. Nach Abschluss dieser Optimierungen zeigt sich ein günstigeres Laufzeitverhalten, wie aus den Messergebnissen ersichtlich wird.

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	1	64	27 / 4047	2
Optimiert	1	62	**23 / 2872**	2
LINQ	1	**69**	30 / 2926	**1**

Tabelle 4-15: Messergebnisse – Gruppierungen

Mit LINQ lässt sich die Aufgabe elegant und einfach lösen. Der iterative Quellcode ist etwas schwerer zu lesen und der erste Versuch ist spürbar langsamer. Erst die optimierte Fassung erreicht eine vergleichbare Geschwindigkeit, wobei nicht auszuschließen ist, dass noch weitere Optimierungen möglich wären. Aus Usability-Sicht geht die Empfehlung an LINQ, da hier sichere Ergebnisse möglich sind.

4.4.2 Sortierte Gruppierungen

Die Konsolenausgabe in der vorherigen Problemstellung wurde nicht alphabetisch nach Orten sortiert, weshalb die Sortierung nun kurz angerissen wird. Die Veränderungen im Quelltext sind marginal:

- Links wurde der Auflistungstyp in ein `SortedDictionary` geändert,
- Rechts der selbsterklärende Operator `OrderBy` eingefügt.

Iteration

```
var result=new SortedDictionary<String,int>();
foreach (var p in tm.Personen){
    int value;
    if (result.TryGetValue(p.Ort, out value))
        value++;
    else
        value = 1;
    result[p.Ort] = value;
}
```

LINQ (Methodensyntax)

```
tm.Personen
    .GroupBy(p => p.Ort)
    .Select(g => new {
                    Name = g.Key,
                    Count = g.Count()})
    .OrderBy(g => g.Name)
    .ForEach(p =>
        Console.WriteLine (p.Name + " "
         + p.Count.ToString()));
```

[31] Quelle: http://msdn.microsoft.com/de-de/library/xfhwa508(VS.85).aspx.

```
foreach (var p in result)
    Console.WriteLine (p.Key + " "
     + p.Value.ToString());
```

Quelltext 4-19: Sortierte Gruppierungen

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	1	64	125 / 23409	2
Optimiert	1	62	88 / 15032	2
LINQ	1	**65**	**42 / 2944**	**1**

Tabelle 4-16: Messergebnisse – Sortierte Gruppierungen

Es ist festzustellen, dass der Performance-Unterschied deutlich ansteigt. Die Bewertung der Usability bleibt jedoch unverändert, denn im Quellcode finden sich kaum Unterschiede zu der unsortierten Variante.

4.4.3 Maximum ermitteln

Neben den bereits vorgestellten Aggregationsoperatoren `Sum` und `Count` existieren Weitere, von denen stellvertretend `Max` vorgestellt wird. Wie aus der Bezeichnung hervorgeht, wird der Operator eingesetzt, um das Maximum in einer Auflistung zu ermitteln.

Aus einer Liste von Personen verschiedenen Alters soll das höchste Alter berechnet werden. Dazu wird eine Hilfsvariable benötigt, welche temporär das jeweilige Maximum aufnimmt. Für jedes Element wird geprüft, ob der Wert höher ist als das derzeitige Maximum und dieses gegebenenfalls überschrieben. Nach der Iteration kann der Wert aus der Variable `maxValue` ausgelesen werden.

Der Operator `Max` arbeitet intern nahezu identisch. Er iteriert über die Elemente der Auflistung, ruft die übergebene Transformationsfunktion `p => p.Alter` auf und gibt das Maximum direkt an die Variable zurück. Eine Initialisierung ist nicht erforderlich, der Defaultwert ist 0.

Iteration

```
long maxValue = 0;
foreach (var p in tm.Personen)
    if (p.Alter > maxValue)
        maxValue = p.Alter;
```

LINQ (Methodensyntax)

```
long maxValue = tm.Personen
                  .Max(p => p.Alter);
```

Quelltext 4-20: Maximum ermitteln

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	1	72	**6 / 340**	2
LINQ	1	**80**	9 / 855	**1**

Tabelle 4-17: Messergebnisse – Maximum ermitteln

Die deutlich kürzere Syntax der Language INtegrated Query ist übersichtlicher und gewinnt den Vergleich der Usability.

4.4.4 Element mit Maximum ermitteln

Soll hingegen nicht der maximale Wert, sondern das Element mit einem maximalen Wert ermittelt werden, ist die Aufgabe mit anderen Mitteln zu lösen. Von der Vorgehensweise ändert sich an der klassischen Variante nicht viel. Es wird eine temporäre Variable initialisiert und anschließend über der Auflistung iteriert. Die Bedingungsklausel ist so konstruiert, das mindestens eine Person innerhalb der Schleife als Älteste festgelegt wird.

Mit LINQ steht diesmal der Operator `Max` nicht zur Verfügung, da dieser nur Werte, und keine Objekte, als Rückgabewert akzeptiert. Stattdessen wird die Auflistung absteigend nach Alter sortiert und anschließend das erste Element als älteste Person bestimmt. `FirstOrDefault` ist eine Erweiterung des `First`-Operators und gibt `Null` zurück, falls kein Element gefunden wurde, d.h. die Auflistung leer ist.

Iteration

```
Person aeltester = null;

foreach (var person in tm.Personen)
    if ((aeltester == null) ||
(person.Alter > aeltester.Alter))
        aeltester = person;
```

LINQ (Methodensyntax)

```
var aeltester =
    tm.Personen
    .OrderByDescending(p => p.Alter)
    .FirstOrDefault();
```

Quelltext 4-21: Element mit Maximum ermitteln

Die Metriken am Ende dieses Abschnittes werden zeigen, dass die Methodensyntax langsamer ist, als die iterative Lösung. Es folgen daher Versuche, durch Umformung der Abfrage diesen Nachteil auszugleichen.

Im Hintergrund von `OrderByDescending` werden komplexe Arbeitsschritte ausgeführt. Durch den .NET-Reflektor wird ersichtlich, dass über eine Hierarchie von `OrderedEnumerable` und `EnumerableSorter` gearbeitet wird. Es wird nun eine Möglichkeit gesucht, auf diesen Operator zu verzichten.

Im vorherigen Verwendungsmuster wusste der Operator `Max` zu gefallen. Es wird nun vorerst das maximale Alter ermittelt. Anschließend werden, mit einer erneuten Abfrage auf der Personenauflistung, alle Personen gefiltert, die dasselbe Alter erreicht haben und das erste Element wird zurückgegeben.

```
1  var maxAlter = tm.Personen.Max(o => o.Alter);

2  var aeltester = tm.Personen
3          .Where(p => p.Alter == maxAlter)
4          .FirstOrDefault();
```

Quelltext 4-22: Element mit Maximum ermitteln (zwei Abfragen)

Die Annahme, das `OrderBy` für den Leistungseinbruch verantwortlich ist, wird bestätigt. LINQ lässt sich allerdings noch weiter optimieren, wie folgendes, aus [MAR08] adaptiertes Beispiel zeigt:

```
var aeltester = tm.Personen.MaxElement(p => p.Alter);
```

`MaxElement` ist eine selbsterstellte Erweiterungsmethode, wie sie bisher auch für `ForEach` genutzt wurde. Die Arbeitsweise ähnelt dem ursprünglichen iterativen

Lösungsansatz sehr. Es lässt sich nun argumentieren, dass dies keine reine LINQ-Lösung sei, da lediglich Quellcode in eine weitere Methode ausgelagert wird.

Erweiterungsmethoden sind aber maßgeblicher Bestandteil von LINQ, so dass die Verlagerung legitim erscheint. `MaxElement` lässt sich, nach einmaligem Arbeitsaufwand, flexibel auf alle LINQ-konformen Auflistungen anwenden.

```
1  public static TElement MaxElement<TElement, TData>(
                                  this IEnumerable<TElement> source,
                                  Func<TElement, TData> selector)
                                  where TData : IComparable<TData> {

2      if (source == null) throw new ArgumentNullException("source");
3      if (selector == null) throw new ArgumentNullException("selector");

4      Boolean firstElement = true;
5      TElement result = default(TElement);
6      TData maxValue = default(TData);
7      foreach (TElement element in source) {
8          var candidate = selector(element);
9          if (firstElement || (candidate.CompareTo(maxValue) > 0)) {
10           firstElement = false;
11           maxValue = candidate;
12           result = element;
13         }
14     }
15     return result;
16 }
```

Quelltext 4-23: Element mit Maximum ermitteln (MaxElement)

	Iterationen	Wartbarkeit	Performance	Usability
Iteration	**1**	72	**6 / 348**	2
LINQ OrderBy	**1**	79	19 / 4433	2
LINQ 2 Abfragen	3	65	11 / 1196	4
MaxElement	**1**	**80** (57)	9 / 586	**1**

Tabelle 4-18: Messergebnisse – Element mit Maximum ermitteln

Kürzer und übersichtlicher als mit einer Erweiterungsmethode `MaxElement`, welche nur einmal definiert werden muss, lässt sich die Aufgabe nicht lösen. Der Wert in Klammern ist die Wartbarkeit der Erweiterungsmethode selbst. Zwei Abfragen zu verwenden ist zwar schneller, erscheint aber zu konstruiert.

4.5 Bidirektionale Navigation

Im Laufe der Fallstudie ist aufgefallen, dass an keiner Stelle bidirektional zwischen verschiedenen Objekten oder Auflistungen navigiert wird. Andere Projekte können diese Anforderung durchaus stellen. Ob LINQ die bidirektionale Navigation unterstützt und mit welchen Mitteln man diese allgemein verfügbar machen könnte, wird an einer konstruierten Aufgabenstellung besprochen.

In der Problemstellung möchte ein Darsteller wissen, zu welchen Probenterminen er erscheinen muss. Es sei bekannt, welche Akte an welchen Tagen geprobt werden. Um pünktlich zu erscheinen muss der Darsteller herausfinden in welchen Akten er auftritt.

Die Klassenstruktur im Unterabschnitt 4.1.3 sieht lediglich eine Navigation vom Stück über die Akte, Szenen und Absätze, zu der dem jeweiligen Absatz zugeordneten Rolle vor. Die umgekehrte Navigation von einer Rolle über die zugeordneten Absätze bis hin zu den Akten ist in der Fallstudie nicht vorgesehen.

Dies kann programmiertechnisch durch entsprechende Verweise realisiert werden. Jede Klasse würde eine Referenz auf die jeweils übergeordnete Klasse erhalten. Beim Hinzufügen z.B. eines neuen Absatzes zu einer Szene wird dann ein entsprechender Objektverweis gesetzt. Ebenso muss dieser beim Löschen eines Objektes aus einer Auflistung wieder entfernt werden. Es gilt aber zu prüfen, ob mit LINQ dieser Aufwand entfallen kann.

LINQ verwendet ausschließlich Erweiterungsmethoden auf IEnumerable, d.h. auf bestehenden Auflistungen. Diese stellen jedoch keine Möglichkeiten der bidirektionalen Navigation bereit. Aus diesem Grund können auch keine Abfragen definiert werden, um dieses Ziel zu erreichen, ebenso, wie es bei der klassischen Iteration der Fall ist. Ohne die entsprechenden (Navigations-)Strukturen bleibt dies verwehrt.

Zu Beginn dieser Arbeit, im Abschnitt 2.1, wurden mehrere LINQ-Implementierungen angesprochen, unter anderem LINQ to ADO.NET welches sich u.a. aus LINQ to SQL und LINQ to Entities zusammensetzt. Ein Blick über die in dieser Arbeit gesetzte Grenze, lediglich LINQ to Objects zu betrachten, enthüllt, dass in den Teilimplementierungen sehr wohl bidirektionale Navigationen unterstützt werden. Es bleibt festzustellen, wie diese implementiert sind und ob sich dies auch für LINQ to Objects nutzen lässt.

LINQ to SQL
Innerhalb einer LINQ-to-SQL-Klasse wurden die drei rudimentären Testklassen `Szene`, `Absatz` und `Rolle` erstellt. Der dahinter liegende Quellcode wird vollautomatisch erzeugt durch die Klasse erzeugt. Ein Blick in den Code zeigt, das nicht nur für die `Szene` ein `EntitySet<Absatz> Absaetze` erstellt wurde, sondern auch in der Klasse `Absatz` eine Relation `Szene Szene`. Zudem entstand Code, der sich um die Zuordnung des Parent-Objektes und dem Entfernen der Zuordnung beim Löschen kümmert.

Dadurch ist die bidirektionale Navigation möglich. Diese Tatsachen sind jedoch hinlänglich bekannt. Mit einer solchen selbsterstellten Implementierung könnten sowohl LINQ to Objects als auch die klassische Iteration die Problemstellung lösen.

LINQ to Entities

Interessanter sind die Ansätze, die LINQ to Entities verfolgt. Auch hier erhält die Klasse `Szene` automatisch eine `EntityCollection<Absatz>` `Absaetze` und `Absatz` wiederum ein Verweis auf das Parent-Objekt `Szene`. Verblüffend ist aber, dass kaum Quellcode für die Verwaltung vorhanden ist. Stattdessen stellt man fest, dass bei der Verwaltung der Relationen ein `Relationshipmanager`[32]-Objekt mit den Methoden

```
InitializeRelatedCollection<Absatz>("SzeneAbsatz", "Absatz", value);
bzw. GetRelatedReference<Szene>("SzeneAbsatz", "Szene").Value;
```

angesprochen wird. Offensichtlich werden die Relationen zwischen Objekten und Auflistungen an zentraler Stelle verwaltet. Leider ist auch dies nicht die gewünschte Lösung, denn der Relationship-Manager kann nur Objekte im „Entity Data Model" verwalten, und ist somit keine Lösung für LINQ to Objects bzw. der klassischen Iteration.

Relationship-Manager Entwurfsmuster

Bei der Suche nach einem adäquaten Ersatz für den Microsoft® Relationship-Manager fand sich ein vielversprechendes Projekt des Softwareentwicklers Andy Bulka[33] Auf seinem Webblog[34] stellt er einen Relationship-Manager vor, der nicht an Datenbankentitäten gebunden ist. Der Vollständigkeit halber wird auch dieser auf seine Verwendungsmöglichkeit zur Lösung der Problemstellung geprüft:

Gegeben sei folgender Quellcodeauszug:[35]

```
1   Rolle rolle1 = new Rolle("1. HEXE");
2   Rolle rolle2 = new Rolle("2. HEXE");
3   Rolle rolle3 = new Rolle("3. HEXE");
4   Absatz absatz;
5   Szene szene;
6   szene = new Szene("Erste Szene");
7   absatz = new Absatz("Wenn kommen wir drey uns wieder entgegen");
8   absatz.Rolle = rolle1;
9   szene.AddAbsatz(absatz);
    .
    .
    .
```

Quelltext 4-24: Testquellcode Relationship-Manager (AndyPatterns)

Es wurden versuchsweise zwei Szenen mit mehreren Absätzen erstellt, so dass der Rolle der „1. Hexe" insgesamt vier Absätze in zwei Szenen zugeordnet wurden.

Die Klassen selbst enthalten nur die Eigenschaften Name bzw. Text. Die Einbindung des Relationship-Managers beschränkt sich auf wenige Zeilen, die in den Klassen selbst umgesetzt wurden. `Szene` enthält z.B. eine Methode `AddAbsatz(Absatz absatz)` mit der Codezeile

```
rm.AddRelationship(this, absatz, "szeneToAbsatz");
```

[32] Siehe http://msdn.microsoft.com/de-de/library/system.data.objects.dataclasses.relationshipmanager.aspx
[33] Andy Bulka ist leitender Softwareentwickler bei „Magian Design Studio", Melbourne.
[34] http://www.andypatterns.com/index.php/design_patterns/relationship_manager/rm_for_net_csharp/
[35] Der vollständige Quellcode liegt der Arbeit auf CD bei.

und die Klasse `Absatz` eine Funktion `GetSzene()` mit

```
return (Szene)rm.FindObjectPointingToMe(this, "szeneToAbsatz");
```

Zur Ausgabe der Texte aller Absätze für `rolle1`, reicht folgende Zeile aus:

```
1   rolle1.GetAbsaetze().ForEach(abs =>  Console.WriteLine(abs.Text));
```

Quelltext 4-25: Ausgabe der Absätze (AndyPatterns)

Die Ausgabe der Namen beider Szenen wird wie folgt realisiert:

```
1   rolle1.GetAbsaetze()
2         .Select(abs => abs.GetSzene())
3         .Distinct()
4         .ForEach(s => Console.WriteLine(s.Name));
```

Quelltext 4-26: Ausgabe der Szenen (AndyPatterns)

Mit vollständiger Ausprogrammierung der hier fehlenden Klassen würde sich so die Problemstellung der bidirektionalen Navigation lösen lassen.

Die Language INtegrated Query setzt durch ihre Erweiterungsmethoden lediglich auf vorhandenen Auflistungen über die `IEnumerable`-Schnittstelle auf. Es wurde kein neuer Auflistungstyp, wie z.B. `BidirectionalCollection` eingeführt. Eine solche müsste in der Lage sein, die Problematik für alle verfügbaren Implementierungen von LINQ einheitlich zu lösen. Die Auflistung könnte sich vielleicht der Techniken des Relationship-Managers bedienen und diese Methoden, nach außen unsichtbar, kapseln. Zudem müssten die Kindobjekte immer eine direkte Zugriffsmöglichkeit auf ihr Elternobjekt erhalten. Änderungen an den Klassenstrukturen sollten allerdings sich möglichst wenig auf den bestehenden Quellcode auswirken. Die zusätzliche Unterstützung generischer Datentypen würde solche bidirektionalen Auflistungen abrunden.

4.6 Messergebnisse: Gesamtübersicht

Eine Zusammenfassung der Messergebnisse zeigt Tabelle 4-19. Die subjektive Metrik der Usability wurde nicht in diese Übersicht aufgenommen, da diese vom Betrachter abhängt. Standen mehrere Lösungsmöglichkeiten zur Verfügung, wurde die jeweils Beste für die klassische Iteration und die Methodensyntax gewählt.

Verwendungsmuster	Metriken	Klassische Iteration	LINQ-Methodensyntax
Direkte Iteration	Iterationen	1	1
	Wartbarkeit	81	81
	Performance	**6 / 328**	8 / 589
Gefilterte Iteration	Iterationen	1	1
	Wartbarkeit	76	**74**
	Performance	**6 / 336**	8 / 1407
Abfrage des ersten Elementes	Iterationen	0	0
	Wartbarkeit	90	90
	Performance	5 / 5	5 / 5
Bilden einer Summe	Iterationen	1	1
	Wartbarkeit	76	**80**
	Performance	**6 / 329**	9 / 1283
Zählen der Elemente	Iterationen	**0**	0-1
	Wartbarkeit	**91**	90
	Performance	**4 / 4**	8 / 5
Zählen einer Teilmenge	Iterationen	1	1
	Wartbarkeit	72	**79**
	Performance	**6 / 328**	8 / 787
Durchschnittswert	Iterationen	**1**	2
	Wartbarkeit	66	**70**
	Performance	**5 / 404**	7 / 1049
Geschachtelte Iteration	Iterationen	2	2
	Wartbarkeit	74	**75**
	Performance	**1311**	1348
Iteration komplexer Dictionaries	Iterationen	2	2
	Wartbarkeit	**64**	59
	Performance	**184**	226
Geschachtelte Iteration zweier Auflistungen	Iterationen	2	2
	Wartbarkeit	60	60
	Performance	**158**	241
Vereinigungsmengen	Iterationen	2	2
	Wartbarkeit	70	**75**
	Performance	556	**423**
Vorzeitiger Abbruch einer Iteration	Iterationen	1	1
	Wartbarkeit	71	**76**
	Performance	**6 / 321**	9 / 586

Aufeinanderfolgende Elemente	Iterationen	**1**	N
	Wartbarkeit	**62**	61
	Performance	**5 / 5**	17 / 1759
Gruppierungen	Iterationen	1	1
	Wartbarkeit	62	**69**
	Performance	**23 / 2872**	30 / 2926
Sortierte Gruppierungen	Iterationen	1	1
	Wartbarkeit	62	**65**
	Performance	88 / 15032	**42 / 2944**
Maximum	Iterationen	1	1
	Wartbarkeit	72	**80**
	Performance	**6 / 340**	9 / 855
Element mit Maximum	Iterationen	1	1
	Wartbarkeit	72	**80** (57)
	Performance	**6 / 348**	9 / 586
Algorithmus „Stabile Heirat“	Iterationen	N/A	N/A
	Wartbarkeit	**43**	40
	Performance	**15579**	27691

Tabelle 4-19: Messergebnisse: Gesamtübersicht

5 Ergebnisse der Fallstudie

Auf den folgenden Seiten werden die Ergebnisse der Fallstudie betrachtet.

Zunächst wirft Abschnitt 5.1 ein Blick auf die reinen Messergebnisse der benötigten Iterationen, der Ausführungsgeschwindigkeit und der Wartbarkeit aus Tabelle 4-19 und interpretiert diese. Abschnitt 5.2 diskutiert abschließend, welche Erfahrungen bei der Transformation in die Methodensyntax gemacht wurden. Darüber hinaus wird erläutert, welcher Art die Änderungen waren, die der Fallstudie widerfahren sind, bzw. welche ggfs. ausstehen könnten.

5.1 Zusammenfassung und Interpretation der Ergebnisse

Die Gesamtübersicht aller Messergebnisse zeigt, dass die klassische Iteration in den meisten Fällen performanter und die Methodensyntax, nach der von Microsoft® verwendeten Metrik, leichter wartbar ist. Die Anzahl der benötigten Iterationen hält sich, bis auf wenige Sonderfälle, die Waage:

- Gleichzeitige Abfrage zweier Elemente innerhalb einer Auflistung ist mit der Abfragesprache schwer zu realisieren. In Abschnitt 4.3.8 wurde versucht, dies mit dem Operator Skip zu lösen. Dieser hat jedoch zur Folge, dass über alle nicht übersprungenen Elemente iteriert wird.

- Der Unterschied in der Anzahl der Iterationen im Abschnitt 4.3.4, beim Zählen der Elemente einer Auflistung, muss differenziert betrachtet werden. LINQ verwendet intern dieselbe Vorgehensweise wie die klassische Iteration: Sofern die Auflistung eine `Count`-Eigenschaft anbietet, wird diese genutzt. Fehlt diese Eigenschaft, muss auch die klassische Variante über alle Elemente iterieren und zählen. Allerdings kann es mit der Abfragesyntax passieren, dass der Entwickler nicht bemerkt, dass eine für diesen Zweck langsamere Auflistung gewählt wurde, da der Operator `Count` immer zur Verfügung steht.

Die vorliegenden Messergebnisse lassen darauf schließen, dass die Verwendung von LINQ in zeitkritischen Anwendungen problematisch sein kann. Daraus sollte jedoch nicht pauschal gegen die Abfragesprache geurteilt werden, denn es wurde in mehreren Verwendungsmustern demonstriert, dass sowohl die klassische Iteration als auch die Methodensyntax in der Ausführungsgeschwindigkeit optimiert werden kann:

- In Unterabschnitt 4.3.6 wurde die Abfragegeschwindigkeit durch den Einsatz von `SelectMany` und `Distinct`, anstatt des nahe liegenden Operators `Union`, verbessert.

- Bei den Gruppierungen in den Unterabschnitten 4.4.1 und 4.4.2 bezogen sich die Optimierungen auf die iterative Variante. Bei unsortierten Gruppen konnte der vorerst erhebliche Rückstand auf LINQ sogar in einen knappen Vorteil umgewandelt werden.

- Für die Ermittlung des Elements mit einem Maximum wurden in Unterabschnitt 4.4.4 sogar zwei Verbesserungen erläutert und die Geschwindigkeit durch eine Erweiterungsmethode um den Faktor 7.5 erhöht.

Es ist anzunehmen, dass sich die Performance der anderen Verwendungsmuster ebenfalls weiter steigern lässt. Optimierungen sind allerdings häufig sehr projektspezifisch, da es teils auf „Kleinigkeiten" ankommt die nicht unbedingt für alle Projekte bzw. Problemstellungen geeignet sind. So würde eine Lösung für Gruppierungen ganz anders ausfallen, wenn z.B. mit einer fest definierten Liste aller erlaubten Orte gearbeitet wird, anstatt beliebige Orte zu erlauben.

Mit Einführung von .NET 4.0 wird LINQ durch die Erweiterung PLINQ[36] zudem die Nutzung von Mehrkernprozessoren unterstützen. Es bleibt daher abzuwarten, wie sich die neue Technik in der Praxis auf die Abarbeitungsgeschwindigkeit auswirkt. So schreibt schon das MSDN-Magazin in 2007:
„PLINQ ist ein Abfrageausführungsmodul, das alle LINQ-to-Objects- oder LINQ-to-XML-Abfragen akzeptiert und für die Ausführung automatisch mehrere Prozessoren oder Kerne verwendet, wenn diese zur Verfügung stehen. Die Änderung am Programmiermodell ist nur geringfügig, was bedeutet, dass Sie kein Parallelitätsgenie sein müssen, um es verwenden zu können."[37]

Anwendungen, die nicht zeitkritisch sind, mögen dem Aspekt der Wartbarkeit einen höheren Stellenwert einräumen als der Ausführungsgeschwindigkeit. In der Regel unterliegt Software wiederkehrenden Änderungsprozessen, verstärkt in der Entwicklungsphase, jedoch auch später im Rahmen der Wartung für den Einsatz auf neuen Betriebssystemen. Quellcode sollte daher gut strukturiert und lesbar sein, um den Zeitaufwand für Wartungsarbeiten und die damit verbunden Kosten bei anstehenden Änderungen zu senken.

In die Wartbarkeitsmetrik des „Microsoft Visual Studio Team System 2008" fließt u.a. das Halstead-Volumen mit ein und bietet daher ein gutes Maß für die textuelle Komplexität des Quellcodes. Der Vergleich zwischen klassischer Iteration mit der Language INtegrated Query verzeichnet ein Resultat von 4:11 für die Abfragesprache. Hierbei ist zu beachten, dass diese Werte von verschiedenen Faktoren abhängig sind:

- Unterabschnitt 4.3.3 zeigt mit der Ermittlung des ersten Elementes einer Auflistung, dass der Wartbarkeitsindex vom verwendeten Auflistungstyp abhängig ist – zumindest auf Seite der klassischen Iteration. Stellt die Auflistung keine Funktion bereit, um direkt auf das erste Element zu dereferenzieren, ist die Methodensyntax kürzer und einfacher zu verstehen.

- Aggregationsoperatoren in Unterabschnitt 4.3.4 vereinfachen z.B. die Summenbildung, das Zählen von Elementen oder stehen sogar für die Kombination beider, der Berechnung von Durchschnittswerten, zur Verfügung.

- Durch Zweckentfremdung der Operatoren `SelectMany` und `Distinct` wurden im Unterabschnitt 4.3.6 nicht nur die Performance verbessert, sondern ebenfalls die Lesbarkeit, wie der Wartbarkeitsindex aufzeigt.

[36] Parallel LINQ.
[37] Siehe http://msdn.microsoft.com/de-de/magazine/cc163329.aspx.

- Letztendlich können komplexere Aufgaben in eigene Erweiterungsmethoden ausgelagert werden. Der Aufruf wird kürzer und kann mehrfach verwendet werden. Beispielhaft sei die Methode `MaxElement` in Unterabschnitt 4.4.4 aufgeführt.

In vielen Fällen kann mit LINQ auf temporäre Hilfsstrukturen verzichtet werden, da alle Erweiterungsmethoden Ineinandergreifen und die Operatoren auf einzelnen Elementen, bedingt durch die verzögerte Ausführung, nacheinander ausgeführt werden. Auch dies trägt durch die Vermeidung von „Boilerplate-Code“ zur Erhöhung der Lesbarkeit bei.

5.2 Erfahrungen und Hinweise

Bei der Transformation der Fallstudie von einer klassischen Iteration zur Language INtegrated Query wurde großen Wert auf spätere Vergleichbarkeit des Quellcodes gelegt. Bestehende Auflistungsarten wurden daher unverändert gelassen.

An manchen Stellen bringt dies leichte Vor- oder Nachteile für die eine oder andere Technik mit sich. Beispielsweise in der Funktion `List<Rolle> Rollen` der Klasse `Szene`. Es könnte hier ebenso ein `HashSet<Rolle>` gewählt werden können. Da in der klassischen Iteration für die Funktion `UnionWith` ein `HashSet` vorausgesetzt wird, muss die temporäre Auflistung vor der Rückgabe in eine `List` umgewandelt werden. Umgekehrt kann LINQ durch die verzögerte Ausführung von `Union` nur mit einer temporären `List` arbeiten und müsste für die Rückgabe ein neues `HashSet` mit der `List` als Parameter im Konstruktor erzeugen. Da sämtliche Erweiterungsmethoden der Abfragesprache auf `IEnumerable` implementiert sind, erlaubt LINQ eine relativ flexible Wahl der Auflistung.

Von Überlegungen bezüglich der verzögerten Ausführung abgesehen, gestaltete sich die Transformation der klassischen Iteration zur LINQ-Methodensyntax ohne Probleme. Durch die verhältnismäßig hohe Ähnlichkeit der Syntaxen ist die Vorgehensweise fast immer dieselbe: Das Auflistungsobjekt wird ermittelt und die gewünschten Operatoren nacheinander angehängt. Im Vergleich zu `foreach`-Schleifen, auch in Verbindung mit `if`-Abfragen, tauschen die Zeilen lediglich die Plätze und die Operatoren (`if` gegen `where`). Begünstigt wird dies durch die vorgestellte Erweiterungsmethode `ForEach`.

Bei der Umstellung eines Projektes sollte auf das Merkmal der verzögerten Ausführung geachtet werden. Vorteilhaft ist, dass der Entwickler genau bestimmen kann, wann Abfragen ausgeführt werden.

So ist es z.B. möglich eine Abfrage an eine Funktion zurückzugeben, so dass diese erst ausgeführt wird, wenn in der aufrufenden Methode über das Abfrageergebnis iteriert wird. Dieses Verhalten kann auch unerwünscht sein, so dass die Ausführung aus zwei Gründen vorher erzwungen werden muss, weil

- sich vom Zeitpunkt der Rückgabe der Abfrage und der tatsächlichen Ausführung der Inhalt der Auflistung geändert haben könnte, und/oder

- im schlechtesten Fall eine leere Auflistung zurückgegeben wird.

Letzteres kann leicht bei Verwendung des Operators `Union` geschehen, wie im Unterabschnitt 4.3.6 beschrieben wurde.

Bezüglich theoretischer Änderungen an dem Projekt der Fallstudie wird abschließend die bereits angesprochene Funktion `List<Rolle> Rollen` diskutiert. Ein Aufruf der Funktion auf der Klasse `Stueck`, gibt eine Liste sämtlicher Rollen zurück, die innerhalb des Stückes vorkommen. Dazu werden gleichnamige Funktionen in den Klassen `Akt`, `Szene` und `Absatz` hierarchisch aufgerufen[38].

Der Umstieg auf eine Abfragesyntax könnte bei oberflächlicher Betrachtungsweise den Anschein erwecken, dass auf diesen geschachtelten Aufruf verzichtet werden kann, indem die Rollen über alle Relationen in einer einzigen Abfrage ermittelt werden. Dies ist auch mit mehreren geschachtelten `foreach`-Schleifen möglich, allerdings ist es wahrscheinlicher, dass dieser Gedanke eher mit der äußerst kompakten LINQ-Syntax verfolgt wird, als bei mehrfach eingerückten Schleifen.

```
1  public List<Rolle> Rollen {
2      get { return _akte
                    .SelectMany(a => a.Szenen)
                    .SelectMany(s => s.Absaetze)
                    .Where(abs => abs.Rolle != null)
                    .Select(abs => abs.Rolle)
                    .Distinct().ToList();
3      }
4  }
```

Quelltext 5-1: Abfrage über mehrere Relationen

Auch wenn der obige Quellcode schneller ausgeführt wird, als Abfragen in geschachtelten Funktionsaufrufen, so stellt keine Empfehlung dar, da er gegen zwei wichtige Prinzipien der Programmierung verstößt – dem Geheimnisprinzip[39] und dem Gesetz von Demeter[40]. Bezogen auf das Geheimnisprinzip lässt sich argumentieren, dass die einzelnen Unterauflistungen durch die Schnittstelle der jeweiligen Klassen nach außen veröffentlicht werden, so dass diese Abfrage überhaupt erst möglich wird. An der Verletzung des Gesetzes von Demeter ändert dies jedoch nichts.

Hinzu kommt, dass nicht immer der Quellcode bekannt ist, z.B. wenn die oben angegebenen Klassen in einer kompilierten DLL vorliegen. Ohne Einsicht in den Quellcode kann nicht bekannt sein, dass in der Klasse `Szenen` nur Rollen zurückgegeben werden, die nicht `null` sind. Ohne diese Information würde der Operator `Where` vermutlich nicht zum Einsatz kommen und die Abfrage immer eine Rolle zuviel zurückgeben.[41]

[38] Siehe Abschnitt 4.1.3.
[39] Das Geheimnisprinzip sagt aus, dass die Art und Weise, wie ein Modul seine Aufgaben erfüllt, im inneren des Moduls verborgen bleiben soll.
[40] Objekte sollen nur mit den Objekten ihrer unmittelbaren Umgebung kommunizieren, und nicht auf über diese auf weitere Objekte zugreifen. Siehe Webseite der Northeastern University of Boston: http://www.ccs.neu.edu/home/lieber/LoD.html
[41] Nämlich die leere Rolle, die der Regieanweisung zugeordnet wurde.

6 Schlussbetrachtungen und Ausblick

6.1 Zusammenfassung

Auch wenn die Interpretation der Messergebnisse aufgezeigt hat, dass an verschiedenen Stellen Spielraum zur Optimierung vorhanden ist, zeichnet sich eine Tendenz ab, welche Technik in welchen Fällen das geeignete Mittel sein könnte. In reinen Fragen der Ausführungsgeschwindigkeit ist der klassischen Iteration der Vorrang einzuräumen, hingegen empfiehlt sich für eine leichtere Les- und Wartbarkeit die Language INtegrated Query.

Während die Performance weiter optimiert werden kann, stellt sich die Frage, ob Entwickler den objektiven Wartbarkeitsindex als wertvolle Metrik betrachten. Denn es ist zu vermuten, dass Techniken danach eingesetzt werden, wie die Les- und Wartbarkeit subjektiv eingeschätzt wird.

In vielen Anwendungen werden beide Faktoren abgewägt und versucht, eine optimale Kombination beider zu erreichen. Im Rahmen der Fallstudie wurde bewusst auf eine solche Kombination verzichtet, um die Vor- und Nachteile beider Techniken für ausgewählte Verwendungsmuster darzustellen. Es spricht in der Praxis nichts dagegen, nicht zwischen der klassischen Iteration und der Abfragesprache zu wechseln bzw. Synergieeffekte zu nutzen. Diese Vorgehensweise ist sogar zu empfehlen, um ein Gleichgewicht aus Geschwindigkeit und Wartbarkeit zu erreichen.

6.2 Ausblick

LINQ wird in den nächsten Jahren sicherlich weiter entwickelt werden. Dies zeigen Projekte wie PLINQ, zur Unterstützung von Abfragen auf Mehrkernprozessoren, welches ein fester Bestandteil des .NET-Framework 4.0 sein wird. Auch für andere Programmiersprachen, ist die LINQ interessant – so liegt mit Quaere eine, von Microsoft® unabhängige, Implementierung der Abfragesyntax für Java vor.[42]

Bezugnehmend auf bidirektionale Navigationsmöglichkeiten verbleibt der Wunsch, dass zukünftige Versionen der Language INtegrated Query diese direkt auf den Auflistungen implementieren oder eine neue Auflistungsart einführen. Die eigene Implementierung der notwendigen Zeilen und des verwaltenden Quellcodes könnte dann entfallen und es müsste nicht auf externe Relationship-Manager zugegriffen werden. Mit Verfeinerung der Erweiterungsmethoden-Technik sollte auch dieses Ziel in erreichbarer Nähe sein.

[42] Siehe http://quaere.codehaus.org/Home

A Quelltext Fallstudie TheaterMaxx

A.1. Klasse: Inszenierung

public List<Darsteller> Darsteller - gibt eine Auflistung aller Darsteller(-Posten) der Inszenierung zurück.
HashSet<Posten> _posten = Auflistung aller Posten der Inszenierung

```
var darsteller = new List<Darsteller>();
foreach (Posten posten in _posten)
    if (posten is Darsteller)
        darsteller.Add((Darsteller)posten);
return darsteller;
```

```
return _posten
       .Where(p => p is Darsteller)
       .Cast<Darsteller>().ToList();
```

public void DarstellerpostenProRolleSchaffen() - erstellt für jede Rolle eines Stück einen Darstellerposten in Erstbesetzung (Rang 1)
Stueck _stueck = Das aktuelle Stück der Inszenierung

```
foreach (Rolle rolle in _stueck.Rollen)
    DarstellerpostenSchaffen(rolle, 1);
```

```
_stueck.Rollen.ForEach(rolle => DarstellerpostenSchaffen(rolle, 1));
```

public List<Posten> DarstellerVon(Rolle rolle) - gibt eine Auflistung aller Darsteller(-Posten) für eine gegebene Rolle zurück.
List<Darsteller> Darsteller = Auflistung aller Darsteller(-Posten) der Inszenierung (Property aus Inszenierung)

```
var antwort = new List<Posten>();
foreach (Darsteller darsteller in Darsteller)
    if (darsteller.Rollen.Contains(rolle))
        antwort.Add(darsteller);
return antwort;
```

```
return Darsteller
        .Where(d => d.Rollen.Contains(rolle))
        .Cast<Posten>().ToList();
```

public ICollection<Ensemblemitglied> Schauspieler - gibt eine Auslistung aller Mitglieder des Ensembles zurück, die Schauspieler sind.
Ensemble _ensemble = Das für diese Inszenierung eingesetzte Ensemble

```
var schauspieler = new List<Ensemblemitglied>();

foreach (Ensemblemitglied ensemblemitglied in _ensemble.Ensemblemitglieder)
    if (ensemblemitglied.IstSchauspieler())
        schauspieler.Add(ensemblemitglied);
return schauspieler;
```

```
return _ensemble.Ensemblemitglieder
       .Where(e => e.IstSchauspieler())
       .ToList();
```

public float GueteDerBesetzung() - gibt einen Wert zurück, der die Güte der Besetzung anzeigt, d.h. wie gut die Darsteller zu den zugeordneten Rollen passen.
List<Darsteller> Darsteller = Auflistung aller Darsteller(-Posten) der Inszenierung (Property aus Inszenierung)

```
float s = 0;
int n = 0;
float retVal = 0;
```

```
float s = 0;
int n = 0;
float retVal = 0;
```

```
foreach (Darsteller darsteller in Darsteller)
    if (darsteller.IstBesetzt()) {
        s += darsteller.Passt(_stueck);
        n += 1;
    }

retVal = (n == 0) ? 1 : s / n;
return retVal;
```

```
var query = Darsteller
            .Where(d => d.IstBesetzt())
            .Select(d => d.Passt(_stueck));

retVal = (n == 0) ? 1 : (float)query.Average();
return retVal;
```

private void Permutieren(List<Darsteller> restlDarsteller,List<Ensemblemitglied> restlEnsemblemitglieder, Pruefer pruefer) - probiert alle denkbaren Besetzungen von Darstellerposten durch Ensemblemitglieder durch, um so die beste Besetzung zu finden (Güte der Besetzung wird hier durch Unterfunktionen mit aufgerufen)
class Optimierer pruefer = Innere Hilfsklasse welche die eigentliche Prüfung der Güte vornimmt.

```
var tempDarst = new List<Darsteller>(restlDarsteller);
if (tempDarst.Count == 0)
    pruefer.Pruefen();
else {
    var darsteller = tempDarst[0];
    tempDarst.RemoveAt(0);

    foreach (Ensemblemitglied ensemblemitglied in restlEnsemblemitglieder)
        {
        darsteller.BesetzenMit(ensemblemitglied);
        var rest = new List<Ensemblemitglied>(restlEnsemblemitglieder);
        rest.Remove(ensemblemitglied);
        Permutieren(ref tempDarst, rest, pruefer);
    }
}
```

```
var tempDarst = new List<Darsteller>(restlDarsteller);
if (tempDarst.Count() == 0)
    pruefer.Pruefen();
else {
    var darsteller = tempDarst.First();
    tempDarst.Remove(darsteller);

    restlEnsemblemitglieder.ForEach(mitglied => {
        darsteller.BesetzenMit(mitglied);
        var rest = new List<Ensemblemitglied>(restlEnsemblemitglieder);
        rest.Remove(mitglied);
        Permutieren(ref tempDarst, rest, pruefer);
    });
 }
```

public float BesteBesetzungFinden() - gibt einen Wert zurück, der die maximal beste Besetzung von Darstellerposten mit Ensemblemitgliedern darstellen würde.
List<Darsteller> Darsteller = Auflistung aller Darsteller(-Posten) der Inszenierung (Property aus Inszenierung)

```
var ensemblemitglieder = new List<Ensemblemitglied>();
var darsteller = new List<Darsteller>();
foreach (Darsteller d in Darsteller){
    darsteller.Add(d);
    ensemblemitglieder.Add(d.Besetzung);
}
ensemblemitglieder.Reverse();

Optimierer optimum = new Optimierer(this);
Permutieren(ref darsteller, ensemblemitglieder, optimum);
return optimum.GetMaximalwert();
```

```
var ensemblemitglieder = new List<Ensemblemitglied>();
var darsteller = new List<Darsteller>();
Darsteller.ForEach(d => {
    darsteller.Add(d);
    ensemblemitglieder.Add(d.Besetzung);
});
ensemblemitglieder.Reverse();

Optimierer optimum = new Optimierer(this);
Permutieren(ref darsteller, ensemblemitglieder, optimum);
return optimum.GetMaximalwert();
```

private class Optimierer : public void Pruefen() - prüft, ob die Güte der Besetzung größer ist als das derzeitige Maximum, und würde in diesem Fall die Besetzung neu festlegen.
Inszenierung _inszenierung = Zeiger auf das äußere Inszenierungsobjekt
float _max = maximale, bisher errechnete Güte
HashSet<Ensemblemitglied> _besetzung = aktuelle Besetzung

```
this._n += 1;
if (_inszenierung.GueteDerBesetzung() > _max){
    _max = _inszenierung.GueteDerBesetzung();

_besetzung = new HashSet<Ensemblemitglied>();
foreach (var d in _inszenierung.Darsteller)
    if (d.IstBesetzt())
        _besetzung.Add(d.Besetzung);
```

```
this._n += 1;
if (_inszenierung.GueteDerBesetzung() > _max){
    _max = _inszenierung.GueteDerBesetzung();

    _besetzung = new HashSet<Ensemblemitglied>(
        _inszenierung.Darsteller
        .Where(d => d.IstBesetzt())
        .Select(d => d.Besetzung)
        .ToList());
```

private class Optimierer : public void Uebernehmen() - übernimmt eine temporäre Besetzung in eine Endgültige, wenn das Optimum der Güte gefunden ist.
Inszenierung _inszenierung = Zeiger auf das äußere Inszenierungsobjekt
HashSet<Ensemblemitglied> _besetzung = aktuelle Besetzung

```
var mitglieder = _besetzung.GetEnumerator();
if (mitglieder.MoveNext()) {
    foreach (Darsteller d in _inszenierung.Darsteller)
        if (d.IstBesetzt()) {
            d.BesetzenMit((Ensemblemitglied)mitglieder.Current);
            mitglieder.MoveNext();
        }
}
```

```
IEnumerable<Ensemblemitglied> mitglieder = _besetzung;

_inszenierung.Darsteller
    .Where(d => d.IstBesetzt())
    .ForEach(d => {
        d.BesetzenMit(mitglieder.FirstOrDefault());
        mitglieder = mitglieder.Skip(1);
    });
```

public void StabilVerheiraten() - ermittelt eine „stabile Besetzung“ (ideale Besetzung) von Darstellerposten (Männer) und Schauspielern (Frauen) und besetzt anschließend die Darstellerposten entsprechend.
IList GetPraeferenzen(Ensemblemitglied ensemblemitglied) = Auflistung der präferierten Darstellerposten eines Ensemblemitgliedes
List<Darsteller> Darsteller = Auflistung aller Darsteller(-Posten) der Inszenierung (Property aus Inszenierung)
ICollection<Ensemblemitglied> Schauspieler = gibt eine Auslistung aller Mitglieder des Ensembles zurück, die Schauspieler sind
IDictionary StabileHeirat.DictStabileHeirat = Auflistung der stabil verheirateten Darstellerposten und Schauspieler

```
Hashtable maenner = new Hashtable();

foreach (Darsteller d in Darsteller) {
    IList praeferenzen = GetPraeferenzen(d);
    maenner[d] = new ArrayList(praeferenzen);
}

Hashtable frauen = new Hashtable();
foreach (Ensemblemitglied schauspieler in Schauspieler) {
    IList praeferenzen = GetPraeferenzen(schauspieler);
    frauen[schauspieler] = praeferenzen;
}
```

```
Hashtable maenner = new Hashtable(); //Darsteller

var qMaenner = Darsteller
               .Select(d => new { d, liste = GetPraeferenzen(d) });

qMaenner.ForEach(entry => maenner[entry.d] = new ArrayList(entry.liste));

Hashtable frauen = new Hashtable(); // Schauspieler

Schauspieler
.Select(s => new { s, liste = GetPraeferenzen(s) })
.ForEach(entry => frauen[entry.s] = new ArrayList(entry.liste));
```

```
IDictionary vorschlag = StabileHeirat.DictStabileHeirat(maenner, frauen);
foreach (Darsteller darsteller in Darsteller){
    if (vorschlag.Contains(darsteller))
        darsteller.BesetzenMit((Ensemblemitglied)vorschlag[darsteller]);
    else
        darsteller.BesetzenMit(null);
}
```

```
IDictionary vorschlag = StabileHeirat.DictStabileHeirat(maenner, frauen);

qMaenner.Select(entry => entry.d)
        .ForEach(d => {
            if (vorschlag.Contains(d))
                d.BesetzenMit((Ensemblemitglied)vorschlag[d]);
            else
                d.BesetzenMit(null);
          });
```

private IList GetPraeferenzen(Ensemblemitglied ensemblemitglied) - gibt für ein übergebenes Ensemblemitglied eine Liste aller Darstellerposten zurück, die dessen Präferenz darstellen.
List<Darsteller> Darsteller = Auflistung aller Darsteller(-Posten) der Inszenierung (Property aus Inszenierung)
Dictionary<Ensemblemitglied, List<Darsteller>> persoenlichePraef = Auflistung aller bevorzugten Darstellerposten für die einzelnen Ensemblemitglieder

```
List<Darsteller> praeferenzen = new List<Darsteller>(Darsteller);

praeferenzen.Sort(new CompareEnsemblemitglied(this, ensemblemitglied));

if (persoenlichePraef.ContainsKey(ensemblemitglied)){
    int einfuegeposition = 0;
foreach (var persPraef in persoenlichePraef[ensemblemitglied]){
    praeferenzen.Insert(einfuegeposition, persPraef);
    praeferenzen.RemoveAt(praeferenzen.Count - 1);
    einfuegeposition = einfuegeposition + 1;
}
return praeferenzen;
```

```
List<Darsteller> praeferenzen = new List<Darsteller>(Darsteller);

praeferenzen.Sort(new CompareEnsemblemitglied(this, ensemblemitglied));

if (persoenlichePraef.ContainsKey(ensemblemitglied)){
    int einfuegeposition = 0;
    persoenlichePraef[ensemblemitglied]
        .ForEach(persPraef => {
            praeferenzen.Insert(einfuegeposition, persPraef);
            praeferenzen.RemoveAt(praeferenzen.Count() - 1);
            einfuegeposition = einfuegeposition + 1;
        });
return praeferenzen;
```

public Terminplan FreieTermineImZeitraum(DateTime von, DateTime bis, Theatertermin termin) - gibt einen Terminplan mit Terminen zurück, bei denen für einen übergebenen Zeitraum alle Ensemblemitglieder zur Verfügung stehen würden.

```
Termin zeitraum = new FreierTermin(von, bis);
Terminplan freieTermine = new FreierTerminplan();
freieTermine.Eintragen(zeitraum);
foreach (Terminplan terminplan in termin.ErwarteteEnsemblemitglieder)
    freieTermine =
freieTermine.DurchschnittMit(terminplan).FreieTermineImZeitraum(von, bis);
return freieTermine;
```

```
Termin zeitraum = new FreierTermin(von, bis);
Terminplan freieTermine = new FreierTerminplan();
freieTermine.Eintragen(zeitraum);
termin.ErwarteteEnsemblemitglieder
    .Select(mitglied => mitglied.Terminplan)
    .ForEach(terminplan => {
        freieTermine =
freieTermine.DurchschnittMit(terminplan.FreieTermineImZeitraum(von, bis));
    });
return freieTermine;
```

public ICollection<Raum> FreieRaeumeAn(Theatertermin termin) - gibt eine Auflistung aller Räume zurück, die an einem bestimmten Aufführungstermin frei und dafür geeignet sind.
List<Raum> _raeume = Auflistung aller Räume der Inszenierung

```
var freieRaeume = new List<Raum>();
foreach (Raum raum in _raeume)
    if (raum.IstFreiAn(termin) &&
            ((!(termin is Auffuehrungseinheit)) ||
            raum.Auffuehrungsgeeignet))
        freieRaeume.Add(raum);
return freieRaeume;
```

```
return _raeume
    .Where(r => (r.IstFreiAn(termin) &&
            (!(termin is Auffuehrungseinheit) ||
            r.Auffuehrungsgeeignet)))
    .ToList();
```

public void RolleStreichen(Rolle rolle) - streicht eine Rolle aus dem aktuellen Stück der Inszenierung bzw. den dafür angedachten Darstellern. Für Darsteller bei denen dies die einzige Rolle war, wird der Posten mit entfernt.
Stueck _stueck = Das aktuelle Stück der Inszenierung
List<Darsteller> Darsteller = Auflistung aller Darsteller(-Posten) der Inszenierung (Property aus Inszenierung)

```
_stueck.RolleStreichen(rolle);
foreach (Darsteller darsteller in Darsteller)
    if (darsteller.Rollen.Contains(rolle)){
        darsteller.RolleLoeschen(rolle);
        if (darsteller.Rollen.Count == 0)
            PostenLoeschen(darsteller);
    }
```

```
_stueck.RolleStreichen(rolle);
Darsteller
    .Where(d => d.Rollen.Contains(rolle))
    .ForEach(d => {
        d.RolleLoeschen(rolle);
        if (d.Rollen.Count() == 0)
            PostenLoeschen(d);
    });
```

A.2. Klasse: StabileHeirat

IDictionary DictStabileHeirat(Hashtable mapF, Hashtable mapM) - gibt eine Auflistung der besten Besetzung von Darstellern mit Schauspielern zurück.

```
IDictionary vorschlag = new Hashtable();
object p;
object e;
while ((mapM.Count != 0) && (mapF.Count != 0)){
    vorschlag = FindePaare(mapF, mapM);
    IEnumerator it = mapF.GetEnumerator();
    while (it.MoveNext()){
        DictionaryEntry entry = (DictionaryEntry)it.Current;
        p = (object)entry.Key;
        e = (object)vorschlag[p];
        if (e != null)
            mapM.Remove(e);
    }
}
return vorschlag;
```

```
IDictionary vorschlag = new Hashtable();
object p;
object e;
while ((mapM.Count != 0) && (mapF.Count != 0)){
    vorschlag = FindePaare(mapF, mapM);


    mapF.Cast<DictionaryEntry>()
        .ForEach(ent => {
            p = ent.Key;
            e = (object)vorschlag[p];
            if (e != null)
                mapM.Remove(e);
        });
}
return vorschlag;
```

IDictionary FindePaare(Hashtable mapF, Hashtable mapM) - implementiert einen Algorithmus, bei denen Männer (Darstellerposten) und Frauen (Schauspieler) einander so zugeordnet werden, das die jeweilige Besetzung am stabilsten (besten) ist

```
int[] next = new int[mapM.Count];
IDictionary prefer = new Hashtable();
IDictionary rank = new Hashtable();
IDictionary fiancee = new Hashtable();
IList praeferenzen;
IDictionary map;
object element1;
object element2;
object d;
int rankWS;
int rankWfW;
IEnumerator iterator_Mann;
IEnumerator iterator_Frau;
DictionaryEntry entry;

iterator_Mann = mapM.GetEnumerator();
while (iterator_Mann.MoveNext()) {
    entry = (DictionaryEntry)iterator_Mann.Current;
    element1 = (object)entry.Key;
    prefer[element1] = mapM[element1];
}

iterator_Frau = mapF.GetEnumerator();
while (iterator_Frau.MoveNext()) {
    entry = (DictionaryEntry)iterator_Frau.Current;
    d = (object)entry.Key;
    praeferenzen = (IList)entry.Value;
    map = new Hashtable();
    for (int j = 0; j < praeferenzen.Count; j++) {
        map[praeferenzen[j]] = j;
    }
    rank[d] = map;
}

iterator_Mann = mapM.GetEnumerator();
iterator_Mann.Reset();

for (int m = 0; m < mapM.Count; m++) {
    iterator_Mann.MoveNext();
    entry = (DictionaryEntry)iterator_Mann.Current;
    element1 = (object)entry.Key;
    do {
        d = (object)((IList)prefer[element1])[next[m]];
```

```
int[] next = new int[mapM.Count];
IDictionary prefer = new Hashtable();
IDictionary rank = new Hashtable();
IDictionary fiancee = new Hashtable();
IList praeferenzen;
IDictionary map;
object element1;
object element2;
object d;
int rankWS;
int rankWfW;




mapM.Cast<DictionaryEntry>()
    .ForEach(ent => prefer[ent.Key] = ent.Value);
mapF.Cast<DictionaryEntry>()
    .ForEach(ent => {
        praeferenzen = (IList)ent.Value;
        map = new Hashtable();
        int j = 0;

        praeferenzen
            .Cast<Object>()
            .ForEach(p => {
                j++;
                map[p] = j;
            });
        rank[ent.Key] = map;
    });

mapM.Cast<DictionaryEntry>()
    .Select(entry => new { entry, Counter = m++ })
    .ForEach(mann => {
        element1 = (object)mann.entry.Key;

        do
        {
            d = prefer.Cast<DictionaryEntry>()
                .Where(pref => pref.Key == element1)
                .Select(pref => (IList)pref.Value)
                .FirstOrDefault()[next[mann.Counter]];
```

```
        element2 = (object)fiancee[d];
        if (element2 != null) {
            rankWS = ((int)((IDictionary)rank[d])[element1]);
            rankWfW = ((int)((IDictionary)rank[d])[element2]);
            if (rankWS < rankWfW) {
                fiancee[d] = element1;
                element1 = element2;
            }
        }
        else {
            fiancee[d] = element1;
            element1 = element2;
        }
    }
    while ((element1 != null) && (++next[m] < mapF.Count));
}
return fiancee;
```

```
            element2 = (object)fiancee[d];
            if (element2 != null) {
                rankWS = ((int)((IDictionary)rank[d])[element1]);
                rankWfW = ((int)((IDictionary)rank[d])[element2]);
                if (rankWS < rankWfW) {
                    fiancee[d] = element1;
                    element1 = element2;
                }
            }
            else {
                fiancee[d] = element1;
                element1 = element2;
            }
        }
        while ((element1 != null) && (++next[mann.Counter] < mapF.Count));
    });
return fiancee;
```

A.3. Klasse: Ensemble

public bool IstFreiAn(Theatertermin termin) - prüft, ob für einen gegebenen Termin alle Mitglieder des Ensembles zur Verfügung stehen. Steht nur ein Mitglied nicht zur Verfügung, wird False zurückgegeben, sonst True.

```
foreach(var beteiligter
       in termin.ErwarteteEnsemblemitglieder)

    if (!(beteiligter.IstFreiAn(termin)))
        return false;
return true;
```

```
return !termin.ErwarteteEnsemblemitglieder
        .Any(b => !b.IstFreiAn(termin));
```

public void TerminEintragen(Theatertermin termin) - trägt einen gegebenen Termin für alle beteiligten Ensemblemitglieder ein (z.B. einen Proben- oder Aufführungstermin)

```
foreach (var beteiligter in termin.ErwarteteEnsemblemitglieder)
    beteiligter.TerminEintragen(termin);
```

```
termin.ErwarteteEnsemblemitglieder
    .ForEach(beteiligter => beteiligter.TerminEintragen(termin));
```

public void TerminLoeschen(Theatertermin termin) - trägt einen gegebenen Termin bei allen beteiligten Ensemblemitgliedern aus.

```
foreach (var beteiligter in termin.ErwarteteEnsemblemitglieder)
    beteiligter.TerminLoeschen(termin);
```

```
termin.ErwarteteEnsemblemitglieder
    .ForEach(beteiligter => beteiligter.TerminLoeschen(termin));
```

public override string ToString() - gibt den Namen des Ensembles (gekürzt mit … im Konstruktor des StringBuilders) sowie die Namen aller Mitglieder (sofern vorhanden) zurück.
ICollection<Ensemblemitglied> _mitglieder = Auflistung aller (Ensemble-)Mitglieder des Ensembles

```
var antwort = new StringBuilder(_name + "\n\tMitglieder:");
if (_ensemblemitglieder.Count == 0)
    antwort.Append(" keine");
else
```

```
var antwort = new StringBuilder(_name + "\n\tMitglieder:");
if (_ensemblemitglieder.Count() == 0)
    antwort.Append(" keine");
else {
```

```
   foreach(var mitglied in _ensemblemitglieder)
        antwort.Append("\n\t" + mitglied);
return antwort.ToString();
```

```
   _ensemblemitglieder
        .ForEach(mitglied => antwort.Append("\n\t" + mitglied));
return antwort.ToString();
```

A.4. Klasse: Person

public float Passt(Stueck stueck, Darsteller darsteller) - gibt für ein gegebenes Stück und einen Darsteller zurück, wie gut die darstellende Person zu den Rollen „passt“.

```
float passt = 0;
foreach (Rolle rolle in darsteller.Rollen)
    passt += Passt(stueck, rolle) / darsteller.Rollen.Count;
return passt;
```

```
return darsteller.Rollen
            .Select(rolle => (Passt(stueck, rolle) /
darsteller.Rollen.Count()))
            .Sum();
```

A.5. Klasse: Darsteller

public override string Name - Gibt für den aktuellen Darsteller die Namen der Rollen sowie den Besetzungsrang zurück.
List<Rolle> _rollen = Auflistung aller Rollen des Darstellers

```
var sRollen = new StringBuilder();
foreach (Rolle r in _rollen)
    sRollen.Append(r.ToString());
return "Rollen " + sRollen.ToString() + " in " + _besetzungsrang + ".
Besetzung";
```

```
var sRollen = new StringBuilder();
_rollen.ForEach(r =>
    sRollen.Append(r.ToString()));
return "Rollen " + sRollen.ToString() + " in " + _besetzungsrang + ".
Besetzung";
```

A.6. Klasse: Rollenprofil

public override string ToString() - gibt eine Zeichenfolge mit der Beschreibung des Rollenprofiles zurück.
IDictionary _akteUndSzenen = Ein Dictionary welches zu für jeden Akt die dazugehörige Szenen zugeordnet hat

```
String praefix = _rolle + "\n" +
                 "\tAnzahl SÃ¤tze: " + _anzahlSaetze + "\n" +
                 "\tAnzahl WÃ¶rter: " + _anzahlWoerter + "\n" +
                  "\tAkte und Szenen: " + "\n";

var antwort = new StringBuilder(praefix);
foreach (Akt akt in _akteUndSzenen.Keys){
    antwort.Append("\t\t" + akt.Name + "\n");
    foreach (Szene szene in (HashSet<Szene>)_akteUndSzenen[akt])
        antwort.Append("\t\t\t" + szene.Name + "\n");
}
return antwort.ToString();
```

```
String praefix = _rolle + "\n" +
                  "\tAnzahl SÃ¤tze: " + _anzahlSaetze + "\n" +
                  "\tAnzahl WÃ¶rter: " + _anzahlWoerter + "\n" +
                   "\tAkte und Szenen: " + "\n";

var antwort = new StringBuilder(praefix);

_akteUndSzenen.Cast<DictionaryEntry>()
.Select(a => new { akt = (Akt)a.Key, szenen = (HashSet<Szene>)a.Value })
.ForEach(entry => {
    antwort.Append("\t\t" + entry.akt.Name + "\n");
    entry.szenen.ForEach(szene => antwort.Append("\t\t\t" + szene.Name +
"\n"));
});
return antwort.ToString();
```

A.7. Klasse: Stück

public List<Rolle> Rollen - gibt eine Auflistung aller Rollen der Akte des Stückes zurück.
List<Akt> _akte = Auflistung aller Akte in diesem Stück

```
var rollen = new HashSet<Rolle>();
foreach (Akt akte in _akte)
    rollen.UnionWith(akte.Rollen);
return new List<Rolle>(rollen);
```

```
var rollen = new List<Rolle>();

_akte.ForEach(akt => {
    rollen.Union(akt.Rollen);
});
return rollen.ToList();
```

public void Bestimme(Rollenprofil rollenprofil) - bestimmt ein Rollenprofil, in dem die Akte, Szenen, sowie die Anzahl der Wörter und Sätze ermittelt werden.
List<Akt> _akte = Auflistung aller Akte in diesem Stück

```
foreach (Akt akte in _akte)
    akte.Bestimme(rollenprofil);
```

```
_akte.ForEach(akt => akt.Bestimme(rollenprofil));
```

public int AnzahlWoerter(Rolle rolle) - ermittelt die Anzahl der Wörter, die von einer Rolle gesprochen werden.
List<Akt> _akte = Auflistung aller Akte in diesem Stück

```
int anzahl = 0;
foreach (Akt akte in _akte)
    anzahl += akte.AnzahlWoerter(rolle);
return anzahl;
```

```
return _akte.Sum(akt => akt.AnzahlWoerter(rolle));
```

public override string ToString() - gibt den Namen des Stückes und die Anzahl der Akte (gekürzt im Konstruktor des StringBuilders) sowie die Namen der Akte zurück.
string _name = Name des Stückes
List<Akt> _akte = Auflistung aller Akte in diesem Stück

```
String praefix = "StÃ¼ck \"" + _name +
                 "\" in " + _akte.Count + " Akten";
var antwort = new StringBuilder(praefix);
foreach (Akt akt in _akte)
    antwort.Append("\n\t" + akt);
return antwort.ToString();
```

```
String praefix = "StÃ¼ck \"" + _name +
                   "\" in " + _akte.Count() + " Akten";
var antwort = new StringBuilder(praefix);
_akte.ForEach(akt => antwort.Append("\n\t" + akt));
return antwort.ToString();
```

public void RolleStreichen(Rolle rolle) - streicht eine Rolle aus jedem Akt des Stückes. Beinhaltet der Akt keine weiteren Rollen mehr, wird dieser ebenfalls gelöscht.
List<Akt> _akte = Auflistung aller Akte in diesem Stück

```
foreach (Akt akt in _akte){
    akt.RolleStreichen(rolle);
    if (akt.IstLeer())
        _akte.Remove(akt);
}
```

```
_akte
    .ForEach(akt => {
        akt.RolleStreichen(rolle);
        if (akt.IstLeer())
            _akte.Remove(akt);
    });
```

public object Clone() - erstellt einen Klon des Stückes und gibt diesen zurück.
List<Akt> _akte = Auflistung aller Akte in diesem Stück

```
Stueck klon = (Stueck)base.MemberwiseClone();
klon._akte = new List<Akt>();
foreach (Akt akt in _akte)
    klon._akte.Add((Akt)akt.Clone());
return klon;
```

```
Stueck klon = (Stueck)base.MemberwiseClone();
klon._akte = new List<Akt>();
_akte
    .Select(akt => akt.Clone())
    .ForEach(a => klon._akte.Add((Akt)a));
return klon;
```

A.8. Klasse: Akt

public List<Rolle> Rollen - gibt eine Auflistung aller Rollen des Aktes zurück
List<Szene> _szenen = Auflistung aller Szenen des Aktes

```
var rollen = new HashSet<Rolle>();
foreach (Szene szene in _szenen)
    rollen.UnionWith(szene.Rollen);
return new List<Rolle>(rollen);
```

```
var rollen = new HashSet<Rolle>();
_szenen.ForEach(szene => {
    rollen.Union(szene.Rollen);
});
return rollen.ToList();
```

internal int AnzahlWoerter(Rolle rolle) - ermittelt für eine gegebene Rolle die Anzahl der Wörter des Aktes.
List<Szene> _szenen = Auflistung aller Szenen des Aktes

```
int anzahl = 0;
foreach (Szene szene in _szenen)
    anzahl += szene.AnzahlWoerter(rolle);
return anzahl;
```

```
return _szenen
            .Select(szene => szene.AnzahlWoerter(rolle))
            .Sum();
```

public void Bestimme(Rollenprofil rollenprofil) - fügt für ein übergebenes Rollenprofil alle Szenen an, in der die Rolle auftritt.
List<Szene> _szenen = Auflistung aller Szenen des Aktes

```
foreach (Szene szene in _szenen)
    szene.Profil(rollenprofil, this);
```

```
_szenen.ForEach(szene => szene.Profil(rollenprofil,this));
```

internal void RolleStreichen(Rolle rolle) - streicht die übergebene Rolle aus allen Szenen. Wird die Szene leer, weil sonst keine weitere Rolle darin auftritt, wird die gesamte Szene gelöscht.
List<Szene> _szenen = Auflistung aller Szenen des Aktes

```
foreach (Szene szene in _szenen) {
    szene.RolleStreichen(rolle);
    if (szene.IstLeer())
        this._szenen.Remove(szene);
}
```

```
_szenen.ForEach(szene => {
        szene.RolleStreichen(rolle);
        if (szene.IstLeer())
            _szenen.Remove(szene);
});
```

public override string ToString() - gibt den Namen des Aktes sowie die Anzahl der enthaltenen Szenen (gekürzt mit … im Konstruktor des StringBuilders), sowie die Namen aller Szenen zurück.
List<Szene> _szenen = Auflistung aller Szenen des Aktes

```
var antwort = new StringBuilder("\tAkt \"" + _name + "\" mit " +
_szenen.Count + " Szenen");
foreach (Szene szene in _szenen)
    antwort.Append("\n\t\t" + szene);
return antwort.ToString();
```

```
var antwort = new StringBuilder("\tAkt \"" + _name + "\" mit " +
_szenen.Count() + " Szenen");
_szenen.ForEach(szene => antwort.Append("\n\t\t" + szene));
return antwort.ToString();
```

public object Clone() - erstellt einen Klon des Aktes (Deep Copy inkl. Klone der Szenen) und gibt diesen zurück.
_szenen = Auflistung aller Szenen des Aktes
List<Szene> _szenen = Auflistung aller Szenen des Aktes

```
Akt klon;
klon = (Akt)base.MemberwiseClone();
klon._szenen = new List<Szene>();
foreach (Szene s in this._szenen)
    klon._szenen.Add((Szene)s.Clone());
return klon;
```

```
Akt klon;
klon = (Akt)base.MemberwiseClone();
klon._szenen = new List<Szene>();
_szenen.Select(s => s.Clone())
       .ForEach(szene => klon._szenen.Add((Szene)szene));
return klon;
```

A.9. Klasse: Szene

public List<Rolle> Rollen - gibt eine Auflistung aller Rollen der Szene zurück.
List<Absatz> _absaetze = Auflistung aller Absätze dieser Szene

```
var rollen = new List<Rolle>();
foreach (Absatz a in _absaetze){
    Rolle rolle = a.Rolle;
    if (rolle != null)
        rollen.Add(rolle);
}
return rollen;
```

```
return _absaetze
       .Where(a => a.Rolle != null)
       .Select(a => a.Rolle)
       .Distinct().ToList();
```

internal int AnzahlWoerter(Rolle rolle) - gibt für eine gegebene Rolle die Anzahl der Wörter in dieser Szene zurück.
List<Absatz> _absaetze = Auflistung aller Absätze dieser Szene

```
int anzahl = 0;
foreach (Absatz absatz in _absaetze)
    anzahl += absatz.AnzahlWoerter(rolle);
return anzahl;
```

```
return _absaetze.Sum(a => a.AnzahlWoerter(rolle));
```

public void Profil(Rollenprofil rollenprofil, Akt akt) - bestimmt ein Rollenprofil, in dem die Akte, Szenen, sowie die Anzahl der Wörter und Sätze ermittelt wird.
List<Absatz> _absaetze = Auflistung aller Absätze dieser Szene

```
foreach (Absatz absatz in _absaetze)
    if (absatz.Bestimme(rollenprofil))
        rollenprofil.AddSzene(this, akt);
```

```
_absaetze.Where(a => a.Bestimme(rollenprofil))
         .ForEach(a => rollenprofil.AddSzene(this, akt));
```

internal void RolleStreichen(Rolle rolle) - streicht eine Rolle aus dieser Szene, d.h. die entsprechenden Absätze werden gelöscht
List<Absatz> _absaetze = Auflistung aller Absätze dieser Szene

```
foreach (Absatz absatz in _absaetze)
    if (absatz.Rolle == rolle)
        _absaetze.Remove(absatz);
```

```
_absaetze.Where(a => a.Rolle == rolle)
         .ForEach(a => _absaetze.Remove(a));
```

public override string ToString() - gibt eine Zeichenfolge mit dem Namen der Szene (gekürzt im Konstruktor des StringBuilders) sowie dem Inhalt der Abätze zurück.
List<Absatz> _absaetze = Auflistung aller Absätze dieser Szene

```
var antwort = new StringBuilder("\t\tSzene \"" + _name + "\"");
foreach (Absatz absatz in _absaetze)
    antwort.Append("\n\t\t\t" + absatz);
return antwort.ToString();
```

```
var antwort = new StringBuilder("\t\tSzene \"" + _name + "\"");
_absaetze.ForEach(a => antwort.Append("\n\t\t\t" + a));
return antwort.ToString();
```

public object Clone() - erstellt einen Klon der Szene und gibt diesen zurück.
List<Absatz> _absaetze = Auflistung aller Absätze dieser Szene

```
Szene klon = (Szene)base.MemberwiseClone();
klon._absaetze = new List<Absatz>();
foreach (Absatz absatz in _absaetze)
    klon._absaetze.Add((Absatz)absatz.Clone());
return klon;
```

```
Szene klon = (Szene)base.MemberwiseClone();
klon._absaetze = new List<Absatz>();
_absaetze.Select(a => a.Clone())
         .ForEach(abs => klon._absaetze.Add((Absatz)abs));
return klon;
```

A.10. Klasse: Terminplan

public virtual bool IstFreiAn(Termin t) - prüft, ob ein bestimmter Termin im Termplan noch frei ist.
_ List<Termin> _termine = Auflistung der vorhandenen Termine in diesem Terminplan

```
foreach (Termin termin in _termine)
    if (termin.KollidiertMit(t))
        return false;
return true;
```

```
return !_termine.Any(termin => termin.KollidiertMit(t));
```

public virtual Terminplan FreieTermineImZeitraum(DateTime von, DateTime bis) - gibt einen Termplan zurück, der freie Termine in einem gegebenen Zeitraum enthält.
_ List<Termin> _termine = Auflistung der vorhandenen Termine in diesem Terminplan

```
Termin zeitraum = new FreierTermin(von, bis);
Terminplan freieTermine = new FreierTerminplan();
freieTermine.Eintragen(zeitraum);
foreach (Termin termin in _termine)
    freieTermine
freieTermine.DurchschnittMit(termin.Differenz(zeitraum));
return freieTermine;
```

=

```
Termin zeitraum = new FreierTermin(von, bis);
Terminplan freieTermine = new FreierTerminplan();
freieTermine.Eintragen(zeitraum);
_termine.ForEach(termin => freieTermine =
                 freieTermine.DurchschnittMit(termin.Differenz(zeitraum)));
return freieTermine;
```

public virtual Terminplan DurchschnittMit(Terminplan tp) - gibt einen neuen Terminplan zurück, der die Termine enthält, die sich zwischen dem aktuellen Terminplan und einem übergebenen Terminplan tp überlappen.
_ List<Termin> _termine = Auflistung der vorhandenen Termine in diesem Terminplan

```
var termineA = new List<Termin>(_termine);
var termineB = new List<Termin>(tp._termine);
Terminplan e = new FreierTerminplan();
termineA.Sort();
termineB.Sort();
foreach (Termin a in termineA){
    foreach (Termin b in termineB){
        if (a.KollidiertMit(b))
            e.Eintragen(a.UeberlappungMit(b));
    }
}
return e;
```

```
var termineA =_termine.ToList();
var termineB = tp._termine.ToList();
Terminplan e = new FreierTerminplan();
termineA.Sort();
termineB.Sort();
termineA.ForEach(a => {
    termineB.Where(b => a.KollidiertMit(b))
            .Select(b => a.UeberlappungMit(b))
            .ForEach(t => e.Eintragen(t));
 });
return e;
```

private bool EnthaeltAnschliessendeTermine() - gibt True zurück, falls innerhalb dieses Terminplanes zwei Termine direkt aufeinanderfolgen. Sonst false.
_ List<Termin> _termine = Auflistung der vorhandenen Termine in diesem Terminplan

```
var temp = new List<Termin>(_termine);
temp.Sort();

IEnumerator termine = temp.GetEnumerator();
if (termine.MoveNext())       {
    Termin a = (Termin)termine.Current;
    while (termine.MoveNext()){
        Termin b = (Termin)termine.Current;
        if (a.AufeinanderfolgendMit(b))
            return true;
        a = b;
    }
}
return false;
```

```
IEnumerable<Termin> query = _termine.OrderBy(t => t).ToList();
Termin a = query.FirstOrDefault();

if (a != null) {
    query = query.Skip(1);
    foreach (Termin b in query) {
        if (a.AufeinanderfolgendMit(b))
            return true;
        a = b;
    }
}
return false;
```

public override string ToString() - gibt eine Zeichenfolge mit den Daten aller Termine des Terminplanes zurück
_ List<Termin> _termine = Auflistung der vorhandenen Termine in diesem Terminplan

```
var antwort = new StringBuilder();
antwort.Append("Terminplan:\n");
if (_termine.Count == 0)
    antwort.Append("\tist leer");
else
    foreach(Termin termin in _termine)
        antwort.Append("\t" + termin.ToString() + "\n");
return antwort.ToString();
```

```
var antwort = new StringBuilder();
antwort.Append("Terminplan:\n");
if (_termine.Count() == 0)
    antwort.Append("\tist leer");
else {
    _termine.ForEach(termin =>
             antwort.Append("\t" + termin.ToString() + "\n"));;
}
return antwort.ToString();
```

A.11. Klasse: Probentermin

public override string Anlass - gibt die Namen aller zu probenden Einheiten (Stück, Akte, Szenen) für diesen Probentermin aus.
_ICollection<Probeneinheit> _probeneinheiten = Auflistung alle Probeneinheiten des Probentermins (Stück, Akte, Szenen)

```
var antwort = new StringBuilder("Probentermin");
antwort.Append("\nProben: ");
if (_probeneinheiten.Count > 0) {

}
else
    antwort.Append("keine");

return antwort.ToString();
```

```
var antwort = new StringBuilder("Probentermin");
antwort.Append("\nProben: ");
if (_probeneinheiten.Count() > 0) {
    _probeneinheiten
        .ForEach(p => antwort.Append(p.Name + ", "));
    antwort.Remove(antwort.Length - 2, 2);

}
else
    antwort.Append("keine");

return antwort.ToString();
```

private ICollection<Posten> BenoetigteDarstellerposten - gibt eine Auflistung aller Darstellerposten zurück, die an dem Probentermin für die zu probenden Einheiten benötigt werden.
_ICollection<Probeneinheit> _probeneinheiten = Auflistung alle Probeneinheiten des Probentermins (Stück, Akte, Szenen)

```
var darstellerposten = new HashSet<Posten>();
var alleRollen = new HashSet<Rolle>();

foreach (Probeneinheit probeneinheit in _probeneinheiten)
    alleRollen.UnionWith(probeneinheit.Rollen);

foreach (Rolle rolle in alleRollen)
    darstellerposten.UnionWith(Inszenierung.DarstellerVon(rolle));
return darstellerposten;
```

```
return new HashSet<Posten>(
        _probeneinheiten
        .SelectMany(p => p.Rollen)
        .SelectMany(r => Inszenierung.DarstellerVon(r))
        .Distinct());
```

A.12. Klasse: Spieltermin

public override ICollection<Ensemblemitglied> ErwarteteEnsemblemitglieder - eine Auflistung aller Ensemblemitglieder zurück, die an dem Spieltermin erwartet werden. Es werden hier nur bereits besetzte Posten berücksichtigt.
ICollection<Posten> this.BenoetigtePosten = Auflistung aller Posten an diesem Termin (Probentermin oder Aufführungstermin)

```
var ensemblemitglieder = new HashSet<Ensemblemitglied>();
foreach (Posten posten in this.BenoetigtePosten) {
    Ensemblemitglied ensemblemitglied = posten.Besetzung;
    if (ensemblemitglied != null)
        ensemblemitglieder.Add(ensemblemitglied);
}
return ensemblemitglieder;
```

```
var ensemblemitglieder = new HashSet<Ensemblemitglied>();
return this.BenoetigtePosten
          .Where(posten => posten.Besetzung != null)
          .Select(posten => posten.Besetzung)
          .ToList();
```

B Inhalt der beiliegenden CD

Auf der beiliegenden CD befindet sich der folgende Inhalt

Verzeichnis	Inhalt
Ausarbeitung	Dieses Dokument als PDF-Datei.
Klassendiagramme	Die im Abschnitt 4.1 verwendeten Klassendiagramme als PNG-Datei.
Code\ PruefungLinqToAdo	C#-Projekt zum Prüfen der bidirektionalen Navigation von LINQ to SQL/Entities
Code\RelationshipManager	Testprojekt für den Relationship-Manager. Enhält die Assemblies von Andy Bulka.[43]
Code\TheaterMaxx	Das TheaterMaxx-Projekt in C#.
Tools\NClass	Die Software, mit der die Klassendiagramme erstellt wurden.
Tools\NET-Reflector	Der .NET-Reflector von RedGate

Die Steuerung der TheaterMaxx-Software erfolgt über ein auf der Konsole ausgegebenes Menü. Für die Performance-Messungen wurde im Hauptmenü ein zusätzlicher Untermenüpunkt Nr. 3 eingeführt. Testdaten liegen bereits im Debug- bzw. Release-Ordner und können beim Start des Programms, nach Rückfrage, eingelesen werden.

[43] Die Verwendung der Assemblies des Relationship-Managers wurde von Andy Bulka genehmigt.

Quelltextverzeichnis

Quelltext 3-1: Beispiel – Abfragesyntax 7
Quelltext 3-2: Beispiel – Methodensyntax 7
Quelltext 3-3: Beispiel – Select mit anonymem Typ 8
Quelltext 3-4: Die Erweiterungsmethode `Where` 9
Quelltext 3-5: Iteration über alle Ensemblemitglieder 11
Quelltext 3-6: Iteration über alle Ensemblemitglieder (Abfragesyntax) 11
Quelltext 3-7: Iteration über alle Ensemblemitglieder per `SelectMany` 12
Quelltext 3-8: Rückgabe einer Teilauflistung von Personen (Iterativ) 12
Quelltext 3-9: Rückgabe einer Teilauflistung von Personen (Abfragesyntax) 12
Quelltext 3-10: Rückgabe einer Teilauflistung von Personen (Methodensyntax) 13
Quelltext 3-11: Implementierung von `ToList` auf `IEnumerable` 13
Quelltext 3-12: Operator `Cast` 14
Quelltext 3-13: Ausgabe der Namen aller Personen über 30 Jahren (Methodensyntax) 14
Quelltext 3-14: Erweiterungsmethode `ForEach` 15
Quelltext 3-15: Ausgabe aller Personen über 30 Jahren (ForEach-Operator) 15
Quelltext 3-16: `ForEach`-Operator mit mehrzeiligem Anweisungsblock 15
Quelltext 4-1: Direkte Iteration 24
Quelltext 4-2: Gefilterte Iteration 24
Quelltext 4-3: Abfrage des ersten Elementes (`List`) 25
Quelltext 4-4 : Abfrage des ersten Elementes (`HashSet`) 25
Quelltext 4-5: Bilden einer Summe über eine numerische Eigenschaft 26
Quelltext 4-6: Bilden einer Summe über numerische Elemente 26
Quelltext 4-7: Zählen der Elemente einer Auflistung 26
Quelltext 4-8: Zählen einer Teilmenge von Elementen 27
Quelltext 4-9: Durchschnittswert berechnen 28
Quelltext 4-10: Geschachtelte Iteration über Auflistungen mit Unterauflistungen 28
Quelltext 4-11: Iteration komplexer Dictionaries 29
Quelltext 4-12: Geschachtelte Iteration zweier Auflistungen 30
Quelltext 4-13: Vereinigungsmengen 31
Quelltext 4-14: Vereinigungsmengen mit Distinct 32
Quelltext 4-15: Vorzeitiger Iterationsabbruch 32
Quelltext 4-16: Aufeinanderfolgende Elemente vergleichen 33
Quelltext 4-17: Gruppierungen 35
Quelltext 4-18: Gruppierungen (Optimiert) 36
Quelltext 4-19: Sortierte Gruppierungen 37
Quelltext 4-20: Maximum ermitteln 37
Quelltext 4-21: Element mit Maximum ermitteln 38
Quelltext 4-22: Element mit Maximum ermitteln (zwei Abfragen) 38
Quelltext 4-23: Element mit Maximum ermitteln (MaxElement) 39
Quelltext 4-24: Testquellcode Relationship-Manager (AndyPatterns) 41
Quelltext 4-25: Ausgabe der Absätze (AndyPatterns) 42
Quelltext 4-26: Ausgabe der Szenen (AndyPatterns) 42
Quelltext 5-1: Abfrage über mehrere Relationen 48

Tabellenverzeichnis

Tabelle 3-1: LINQ-Standardabfrageoperatoren 10
Tabelle 4-1: Messergebnisse – Direkte Iteration 24
Tabelle 4-2: Messergebnisse – Gefilterte Iteration 24
Tabelle 4-3: Messergebnisse – Abfrage des ersten Elementes 25
Tabelle 4-4: Messergebnisse – Bilden einer Summe über numerische Elemente 26
Tabelle 4-5: Messergebnisse – Zählen der Elemente einer Auflistung 26
Tabelle 4-6: Messergebnisse – Zählen einer Teilmenge von Elementen 27
Tabelle 4-7: Messergebnisse – Durchschnittswert berechnen 28
Tabelle 4-8: Messergebnisse – Geschachtelte Iteration über Auflist. mit Unterauflist. . 29
Tabelle 4-9: Messergebnisse – Iteration komplexer Dictionaries 30
Tabelle 4-10: Messergebnisse – Geschachtelte Iteration zweier Auflistungen 30
Tabelle 4-11: Messergebnisse – Vereinigungsmengen 32
Tabelle 4-12: Messergebnisse – Vorzeitiger Iterationsabbruch 32
Tabelle 4-13: Messergebnisse – Aufeinanderfolgende Elemente vergleichen 34
Tabelle 4-14: Messergebnisse – Algorithmus „Stabile Heirat“ 34
Tabelle 4-15: Messergebnisse – Gruppierungen 36
Tabelle 4-16: Messergebnisse – Sortierte Gruppierungen 37
Tabelle 4-17: Messergebnisse – Maximum ermitteln 37
Tabelle 4-18: Messergebnisse – Element mit Maximum ermitteln 39
Tabelle 4-19: Messergebnisse: Gesamtübersicht 44

Abbildungsverzeichnis

Abbildung 4-1: Klassenmodell `stamm` 18
Abbildung 4-2: Klassenmodell `inszenierung` 19
Abbildung 4-3: Klassenmodell `stueck` 19
Abbildung 4-4: Klassenmodell `termin` 20

Literaturverzeichnis

[BRU04] F. Thomas Bruss, „Strategien der ersten Wahl“, Spektrum der Wissenschaft Mai 2004, online verfügbar unter http://www.wissenschaft-online.de/artikel/835690

[HAL77] Maurice Howard Halstead, „Elements of software science“, Elsevier, New York u.a. 1977, ISBN 0-444-00205-7

[HEI07] A. Hejlsberg, D. Box, „LINQ: .NET Language-Integrated Query“, 2007, online verfügbar unter http://msdn.microsoft.com/fr-fr/library/bb308959(en-us).aspx

[KUE08] A. Kuehnel, „Visual C# 2008“, Galileo Press, 4. Auflage 2008, online verfügbar unter http://openbook.galileocomputing.de/visual_csharp/index.htm

[MAR08] F. Marguerie, S. Eichert, J. Wooley, „LINQ in Action“, Manning Publishing 2008, ISBN 978-1933988160

[MIC07] Microsoft Corp, „Visual C#-IntelliSense“, 2007, online verfügbar unter http://msdn.microsoft.com/de-de/library/43f44291.aspx

[MIC07a] Microsoft Corp, „Übersicht über Standardabfrageoperatoren“, 2007 , online verfügbar unter http://msdn.microsoft.com/de-de/library/bb397896.aspx

[MIC07b] Microsoft Corp., „Übersicht über Codemetrik“, 2007 , online verfügbar unter http://msdn.microsoft.com/de-de/library/bb385914.aspx

[MIC07c] Microsoft Corp., „Lernprogramm: Verketten von Abfragen“, 2007, online verfügbar unter http://msdn.microsoft.com/de-de/library/bb943899.aspx

[MIC09] Microsoft Corp., „Lambda Expressions“, 2009, online verfügbar unter http://msdn.microsoft.com/en-us/library/bb397687.aspx

[MIC09a] Microsoft Corp., „Parallel Programming in the .NET Framework”, 2009, online verfügbar unter http://msdn.microsoft.com/en-us/library/dd460693%28VS.100%29.aspx

[WIR98] N. Wirth, „Algorithmen und Datenstrukturen. Pascal-Version“, Teubner Verlag 5. Auflage November 1998, **ISBN-:** 3-519-22250-7

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Jens Duczmal
Bremen, den 21.Oktober 2009