Computer Science Resources
Developer's Guide

This document describes a set of coding standards and recommendations for programs written in the C# language

First Edition

C# Coding Style Guide

CodePlanet
<docs@codeplanet.eu>

Munich, 20 March 2008

Inhaltsverzeichnis

1 Einführung **4**
1.1 Über dieses Dokument . . . 4
1.2 Design Richtlinien . . . 4

2 Dateiorganisation **5**
2.1 Quelldateien . . . 5
2.2 Benennung von Quelldateien . . . 5
2.3 Aufbau der Projektverzeichnisse . . . 5

3 Sprache **6**

4 Namenskonventionen **6**
4.1 Hintergrund . . . 6
4.2 Zulässige Zeichen . . . 6
4.3 Ungarische Notation . . . 7
4.4 Minimale und maximale Anzahl Zeichen . . . 7
4.5 Gute Variablennamen . . . 7

5 Variablen **8**
5.1 Primitive Typen in der Programmiersprache . . . 8
5.2 Deklaration . . . 9
5.3 Initialisierung . . . 9
5.4 Gültigkeitsbereich . . . 9
5.5 Persistenz . . . 10

6 Layout und Stil **10**
6.1 Zeilenlänge . . . 10
6.2 Zeilenumbrüche . . . 10
6.3 Codeeinrückungen . . . 11
6.4 Zwischenräume . . . 11
6.4.1 Verwendung von Leerzeichen . . . 11
6.4.2 Vermeidung von Leerzeichen . . . 11
6.5 Klammern . . . 12
6.6 Datendeklarationen . . . 12

7 Kommentare **12**
7.1 Mehrzeilen-Kommentare . . . 12
7.2 Einzeilen-Kommentare . . . 13
7.3 Dokumentations-Kommentare . . . 14

8 Anweisungen **15**
8.1 Einfache Anweisungen . . . 15
8.1.1 Zuweisungen und Ausdrücke . . . 15
8.1.2 Lokale Variablendeklaration . . . 16
8.1.3 Array Deklarationen . . . 16
8.1.4 Return Anweisung . . . 16
8.2 Zusammengesetzte Anweisungen . . . 16
8.2.1 if-else Verzweigungen . . . 17
8.2.2 for- und foreach-Schleifen . . . 17
8.2.3 while- und do-while-Schleifen . . . 17
8.2.4 switch-Anweisungen . . . 17
8.2.5 goto-Anweisungen . . . 18
8.2.6 try-catch-finally Blöcke . . . 18

9 Analysetools **18**
9.1 FxCop . . . 19

10 Quellen 20

11 Anhang 21
.1 Beispiele für Namenskonventionen 21
.2 C# Quellcode Beispiel A 24
.3 C# Quellcode Beispiel B 25

1 Einführung

1.1 Über dieses Dokument

Dieses Dokument beschreibt eine Reihe von Standards und Richtlinien für die Entwicklung von Programmen in der Programmiersprache C# in einem konsistenten Stil. Der Fokus richtet sich auf das .NET und Mono Framework[1]. Darüberhinaus bezieht es sich auf die Sprachspezifikation **C# 2.0** und die vierte Version der *Common Language Infrastructure* (CLI). Diese werden von der internationalen Normungsorganisation ECMA im Ecma-334 und Ecma-335 Standard beschrieben. Der *C# Coding Style Guide* darf frei unter den Bedingungen der GPL verteilt werden, solange der Inhalt und die Form des Textes unverändert bleiben. Feedback in Form von Korrekturen oder Verbesserungsvorschlägen ist willkommen. Kommentare können an die Adresse docs@sun-projects.de1.cc gesendet werden.

Bei den empfohlenen Konventionen handelt es sich überwiegend um allgemein anerkannte Richtlinien, teilweise aber auch um gängige Standards in der Softwarebranche. Die Konventionen sind verbindlich und gelten solange, bis dieses Dokument durch eine neue Version abgelöst wird. Vorab wird darauf hingewiesen, dass Quelltexte von Projekten, die unter der Kontrolle einer Versionsverwaltung stehen, bei sich wiederholenden oder gravierenden Verstößen gegen die aufgelisteten Konventionen, vorbehaltlos von anderen Entwicklern in ihren alten Revisionszustand zurückversetzt werden dürfen. Unter diese Konventionen fallen auch Regelungen, die bei der Programmierung zu beachten sind, aber nicht bzw. nicht ausschließlich oder nur indirekt, den Quellcode betreffen. Hierzu gehören beispielsweise grundlegende Regeln über die Wahl von Namen, über den Umgang mit Dateien (wo abzulegen bzw. einzuchecken?), usw. Hintergrund ist, dass beispielsweise Regelungen bzgl. der Namensvergabe bereits in der Analyse- und Designphase von Bedeutung sind.

1.2 Design Richtlinien

Das Ziel der Design Richtlinien im .NET Framework ist es einen allgemeinen Standard einzuführen um eine Konsistenz und Vorhersagbarkeit der öffentlichen APIs zu erreichen. Aus diesem Grund ist jeder Softwareentwickler angehalten sich an diese Richtlinien zu halten wenn Klassen und Komponenten entwickelt werden sollen die das .NET Framework erweitern sollen. Ein inkonsistentes Design beeinflußt nachhaltig die Produktivität der Entwickler. Nicht konforme Komponenten funktionieren in der Regel weiterhin, sind aber in ihrem Potential eingeschränkt. Nachfolgend werden einige Vorteile eines konsistenten Stils aufgezählt:

- Die Lesbarkeit und somit die Wartung von Code wird erhöht.
- Der Codebestand kann leicht von verschiedenen Programmierern verwaltet werden. Dieser Vorteil ist besonders bei der Entwicklung von Software in einem Team von großer Bedeutung.
- Einmal von den Entwicklern erlernt, erlauben es die Richtlinien den Programmierern sich auf die Semantik des Quellcodes zu konzentrieren, anstatt Zeit für die korrekte Formatierung aufwenden zu müssen.
- Ein standardisierter Stil ermöglicht die Anwendung von automatisierten Tools die die Softwareentwicklung unterstützen.

Auch für die in diesem Dokument genannten Regeln gibt es Ausnahmen. Es ist nicht möglich alle denkbaren Situationen die sich in einem spezifischen Softwareprojekt ergeben von vornherein abzudecken. Es mag deshalb Situationen geben, in denen Abweichungen von den Richtlinien zugunsten eines besseren Designs ratsam sind, in der Regel dürften solche Fälle jedoch eine Ausnahme bilden.

Die genannten Standards sind allgemein gehalten und richten sich nicht an ein bestimmtes Projekt. Projektteams können einen begrenzten Satz an zusätzlichen Richtlinien für ihre Projekte auswählen, die als Untermenge das vorliegende Dokument erweitern.

[1] In diesem Dokument ist, sofern nicht anders ausgewiesen, mit der Bezeichnung .NET auch das Mono Framework gemeint.

2 Dateiorganisation

2.1 Quelldateien

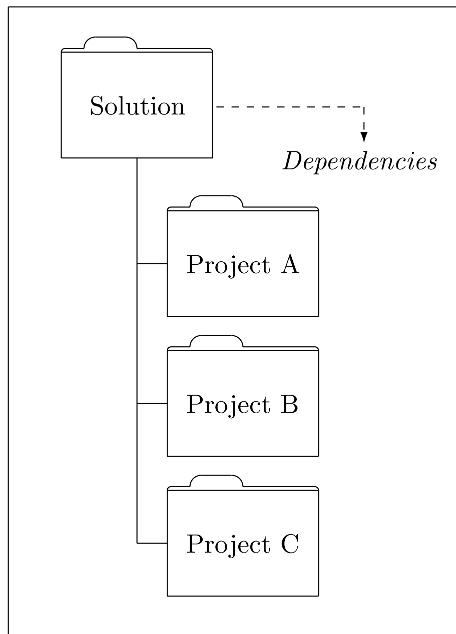
Die kleinste Einheit eines C# Projektes ist die Quelldatei. Eine C# Quelldatei sollte nur *eine* öffentliche Klasse (engl. public class) oder Schnittstellendefinition enthalten. Sie kann darüberhinaus zusätzliche nicht öffentliche Hilfsklassen und Schnittstellen beinhalten. Die Klassen bzw. Dateien sollten kurz gehalten werden. Damit ist gemeint das die Grenze von 2000 Zeilen Code nicht überschritten werden sollte. Der Code sollte klar strukturiert sein und in zusammenhängende Anschnitte unterteilt werden. Für die meisten Projekte sollten Quelldateien unter einem Versionskontrollsystem (z.B. CVS oder Subversion) verwaltet werden.

2.2 Benennung von Quelldateien

Gemäß den Java Konventionen sollte auch in C# jede Klasse in eine eigene Datei geschrieben werden. Dabei entspricht der Dateiname dem Namen der öffentlichen Klasse oder der Schnittstelle. Die Namenskonventionen für Dateien richten sich demnach nach den Konventionen für Klassennamen. Heißt die Klasse oder die Schnittstelle *MyFileWriter* so lautet der Dateiname *MyFileWriter.cs.* Auf diese Weise lassen sich Klassen in einem Projekt sehr schnell identifizieren.

2.3 Aufbau der Projektverzeichnisse

Beim Aufbau der Verzeichnisse ist auf eine einheitliche Struktur Wert zu legen. In dem Großteil aller Softwareprojekte kommt das *„single solution model“* zum Einsatz. Bei diesem Modell wird eine Projektmappe erzeugt die als Container für alle Projekte in der Anwendung dient.



Die Abhängigkeiten der einzelnen Projekte nehmen absteigend zu. Für jeden Namensraum (engl. Namespace) ist ein eigenes Verzeichnis anzulegen. Beispielsweise sollte der Namensraum *MyProject.TestSuite.Test* im Pfad *MyProject/TestSuite/Test* zu finden sein. Das erleichtert die Zuordnung der Namensräume zu der Verzeichnisstruktur.

3 Sprache

Gemäß dem internationalen Standard in der IT-Branche ist *Englisch* (US) die Standardsprache in der Softwareentwicklung. Das gilt nicht nur für die Auswahl von Namen für Klassen, Variablen und Verzeichnissen, sondern auch für die Kommentierung von Code und für die Commit Logs von neuen Revisionen in einem Repository.

4 Namenskonventionen

4.1 Hintergrund

Die in diesem Dokument genannten Namenskonventionen stellen allgemein akzeptierte Regeln für C# Programme dar. Obwohl es sich bei den Konventionen im Prinzip um flexibel auslegbare und nicht formal festgeschriebene Regeln handelt, haben sich diese in den letzten Jahren de facto zu einem Standard in der Industrie entwickelt. Die Namenskonventionen sollten in allen .NET Entwicklungen befolgt werden, einschließlich ASP.NET und Windows Forms Applikationen. In der MSDN Hilfe finden sich weitere detaillierte Informationen zu diesen Konventionen.

Konventionen weisen eine Reihe von Vorzügen auf die sich bei der Softwareentwicklung als äußerst vorteilhaft herausstellen.

- Konventionen nehmen Ihnen Entscheidungen ab.
- Der Quelltext eines fremden Projekts erschließt sich schneller. Der Programmierer muss sich nicht mehr auf den persönlichen Stil von verschiedenen anderen Programmierern einstellen.
- Konventionen bügeln Schwachstellen der Programmiersprache aus. Sie können Konventionen für die Emulation von Aufzählungstypen oder benannten Konstanten einsetzen. Über Konventionen können Sie leicht zwischen lokalen, klasseninternen und globalen Variablen unterscheiden.
- Mithilfe von Konventionen können Programmierer eine projektübergreifende Wissensbasis aufbauen.
- Konventionen sind ein gutes Gegengift gegen die kaninchenartige Vermehrung von Namen. Ohne feste Konventionen können Sie ein und derselben Variablen zig Namen geben. Beispielsweise könnte eine Variable für die Angabe einer Gesamtpunktzahl *pointTotal* und *totalPoint* heißen.

In der Praxis ist jede Konvention besser als gar keine. Der positive Einfluss von Namenskonventionen beruht nicht auf spezifischen Regeln, sondern eher auf der bloßen Existenz einer Konvention, die sozusagen Querstreben in den Quelltext einzieht und Ihnen viele Entscheidungen abnimmt. Im Anhang unter Punkt .1 finden Sie einige Beispiele und Beschreibungen zu den in C# verwendeten Namenskonventionen.

4.2 Zulässige Zeichen

Alle Namen dürfen nur die Buchstaben von A bis Z und in Ausnahmefällen Ziffern enthalten. Der Gebrauch von Unterstrichen („_“) zur Trennung von Wortbestandteilen ist für alle Arten von Bezeichnern untersagt. Dies gilt gleichermaßen für Klassen-, Variablen-, Prozedur- und Dateinamen. Um Wortbestandteile sichtbar zu machen, ist GroßKleinSchreibung („Intercaps“, „mixed case“) zu verwenden. Bestandteile von Namen, die normalerweise ausschließlich aus Großbuchstaben bestehen – wie beispielsweise viele Abkürzungen – werden nur mit einem großen Anfangsbuchstaben geschrieben.

4.3 Ungarische Notation

Die Ungarische Notation (engl. Hungarian Notation) ist besonders vorteilhaft für nicht strengtypisierte Sprachen wie C, für die sie ursprünglich entwickelt wurde. Für moderne, objektorientierte Hochsprachen wie C# oder Java ist sie nicht mehr von relevanter Bedeutung. Microsoft rät daher von der Benutzung der ungarischen Notation im .NET Framework ab.

Die Hauptintention dieser Notation, nämlich die Zuordenbarkeit von Typ zu Variable, kann von aktuellen Entwicklungsumgebungen besser und zuverlässiger geboten werden.

4.4 Minimale und maximale Anzahl Zeichen

Der Wunsch nach kurzen Variablennamen ist in der Regel nichts anderes als ein Relikt aus der guten alten Zeit des Computers. Sprachen wie Assembler, Basic oder Fortran erlaubten nur 2 bis 8 Zeichen für Variablennamen und zwangen den Programmierer kurze Namen zu ersinnen. Die Programmiersprachen waren eng an die Mathematik angelehnt und Quelltexte wirkten auf Außenstehende wie kryptische Dokumente.

Moderne Sprachen kennen kaum Begrenzungen der Länge von Variablennamen. Bezeichner sollten deshalb mindestens 4 und maximal 15 Zeichen lang sein. In der Praxis haben sich Namen mit einer Zeichenlänge zwischen 10 und 16 Zeichen bewährt. Namen, die aus weniger als 4 Zeichen bestehen, sind fast immer unverständlich oder nicht eindeutig. Zu lange Namen vermindern die Effektivität durch zu viel Tipparbeit bei der Eingabe. Allgemein übliche Abkürzungen, die nur zwei oder drei Zeichen lang sind, dürfen verwendet werden. Indexvariablen dürfen 1 Zeichen lang sein.

4.5 Gute Variablennamen

Ein Variablenname soll den Sachverhalt, den die Variable repräsentiert, vollständig und genau beschreiben. Halten Sie in einfachen Worten fest, wofür die Variable steht. Ein guter mnemonischer Name verweist normalerweise auf das Problem, und nicht auf die Lösung. Benutzen Sie Gegensatzpaare in Variablennamen.

- begin / end
- first / last
- locked / unlocked
- min / max
- old / new
- opened / closed
- up / down
- next / previous
- source / target

Schleifenvariablen werden in der Regel einfach nur mit i, j oder k bezeichnet. Wenn eine Schleife etwas länger ist und mit anderen Schleifen ineinander verschachtelt wird, gerät die Bedeutung von i leicht in Vergessenheit. In diesen Fällen sollten Schleifenvariablen einen sinnvolleren Namen erhalten.

```
for(int teamIndex = 0; teamIndex < teamCount; teamUndex++) {
    for(int eventIndex = 0; eventIndex < eventCount[teamIndex]; eventIndex++) {
        score[teamIndex][eventIndex] = 0;
    }
}
```

Mit sorgfältig gewählten Namen für Schleifenvariablen entschärfen Sie auch das bekannte Problem der Verwechslung dieser Variablen: score[teamIndex][eventIndex] ist einfach informativer als score[i][j].

Statusvariablen sollten nicht einfach mit flag oder statusFlag bezeichnet werden. Verwenden Sie stattdessen aussagekräftige Namen wie dataReady oder recalcNeeded. Verwenden statt 0x01, 0x02 und 0x03 sinnvolle Aufzählungstypen.

```
enum Days {
    Sat,
    Sun,
    Mon,
    Tue,
    Wed,
    Thu,
    Fri
};
```

Auch boolsche Variablen sollten Richtlinien folgen. Boolsche Variablen sollten Namen tragen die True beziehungsweise False implizieren. Namen wie done oder success eignen sich für boolsche Variablen, da sie sozusagen binär sind; es gibt nur wahr oder falsch, ja oder nein - nichts dazwischen. Namen wie sourceFile oder status sind ungeeignet, da sie nicht eindeutig wahr oder falsch sind. Verwenden Sie sattdessen Namen wie *done*, *error*, *found*, *success* oder *ok*.

5 Variablen

5.1 Primitive Typen in der Programmiersprache

Primitive Datentypen (pimitive types, dt. auch Grundtypen) werden direkt auf die Typen der Framework Class Library (FCL) abgebildet. In C# stehen vordefinierte Typen zur Verfügung die einem FCL-Typen des Systems entsprechen. Beispielsweise ist das Schlüsselwort *int* stellvertretend für *System.Int32*. Diese zweideutige Abbildung ermöglicht es Entwicklern zwischen den vordefinierten Typen und den FCL-Typen zu wählen und auf diese Weise einen inkonsistenten Stil in den Quellcode einzubringen. Dies führt in der Regel zu Verwirrungen sobald das Projekt von mehreren Personen verwaltet wird. So kann ein Programmierer das Schlüsselwort *string* verwenden, ein anderer widerum *String*. Beide werden letztenendes auf System.String abgebildet.

Die Sprachspezifikation von C# definiert: „Es ist guter Stil, das Schlüsselwort zu verwenden, nicht den vollständigen Typnamen“. Es gibt berechtigte Gründe dieser Anweisung zu folgen oder aber sie ganz zu verwerfen. Aufgrund der Interoperabilität der Sprachen in .NET ist nicht sichergestellt das auch in anderen Sprachen das Schlüsselwort auf den erwarteten FCL-Typen abgebildet wird. In C# wird *long* auf System.Int64 abgebildet. In anderen Sprachen könnte long jedoch auf Int16 oder In32 abgebildet werden. Zudem definiert die FCL viele Methoden, die Typbezeichnungen im Namen tragen. Zum Beispiel bietet der Typ BinaryReader Methoden wie ReadBoolean, ReadInt32, ReadSingle und so weiter. Dadurch können bestimmte Codefragmente auf den ersten Blick unklar erscheinen:

```
BinaryReader br = new BinaryReader(...);
float  val = br.ReadSingle(); // seems strange
Single val = br.ReadSingle(); // plausible
```

Dieses Dokument folgt den Empfehlungen der ECMA und verwendet das Schlüsselwort. In jedem Fall sollten Sie vor dem Projektbeginn eindeutig definieren welchen Weg Sie einschlagen wollen. Wenn Sie sich für die Verwendung der C# Schlüsselwörter entscheiden, sollten Sie diese konsequent im Quellcode verwenden und nicht mehr auf die FCL-Typen zurückgreifen. Gleiches gilt für den umgekehrten Fall.

5.2 Deklaration

Die Programmiersprache C# verlangt das Variablen explizit deklariert werden. Variablen sollten sich an die Namenskonventionen halten, so dass ihre Bedeutung schnell ersichtlich wird. Jede Variable soll nur einem Zweck dienen und dort deklariert und definiert werden, wo sie zum ersten Mal benutzt wird.

```
int recordIndex = 0;
while (recordIndex < 10) {
    recordIndex += 2;
    ++recordIndex;
}
```

Bei der Deklaration einer Variablen sollte darüber entschieden werden, ob es vernünftig ist die Variable mit dem Schlüsselwort *const* zu deklarieren.

5.3 Initialisierung

Die Initialisierung von Daten ist eine der häufigsten Fehlerquellen in der Softwareentwicklung. Obwohl C# und das .NET Framework so konzipiert wurden das klassische Fehlerquellen, wie die Verwendung von nicht initialisierten Variablen, unzulässig sind gibt es dennoch einige Punkte zu beachten. Die Richtlinien legen fest das jede Variable direkt in der Deklaration initialisiert werden sollte.

```
int childNum = 0;
float studentGrades[MAX_STUDENTS] = {0.0};
```

Ist eine Initialisierung bei der Deklaration nicht möglich, so ist die Variable in der Nähe der Stelle zu initialisieren, an der sie zum ersten Mal benutzt wird. Memberdaten einer Klasse sollten stets im Konstruktor initialisiert werden. Warnungen des Compilers über nicht initialisierte und nicht referenzierte Variablen sind zu beachten!

5.4 Gültigkeitsbereich

Der Gültigkeitsbereich (engl. Scope) einer Variablen ist so klein wie möglich zu halten. Die Lebensdauer und Spanne einer Variablen ist kurz zu halten. Eine Spanne stellt die Beziehung zwischen Variable und all ihrer Referenzen dar. Sie errechnet sich aus dem arithmetischen Mittelwert sämtlicher Spannen.

```
int a = 1;
int b = 2;    // b is declared and defined
int c = 3;

for (int i = 0; i < 10; ++i) {
    a *= i;
}

b = a + 1;    // b has span 6
b = b * c;    // b has span 0

// Result: (6 + 0) / 2 = 3.0
```

Referenzen und Variablen sollten dicht beieinander liegen. Auf diese Weise kann die Lebensdauer und Spanne so kurz wie möglich gehalten werden.

Bemerkung: Falls ein Dialog initialisiert wird, sollte die Verwendung der using-Anweisung in Erwägung gezogen werden. Dies gilt im Übrigen für alle Situationen in denen Dispose() so früh wie möglich aufgerufen werden soll. Das folgende Beispiel veranschaulicht diese Situation, indem für das Objekt theFont ein Scope definiert wird.

```
using (Font theFont = new Font("Arial", 10.0f)) {
    // theFont is used
}   // Compiler calls Dispose on theFont
```

5.5 Persistenz

Die Persistenz, also die Lebensdauer eines bestimmten Datenelements, kann unterschiedliche Formen annehmen. Variablen bleiben zum Beispiel bestehen bis zum Ende eines Codeblocks oder einer Routine oder bis zu dem Zeitpunkt an dem der Garbage Collector sie beseitigt. Stellen Sie sicher das Sie die Lebensdauer so kurz wie möglich halten und gegebenenfalls explizit diese beenden. In C# werden Objekte mit der *nicht-deterministischen* Finalisierung beseitigt. Ein expliziter Aufruf des Destruktors ist nicht zulässig und für verwaltete Objekte nicht ratsam. Bei der Arbeit mit unverwalteten Code sollte das Interface IDisposable implementiert werden. Für konstant deklarierte Variablen ist die Persistenz nicht von Bedeutung.

Ein Variablenname soll vollständig und genau beschreiben, wofür eine Variable steht. Sie soll an einen realen Sachverhalt angelehnt sein und nicht Internas über die Rechnerarechitektur preisgeben. Der Name soll weder zu kurz noch zu lang sein. Temporäre Variablen sollten sinnvollere Namen als einfach nur *temp* erhalten. Halten Sie sich an die .NET Namenskonventionen und befolgen Sie einen *konsistenten Stil* in Ihrem Code und darüberhinaus im gesamten Projekt.

6 Layout und Stil

6.1 Zeilenlänge

Alle Zeilen sollten maximal *80 Zeichen* lang sein. Die Zeilenlänge kann diese Vorgabe bei Bedarf auch um bis zu 15 Zeichen überschreiten. In der Regel sollte man die Zeilen jedoch nicht zu lang werden lassen. Je nach Bildschirmauflösung kann eine sichtbare Codezeile auf einem Bildschirm mit niedrigen Auflösungen über den Bildschirm hinausragen, so dass ein Scrolling notwendig wird. Die Vorgabe der Zeilenlänge schafft eine einheitliche Richtlinie und verhindert das die maximalen Zeilenlängen zu stark variieren.

6.2 Zeilenumbrüche

Wenn ein Ausdruck nicht in eine einzelne Zeile passt, so muss diese Zeile nach einheitlichen Regeln getrennt werden. Dabei gelten die folgenden

- Trenne nach einem Komma.
- Trenne nach einem Operator.
- Bevorzuge höherwertige Trennungen gegenüber niederwertigen Trennungen.
- Synchronisiere das erste Zeichen der alten Zeile, mit dem die Signatur begonnen hat, mit dem ersten Zeichen der neuen Zeile. Die Zeichen sollten in derselben Spalte stehen.

Beispiel wie man eine Methode in einer weiteren Zeile fortführt:

```
longMethodCall(expr1, expr2,
               expr3, expr4, expr5);

var = a * b / (c - g + f) +
      4 * z;
```

6.3 Codeeinrückungen

Ein allgemeiner Standard für die Art und Weise wie Zeilen in Quellcodes eingerückt werden, konnte sich bisher nicht durchsetzen. Deshalb wird hiermit festgelegt das Zeilen stets und auf allen Ebenen mit **4 Leerzeichen** (engl. Spaces) eingerückt werden müssen. Nachfolgend werden einige Vorteile einer durchgängigen Anwendung von Leerzeichen angeführt:

- Die Verwendung von Leerzeichen vermeidet Darstellungsfehler und stellt sicher das der Quellcode in jedem Editor konsistent gerendert wird. Dies ist mit Tabs nur unter bestimmten Voraussetzungen zu gewährleisten.
- Die strikte Anwendung dieser Richtlinie garantiert das der Quellcode einheitlich formatiert wird. Das gilt nicht nur nur für Codeeinrückungen, sondern auch für Codeanordnungen.
- Moderne Editoren erlauben es der Tab-Taste eine bestimmte Anzahl an Leerzeichen zuzuordnen. Auf diese Weise kann der Code mir derselben Funktionalität bearbeitet werden, wie das auch mit Tabs möglich wäre. Soll der Grad der Einrückung reduziert werden ist das mit diesen Editoren auch mit Shift-Tab problemlos möglich.
- Der Nachteil einer größeren Datei bei Anwendung von Leereichen, ist unter dem Gesichtspunkt der Leistungsfähigkeit aktueller Systeme nicht mehr als relevant einzustufen.
- Viele namhafte Projekte und Institutionen verwenden Leerzeichen. Sowohl die offiziellen „Java™Coding Style Guides“ als auch das „PEAR Projekt“ und „C++ Boost“ geben diese Richtlinie vor.

Die Anwendung von Tabs hat unbestritten gewisse Vorteile. Wägt man Vor- und Nachteile beider Verfahren sorgfältig gegeneinander ab, so ist der Anwendung von Leerzeichen für Codeeinrückungen Priorität einzuräumen.

6.4 Zwischenräume

Durch Zwischenräume wird die Lesbarkeit von Quellcode signifikant erhöht. Zwischenräume werden mithilfe von Leerzeichen, Tabulatoren und Zeilenumbrüchen generiert. Achten Sie darauf Ihren Code sinnvoll zu gliedern und gestalten Sie Ihre Zeilen übersichtlich. Setzen Sie einzelne Bedingungsabfragen in komplexen Ausdrücken in getrennte Zeilen.

6.4.1 Verwendung von Leerzeichen

1. Zwischen einem Schlüsselwort und der zugehörigen öffnenden runden Klammer sollte ein Leerzeichen eingefügt werden. Diese Regel trifft z.B. auf die Schlüsselwörter *for*, *if*, *switch*, *while*, *foreach* und *catch* zu. Sie gilt nicht für die Schlüsselwörter *this* oder *base*.
2. Nach ledem Schlüsselwort das ein Argument entgegennimmt.
3. Zwischen zwei angrenzenden Schlüsselwörtern.
4. Zwischen einem Schlüsselwort und schließeneden runden Klammern und öffnenden geschweiften Klammern „{„.
5. Vor und nach einem binären Operator mit Ausnahme des Punktoperators „.“.
6. Nach einem Komma in einer Liste.
7. Nach einem Semikolon in einer *for* Anweisung.

6.4.2 Vermeidung von Leerzeichen

1. Zwischen einem Methodennamen und der zugehörigen öffnenden runden Klammer.
2. Vor oder nach dem Punktoperators „.“.
3. Zwischen einem unärem Operator und seinem Operanden.

4. Zwischen einem Cast und dem Ausdruck der gecastet wird.
5. Nach einer öffnenden runden Klammer oder vor einer schließenden runden Klammer.
6. Nach einer schließenden rechteckigen Klammer „]“.

```
int? x = 123;
int? y = null;

if (x.HasValue) {
    Console.WriteLine(x.Value);
}
if (y.HasValue) {
    Console.WriteLine(y.Value);
}

a += c[i + j] + (int)d + foo(bar(i + j), e);

// White Space between if and (
if ((('0' <= inChar) && (inChar <= 'g')) ||
    (('a' <= inChar) && (inChar <= 'z')) ||
    (('A' <= inChar) && (inChar <= 'g')))
    ...
```

6.5 Klammern

Benutzen Sie Klammern um mathematischen Ausdrücken einen Sinn zu geben. Der Anweisungsblock von Kontrollstrukturen muss auch dann mit Klammern eingeschlossen werden, wenn nur eine Anweisung im Block steht.

```
if (success) {
   CallMethod();
}
```

6.6 Datendeklarationen

Deklarieren Sie nur eine Variable pro Zeile. Sie sollten jede Deklaration in eine eigene Zeile schreiben. Auf diese Weise lassen sich die Variablen leichter kommentieren. Syntaktische Fehler sind leichter auszubügeln, da die Zeilennummer, die mit der Fehlermeldung des Compilers angezeigt wird, nur eine Deklaration enthält.

```
int rowIndex; // represents the current row
Color[NUM_COLORS] choices; // screen color
Point nextTop;
```

7 Kommentare

7.1 Mehrzeilen-Kommentare

Die bekannten Blockkommentare /**/ sollten auch in C# vermieden werden, da sie nicht geschachtelt werden können. An ihre Stelle treten Kommentare die durch drei Slashes gekennzeichnet werden. Neue Entwicklungsumgebungen, wie Visual Studio 2005 unterstützen diese Kommentare mit zusätzlichen Features. Falls dennoch mehrzeilige Kommentare gebraucht werden, um beispielsweise bestimmte Dokumentationssysteme für Quelldateien verwenden zu können, ist folgender Stil zu verwenden:

```
/**
 * Documentation Comments.
 */

/* Old
 * Style
 * Block Comment.
 */

/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *    one
 *        two
 *            three
 */
```

In der Regel finden Mehrzeilen-Kommentare dieser Art Anwendung in der Auskommentierung von Code. Die Auskommentierung von Code sollte stets begründet werden und sich eindeutig vom restlichen Quellcode hervorheben.

Abzuraten ist von den Einzeilen-Kommentaren mithilfe des Mehrzeilen-Kommentar Ausdrucks. Diese Methode ist nur noch selten in alten C-Quelldateien zu finden.

```
/* Old Fashioned - not recommended. */
```

7.2 Einzeilen-Kommentare

Einzeilen-Kommentare sollten mit dem // Ausdruck beginnen. Die Kommentare müssen soweit eingerückt werden, dass sie mit dem Einrückungsgrad der aktuellen Codezeile übereinstimmen. Die Kommentare beziehen sich dabei auf die nachfolgenden Zeilen. Es ist ratsam auch ganze Codeblöcke auf diese Weise auszukommentieren. Soll ein Codeblock auskommentiert werden, so ist aufgrund einer besseren Sichtbarkeit der Code bereits ab der ersten Spalte auszukommentieren.

```
    private int QuiescentSearch(int alpha, int beta)
    {
//      Reason for outcommenting.

//      // exit if the move was aborted
//      if (beta < alpha) {
//          throw new InvalidOperationException();
//      }

//      pvLength[ply] = ply;

//      // check if it's too deep
//      if (ply >= MAX_PLY - 1) {
//          return Evaluate(alpha);
//      }

        return Evaluate(alpha);
    }
```

Ein Einzeilen-Kommentar sollte eine kurze und prägnante Erklärung abgeben und sich, wie der Name schon vermuten lässt nicht über mehr als eine Zeile erstrecken. Zu lange Einzeilen-Kommentare

sind ein Indiz für zu komplexen oder schlechten Code. Das nachfolgende Beispiel zeigt, wie sich Variablen und RegEx Ausdrücke kommentieren lassen. Beachten Sie, dass innerhalb von regulären Ausdrücken in C# der aus Perl stammende Ausdruck # (Rautesymbol) verwendet werden muss.

```
public static int Parse(string source)
{
    // verbatim string, ignoring spaces (@"(?x))
    Regex re = new Regex(@"          # This pattern matches Foo
                        (?i)         # turn on insensitivity
                                     # The Foo bit
                        \b(Foo)\b", RegexOptions.IgnorePatternWhitespace);

    bool found = false;  // my var

    found = re.IsMatch(source);

    if (found) {
        return 0x67;
    }

    return 0x68;
}
```

7.3 Dokumentations-Kommentare

Mit dem .NET Framework und der Sprache C# wurde ein neues Dokumentationssystem basierend auf XML Kommentaren eingeführt. Bei den Kommentaren handelt es sich in der Regel um Einzeilen-Kommentare die XML-Tags beinhalten und mit /// ... beginnen. Kommentare die sich über mehrere Zeilen erstrecken haben die Form /** ... */. Kommentare können entweder mithilfe des Compilers und dem Befehl /doc:<xml_outfile.xml> extrahiert werden oder unter Verwendung von entsprechenden Tools. Beispiele für die Verwendung von XML Tags in C# können wie folgt aussehen:

```
/// <summary>
/// This class provides methods for File I/O. It is
/// based on...
/// </summary>

/**
* <remarks>
* Class <c>Point</c> models a point in a two-dimensional plane.
* </remarks>
*/
public class Point
{
    /// <remarks>Method <c>Draw</c> renders the point.</remarks>
    void Draw() { ... }
}

/// <exception cref="IOException">The specified file does not exist.</exception>
```

Allen Einzeilen-Dokumentations-Kommentaren /// sollte ein Leerzeichen folgen. Dieses Zeichen ist in der XML Ausgabe nicht enthalten. Der Text innerhalb der Dokumentations-Kommentare muss nach den gängigen XML Regeln (http://www.w3.org/TR/REC-xml) formuliert werden. Der Compiler generiert bei invaliden Kommentaren eine Fehlermeldung. Ihnen steht es frei eigene XML

Tag	*Beschreibung*
<c>	Markiert einen Teil des Kommentars der als Code formatiert werden soll
<code>	Wie oben, aber mehrzeilig
<example>	Für das Einbetten von Beispielen in Kommentaren
<exception>	Dokumente und Exception Klassen
<include>	Inkludiert Dokumentationen aus anderen Dateien
<list>	Eine Liste an <term>s definiert durch <description>s
<para>	Strukturiert Textblöcke, z.B. in einem <remark>
<param>	Beschreibt den Parameter einer Methoden
<paramref>	Zeigt an das ein Wort als Referenz auf einen Parameter dient
<permission>	Gibt die Zugriffsberechtigung auf ein Memberobjekt an
<remarks>	Für den Überblick über das was der Typ oder die Klasse macht
<returns>	Beschreibung des Rückgabewertes
<see>	Referenz auf einen Member oder auf ein Feld
<seealso>	Wie oben, zeigt zusätzlich einen 'See also' Bereich an
<summary>	Die Zusammenfassung für ein Objekt

Tabelle 1: XML Tags

Tags zu definieren, solange diese nicht mit bestehenden Tags kollidieren. Allgemein bekannte Tags sind der nachfolgenden Tabelle 1 zu entnehmen.

Alle Dokumentations-Kommentare müssen mit /// beginnen. Die XML Tags lassen sich in die folgenden Kategorien einteilen:

- Dokumentation
- Formatierung / Referenzierung

Die erste Kategorie enthält Tags, wie <summary>, <param> oder <exception>. Diese Tags repräsentieren die Punkte, die die Elemente der Programm API beschreiben, welche dokumentiert werden sollten um anderen Programmierern das Lesen des Quellcodes zu erleichtern. Diese Tags besitzen gewöhnlich Attribute, wie cref="" dessen Anwendung im oben gezeigten Beispiel demonstriert wird.
Die zweite Kategorie beinhaltet Tags die der Formatierung dienen. Das können <code>, <list> oder <c> sein. Diese Tags produzieren das Layout der Dokumentation oder referenzieren andere Stellen in der Dokumentation.

Für die Erstellung einer *konsistenten* und *robusten* Dokumentation ist auf einen großzügigen Gebrauch dieser Tags Wert zu legen. Jedoch sollte darauf geachtet werden den Code sinnvoll zu dokumentieren!

8 Anweisungen

8.1 Einfache Anweisungen

8.1.1 Zuweisungen und Ausdrücke

Zuweisungen und Ausdrücke sollten den „Layout und Stil“ Richtlinien folgen. Dabei ist besonders auf die Verwendung von Zwischenräumen zu achten.

```
a = b + c;
count++;
```

8.1.2 Lokale Variablendeklaration

Lokale Variablen sollten in seperaten Zeilen deklariert werden. Eine Ausnahme besteht für temporäre Variablen die nicht umgehend initialisiert werden.

```
int i, k;
int j = 4;
```

8.1.3 Array Deklarationen

Die eckigen Klammern „[]“ in Array Deklarationen müssen unmittelbar dem Arraytypen folgen. Dazwischen sollte kein Leerzeichen stehen.

```
int[] intArray;
int[] lengthArray = {2, 3};
Employee[] empArray = new Employee[3];
string[] strArray = new string[5] {"Ronnie", "Jack", "Lori", "Max", "Tricky"};

// implicit declaration
string[] errorMessages = {
    "No such file in directory",
    "Unable to open file",
    "Unmatched parantheses in expression"
};

// multi-dimensional array
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };

// jagged array
double[][] jaggedArray = new double[5][];
```

8.1.4 Return Anweisung

Eine Returnwert sollte niemals in runde Klammern gesetzt werden solange kein komplexer Ausdruck zurückgegeben wird. Komplexe Ausdrücke können zusammengesetzte Anweisungen mit mehreren Operatoren sein.

```
return true;

return (s.Length() + s.Offset());
```

8.2 Zusammengesetzte Anweisungen

Zusammengesetzte Anweisungen sind Anweisungen die einen Anweisungsblock enthalten, welcher in geschweifte Klammern „{}“ gesetzt wurde. Alle zusammengesetzten Anweisungen folgen dem bekannten *Kernighan und Ritchie* Stil (K & R), der von den gleichnamigen Personen mit der Sprache „C“ eingeführt wurde. Diese Regel gilt für Klassen, Schnittstellen, und Methodendeklarationen. Der Stil ist folgendermaßen spezifiziert.

1. Die öffnende linke Klammer bildet das letzte Zeichen einer Zeile, in der der Anweisungsblock eröffnet wird.
2. Die rechte schließende Klammer steht allein in einer Zeile und ist auf gleicher Höhe, wie die beginnende Anweisung eingerückt.
3. Anweisungen innerhalb des Blocks werden eine Ebene weiter eingerückt.

In Fällen in denen der Ausdruck eines Anweisungsblock nur aus einem Semikolon besteht kann auf die geschweiften Klammern verzichtet werden.

8.2.1 if-else Verzweigungen

if, if-else und if else-if else Anweisungen sollten folgendermaßen aussehen:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

8.2.2 for- und foreach-Schleifen

```
for (initialization; condition; update) {
    statements;
}
```

8.2.3 while- und do-while-Schleifen

```
while (condition) {
    statements;
}

// infinitive loops should look like this
while (true) {
    statements;
}

do {
    ...
} while (condition);
```

8.2.4 switch-Anweisungen

Eine switch-Anweisung sollte die folgende Form besitzen:

```
switch (condition) {
    case A:
        ...
        break;
```

```
    case B:
        ...
        break;
    case C:
        ...
        break;
    default:
        ...
        break;
}
```

8.2.5 goto-Anweisungen

```
public static void Main()
{
    int i = 0;
repeat:    // label, backintend
    Console.WriteLine("i: {0}", i};
    i++;
    if (i < 10) {
        goto repeat;
    }
    return;
}
```

8.2.6 try-catch-finally Blöcke

```
class Test
{
    static void Fill(object[] array, int index, int count, object value)
    {
        for (int i = index; i < index + count; i++)
            array[i] = value;
    }

    public static void Main()
    {
        try {
            string[] strings = new string[100];
            Fill(strings, 0, 100, "Undefined");
            Fill(strings, 0, 10, null);     // Exception
            Fill(strings, 90, 10, 0);
        } catch (System.ArrayTypeMismatchException) {
            System.Console.WriteLine("Exception Thrown.");
        } finally {
            // do something...
        }
    }
}
```

9 Analysetools

9.1 FxCop

Die Einhaltung der offiziellen *„Microsoft .NET Framework Design Guidelines“* kann mithilfe von geeigneten analytischen Programmen überprüft werden. Eines dieser Tools trägt den Namen *FxCop* und lässt sich von der Seite http://www.gotdotnet.com/team/FxCop/ beziehen. Bei dem Programm handelt es sich um ein Code-Analysis-Tool, welches den verwalteten .NET Code in den Assemblies auf Konformität überprüft. Es verwendet Reflection, syntaktische Analyse der MSIL und eine Callgraph Analyse um die Assemblies zu inspizieren. Das Tool überprüft die folgenden Punkte:

- Design der Bibliotheken
- Lokalisierung
- Namenskonventionen
- Performance
- Sicherheit

FxCop ist ausschließlich für die Analyse von .NET Code vorgesehen und läuft deshalb nur auf Windows! Auf dem Mono Framework basierender Code muss zunächst in .NET Code kompiliert werden, bevor er analysiert werden kann.

10 Quellen

[1] Steve McConnell, „Code Complete, Second Edition (Paperback)“, http://cc2e.com

[2] Peter M. Brown, „Common .NET Naming Conventions“, http://www.irritatedvowel.com/Programming/Standards.aspx

[3] .NET Framework General Reference, „Design Guidelines for Class Library Developers“, http://msdn2.microsoft.com/en-us/default.aspx

[4] Official Code Conventions, „Java Programming Language“, http://java.sun.com/docs/codeconv/

[5] Standard ECMA-334, „C# Language Specification“, http://www.ecma-international.org/publications/standards/Ecma-334.htm

[6] Standard ECMA-335, „ Common Language Infrastructure (CLI)“, http://www.ecma-international.org/publications/standards/Ecma-335.htm

[7] Reddy, A., „C++ Sytle Guide“, Sun Internal Paper

[8] Brian W. Kernighan and Dennis M. Ritchie, „The C Programming Language, Second Edition“, http://cm.bell-labs.com/cm/cs/cbook/

11 Anhang

.1 Beispiele für Namenskonventionen

Element	*Beispiel*	*Beschreibung*
Namensräume	*AppliedIS.TimeCard.BusinessRules* *IrritatedVowel.Controllers* *PeteBrown.DotNetTraining.XmlDemo* *PeteBrown.DotNetTraining.Xml*	Pascal Case, keine Unterstriche. Verwenden Sie CompanyName.TechnologyName als Stammverzeichnis. Falls das Projekt keinem Unternehmen angehört können Sie einen Domainnamen oder Ihre Initialen verwenden. Alle Akronyme mit drei oder mehr Buchstaben sollten in Pascal Case (z.B. Xml) statt in Großbuchstaben geschrieben werden.
Assemblies	*AppliedIS.TimeCard.BusinessRules.dll* *IrritatedVowel.Controllers.dll*	Falls das Assembly einen einzigen Namensraum enthält oder einen eigenen Stammnamensraum hat, sollte das Assembly nach dem Namen des Namensraumes benannt werden.
Klassen und Strukturen	*Widget* *InstanceManager* *XmlDocument* *MainForm* *DocumentForm* *HeaderControl* *CustomerListDataSet (typed dataset)*	Pascal Case, keine Unterstriche oder Verwendung von „C“ bzw. „cls“ als Postfix. Klassen dürfen mit einem „I“ beginnen sofern der nachfolgende Buchstabe klein geschrieben wird. Verwechslungen mit Interfaces werden auf diese Weise vermieden. Klassen sollten nicht denselben Namen wie Namensräume haben in denen sie definiert sind. Alle Akronyme mit drei oder mehr Buchstaben sollten in Pascal Case geschrieben sein. Versuchen Sie Abkürzungen zu vermeiden und Nomen zu verwenden.

Collection Klassen	*WidgetCollection*	Folgen Sie den Namenskonventionen für Klassen, aber fügen Sie Collection als Suffix an den Namen.
Delegate Klassen	*WidgetCallbackDelegate*	Folgen Sie den Namenskonventionen für Klassen, aber fügen Sie Delegate als Suffix an den Namen.
Exception Klassen	*InvalidTransactionException*	Folgen Sie den Namenskonventionen für Klassen, aber fügen Sie Exception als Suffix an den Namen.
Attribute Klassen	*WebServiceAttribute*	Folgen Sie den Namenskonventionen für Klassen, aber fügen Sie Attribute als Suffix an den Namen.
Interfaces	*IWidget*	Folgen Sie den Namenskonventionen für Klassen, aber beginnen Sie den Namen mit einem „I“ mit nachfolgendem Großbuchstaben.
Enumerationen	*SearchOptions (bitwise flags) AcceptRejectRule (normal enum)*	Folgen Sie den Namenskonventionen für Klassen. Fügen Sie nicht „Enum“ oder dergleichen an das Ende des Namens. Falls ein Aufzählungstyp einen Satz von bitweisen Flags repräsentiert sollte der Name mit einem Plural enden.
Funktionen und Methoden	*public void DoSomething(...)*	Pascal Case, keine Unterstriche außer in Event Handlern. Versuchen Sie Abkürzungen zu vermeiden. Viele Programmierer neigen zu übermäßigem Gebrauch von Abkürzungen. Machen Sie sich bewußt das Ihr Code viel öfters gelesen als geschrieben wird. Die Namen sollten sich nicht nur durch ihre Buchstabengröße unterscheiden um die Kompatibilität mit nicht case-sensitiven Sprachen zu gewährleisten.
Eigenschaften und öffentliche Klassenelemente	*public int RecordID*	Pascal Case, keine Unterstriche. Versuchen Sie Abkürzungen zu vermeiden. Die Namen sollten sich nicht nur durch ihre Buchstabengröße unterscheiden um die Kompatibilität mit nicht case-sensitiven Sprachen zu gewährleisten.

Parameter	*ref int recordID*	Camel Case. Versuchen Sie Abkürzungen zu vermeiden. Die Namen sollten sich nicht nur durch ihre Buchstabengröße unterscheiden um die Kompatibilität mit nicht case-sensitiven Sprachen zu gewährleisten.
Prozedur-Level Variablen	*int recordID ;*	Camel Case.
Private und geschützte Variablen in Klassen	*private int _recordID;* *oder* *protected int mRecordID;*	In der Regel werden diese Klassenelemente mit dem Präfix m_ gekennzeichnet. In den letzten Jahren hat sich jedoch vermehrt nur ein führender Unterstrich durchgesetzt. Microsoft empfiehlt weder m_ noch _ als Präfix. Da C# eine case-sensitive Sprache ist, kann hier auch eine Unterscheidung nur anhand der Buchstabengröße erfolgen.
Controls auf Forms	*txtUserID, lblHeader, lstChoices, btnSubmit or „ux“ prefix (preferred!) uxUserIDText, uxHeaderLabel, uxChoiceList, uxSubmitButton*	Hier wird entweder der normale .NET Klassenname oder „control“ bzw. „ui“ als Präfix verwendet. Dieses Verfahren hält die Controls in IntelliSense zusammen und erleichtert die UI Programmierung.
Konstanten	*SomeClass.MYCONSTANT* *CONSTANT*	Bennante Konstanten werden vollständig in Großbuchstaben geschrieben.

.2 C# Quellcode Beispiel A

```
using System;
using System.IO;

namespace MyProject
{
    /// <summary>Demonstrates some of the main members of the File class.</summary>
    class Test
    {
        /// <summary>Main entry point.</summary>
        /// <param name="args"></param>
        public static void Main(string[] args)
        {
            string path = @"MyTest.txt";
            if (!File.Exists(path)) {
                // Create a file to write to.
                using (StreamWriter sw = File.CreateText(path)) {
                    sw.WriteLine("Hello");
                    sw.WriteLine("And");
                    sw.WriteLine("Welcome");
                }
            }

            // Open the file to read from.
            using (StreamReader sr = File.OpenText(path)) {
                string s = "";
                while ((s = sr.ReadLine()) != null) {
                    Console.WriteLine(s);
                }
            }

            try {
                string path2 = path + "temp";
                // Ensure that the target does not exist.
                File.Delete(path2);

                // Copy the file.
                File.Copy(path, path2);
                Console.WriteLine("{0} was copied to {1}.", path, path2);

                // Delete the newly created file.
                File.Delete(path2);
                Console.WriteLine("{0} was successfully deleted.", path2);
            } catch (Exception e) {
                Console.WriteLine("The process failed: {0}", e.ToString());
            }
        }
    }
}
```

.3 C# Quellcode Beispiel B

```
using System;
using System.Collections;
using System.Collections.Generic;

/// <summary>Demonstrates a generic linklist.</summary>
class Node<K, T>
{
    public Node()
    {
        Key = default(K);
        Item = default(T);
        NextNode = null;
    }

    public Node(K key, T item, Node<K, T> nextNode)
    {
        Key = key;
        Item = item;
        NextNode = nextNode;
    }

    public K Key;
    public T Item;
    public Node<K, T> NextNode;
}


public class LinkedList<K, T> : IEnumerable<T> where K : IComparable<K>
{
    Node<K, T> m_Head;

    public LinkedList()
    {
        m_Head = new Node<K, T>();
    }

    public void AddHead(K key, T item)
    {
        Node<K, T> newNode = new Node<K, T>(key, item, m_Head.NextNode);
        m_Head.NextNode = newNode;
    }

    public T this[K key]
    {
        get
        {
            return Find(key);
        }
    }

    T Find(K key)
    {
        Node<K, T> current = m_Head;

        while (current.NextNode != null) {
```

```
            if (current.Key.Equals(key)) {
                break;
            } else {
                current = current.NextNode;
            }
        }
        return current.Item;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node<K, T> current = m_Head;
        while (current != null) {
            yield return current.Item;
            current = current.NextNode;
        }
    }

    public static LinkedList<K, T> operator +(LinkedList<K, T> lhs, LinkedList<K, T> rhs)
    {
        return concatenate(lhs, rhs);
    }

    static LinkedList<K, T> concatenate(LinkedList<K, T> list1, LinkedList<K, T> list2)
    {
        LinkedList<K, T> newList = new LinkedList<K, T>();
        Node<K, T> current;

        current = list1.m_Head;
        while (current != null) {
            newList.AddHead(current.Key, current.Item);
            current = current.NextNode;
        }

        current = list2.m_Head;

        while (current != null) {
            newList.AddHead(current.Key, current.Item);
            current = current.NextNode;
        }
        return newList;
    }
}
```