CoDeSys Automation Platform
Codierrichtlinien

Dokument Version 1.3

INHALT

1 Einleitung

Dieses Dokument stellt Codierrichtlinien vor, die innerhalb der Firma 3S-Smart Software Solutions GmbH für Entwicklungen, die auf dem Microsoft .NET-Framework beruhen, für alle Entwickler verbindlich gelten. Abweichungen von diesen Regeln sind nur in gut begründeten Ausnahmefällen gestattet.

Als Vorlage für dieses Codierrichtlinien diente das Dokument *.NET Framework General Reference – Design Guidelines for Class Library Developers* [1]. Es ist in großen Teilen einfach ins Deutsche übersetzt worden. Die wesentlichsten Unterschiede können wie folgt zusammengefasst werden:

- Einige Regeln sind auf die Architektur der *CoDeSys Automation Platform*, eines der Hauptprodukte von 3S, abgestimmt worden. Beispielsweise der Vergleich, wann Schnittstellen und wann abstrakte Basisklassen zum Einsatz kommen sollen, fällt hier aus prinizipiellen Gründen zugunsten der Schnittstelle aus.
- Richtlinien zur Benennung von Bezeichnern orientieren sich an traditionellen Gesichtspunkten. Obwohl beispielsweise die ungarische Notation heute nicht mehr empfohlen wird, ist sie dennoch für diese Richtlinien übernommen worden, da sie zum einen firmenintern weitgehend akzeptiert sind und Entwicklern den Wechsel zwischen Win32- und .NET-Programmierung und umgekehrt deutlich erleichern.

2 Compilerfehler und Compilerwarnungen

- Visual Studio-Projekte werden mit Warnungsstufe 4 übersetzt.
- Es darf kein Quellcode in die Quellcodeverwaltung committed werden, bei dem Compilerfehler oder Compilerwarnungen auftreten. Jedes Projekt muss also mit 0 Fehlern und 0 Warnungen übersetzbar sein. Eine Ausnahme bilden Warnungen, die bei Aktivierung der XML-Dokumentationsdatei durch falsche Kommentarkonstrukte auftreten.
- Zur Umgehung von Compiler-Warnungen gilt folgendes:
  - Grundsätzlich ist die Ursache der Warnung zu beheben.
  - Ist dies nicht möglich, weil das Änderungsrisiko zu hoch eingeschätzt wird (dies kann insbesondere der Fall sein, wenn verwendete Symbole in einer neuen Version als *obsolet* markiert worden sind, die Auswirkungen einer Anpassung aber nicht eingeschätzt werden können), dann ist an dieser Stelle eine #pragma warning-Klammer einzusetzen, und zwar so eng wie möglich um die gemeldete Warnungsposition. Außerdem muss ein Kommentar mit ausreichender Erklärung hinzugefügt werden.

    ```
    #pragma warning disable 618   // Methode ist obsolet, aber…
    CallToObsoleteMethod();
    #pragma warning restore 618
    ```

    Darüber hinaus ist ein Eintrag in die Fehlerdatenbank zu machen, um zu einem späteren Zeitpunkt die Warnung ursächlich beheben zu können.
  - Ist auch dies nicht möglich, weil eine Warnung in Quellcode auftritt, der durch ein Tool generiert wurde (z.B. dem Visual Studio Designer), dann darf diese Warnung ausnahmsweise global in den Projekteinstellungen deaktiviert werden.

3 Formatierung

3.1 Editoreinstellungen

Für die Konfiguration eines Quellcode-Editors gelten folgende Einstellungen:

Tab-Breite (Tab size):	4
Einrückungs-Breite (Indent size):	4
Tabs beibehalten (Keep tabs):	ja

3.2 Einrückungen

Regel	Positivbeispiel	Negativbeispiel

Blöcke einrücken	`class MyClass`
	`{`
	`  int Method()`
	`  {`
	`  return 3;`
	`  }`
	`}`
		`class MyClass`
		`{`
		`  int Method()`
		`  {`
		`  return 3;`
		`  }`
		`}`

Öffnende und schließende Klammern nicht einrücken	`class MyClass`
	`{`
	`  int Method()`
	`  {`
	`  return 3;`
	`  }`
	`}`
		`class MyClass`
		`  {`
		`  int Method()`
		`  {`
		`  return 3;`
		`  }`
		`  }`

`case`-Blöcke einrücken	`switch (name)`
	`{`
	`  case "John":`
	`  nMyVar = 3;`
	`  break;`
	`}`
		`switch (name)`
		`{`
		`  case "John":`
		`  nMyVar = 3;`
		`  break;`
		`}`

`case`-Marken einrücken	`switch (name)`
	`{`
	`  case "John":`
	`  break;`
	`}`
		`switch (name)`
		`{`
		`case "John":`
		`  break;`
		`}`

`break`-Statements nicht ausrücken	`switch (name)`
	`{`
	`  case "John":`
	`  nMyVar = 3;`
	`  break;`
	`}`
		`switch (name)`
		`{`
		`  case "John":`
		`  nMyVar = 3;`
		`  break;`
		`}`

Sprungmarken eine Einrückungsebene kleiner einrücken als restlicher Block	`class MyClass`
	`{`
	`  public void Method()`
	`  {`
	`  goto MyLabel;`
	`  MyLabel:`
	`  return;`
	`  }`
	`}`
		`class MyClass`
		`{`
		`  public void Method()`
		`  {`
		`  goto MyLabel;`
		`MyLabel:`
		`  return;`
		`  }`
		`}`
		oder
		`class MyClass`
		`{`
		`  public void Method()`
		`  {`
		`  goto MyLabel;`
		`  MyLabel:`
		`  return;`
		`  }`
		`}`

3.3 Zeilenumbrüche

Regel	Positivbeispiel	Negativbeispiel

Öffnende geschweifte Klammer in eine neue Zeile setzen für Typen	`class MyClass`
	`{`
	`  // …`
	`}`
		`class MyClass {`
		`  // …`
		`}`

Öffnende geschweifte Klammer in eine neue Zeile setzen für Methoden	`class MyClass`
	`{`
	`  int Method()`
	`  {`
	`  return 3;`
	`  }`
	`}`
		`class MyClass`
		`{`
		`  int Method() {`
		`  return 3;`
		`  }`
		`}`

Öffnende geschweifte Klammer in eine neue Zeile setzen für anonyme Methoden	`timer.Tick += delegate`
	`(object sender, EventArgs e)`
	`{`
	`  MessageBox.Show("…");`
	`}`
		`timer.Tick += delegate`
		`(object sender, EventArgs e){`
		`  MessageBox.Show("…");`
		`}`

Öffnende geschweifte Klammer in eine neue Zeile setzen für Kontrollblöcke	`int Method()`
	`{`
	`  if (a > b)`
	`  {`
	`  return 0;`
	`  }`
	`}`
		`int Method()`
		`{`
		`  if (a > b) {`
		`  return 0;`
		`  }`
		`}`

Schlüsselwort `else` in eine neue Zeile setzen	`if (a > b)`
	`{`
	`  return 3;`
	`}`
	`else`
	`{`
	`  return 0;`
	`}`
		`if (a > b)`
		`{`
		`  return 3;`
		`} else`
		`{`
		`  return 0;`
		`}`

Schlüsselwort `catch` in eine neue Zeile setzen	`try`
	`{`
	`  // …`
	`}`
	`catch (Exception ex)`
	`{`
	`  // …`
	`}`
		`try`
		`{`
		`  // …`
		`} catch (Exception ex)`
		`{`
		`  // …`
		`}`

Schlüsselwort `finally` in eine neue Zeile setzen	`try`
	`{`
	`  // …`
	`}`
	`finally`
	`{`
	`  // …`
	`}`
		`try`
		`{`
		`  // …`
		`} finally`
		`{`
		`  // …`
		`}`

Zeilenumbruch nach einem Semikolon einfügen	`i = 0;` `stName = "John";`	`i = 0; stName = "John";`

3.4 Leerzeichen

3.4.1 Methodendeklarationen

Regel	Positivbeispiel	Negativbeispiel

Kein Leerzeichen zwischen Methodenname und öffnender Klammer einfügen	`int Method()`
	`{`
	`  return 3;`
	`}`
		`int Method ()`
		`{`
		`  return 3;`
		`}`

Kein Leerzeichen zwischen öffnender Klammer und Argumentliste einfügen	`int Method(int input)`
	`{`
	`  return input;`
	`}`
		`int Method( int input)`
		`{`
		`  return input;`
		`}`

Regel	Positivbeispiel	Negativbeispiel

Kein Leerzeichen zwischen Klammern einfügen, wenn die Argumentliste leer ist	`int Method()`
	`{`
	`  return 3;`
	`}`
		`int Method( )`
		`{`
		`  return 3;`
		`}`

3.4.2 Methodenaufrufe

Regel	Positivbeispiel	Negativbeispiel
Kein Leerzeichen zwischen Methodenname und öffnender Klammer einfügen	`Console.Write("Hallo");`	`Console.Write ("Hallo");`
Kein Leerzeichen zwischen öffnender Klammer und Argumentliste einfügen	`Console.Write("Hallo");`	`Console.Write( "Hallo");`
Kein Leerzeichen zwischen Klammern einfügen, wenn die Argumentliste leer ist	`dialog.Show();`	`dialog.Show( );`

3.4.3 Eckige Klammern

Regel	Positivbeispiel	Negativbeispiel
Kein Leerzeichen vor einer öffnenden eckigen Klammer einfügen	`args = new string[10];`	`args = new string [10];`
Kein Leerzeichen zwischen leeren eckigen Klammern einfügen	`string[] args;`	`string[ ] args;`
Kein Leerzeichen zwischen nicht-leeren eckigen Klammern einfügen	`args = new string[10];`	`args = new string[ 10 ];`

3.4.4 Begrenzungszeichen

Regel	Positivbeispiel	Negativbeispiel

Leerzeichen vor dem Doppelpunkt für Basisklasse oder Schnittstelle in einer Typdeklaration einfügen	`class MyClass : IDisposable`
	`{`
	`  // …`
	`}`
		`class MyClass: IDisposable`
		`{`
		`  // …`
		`}`

Leerzeichen nach dem Doppelpunkt für Basisklasse oder Schnittstelle in einer Typdeklaration einfügen	`class MyClass : IDisposable`
	`{`
	`  // …`
	`}`
		`class MyClass :IDisposable`
		`{`
		`  // …`
		`}`

Kein Leerzeichen vor einem Komma einfügen	`int Sum(int a, int b)`
	`{`
	`  return a + b;`
	`}`
		`int Sum(int a ,int b)`
		`{`
		`  return a + b;`
		`}`

Leerzeichen nach einem Komma einfügen	`int Sum(int a, int b)`
	`{`
	`  return a + b;`
	`}`
		`int Sum(int a,int b)`
		`{`
		`  return a + b;`
		`}`

Kein Leerzeichen vor einem Punkt einfügen	`Console.Write("Hallo");`	`Console .Write("Hallo");`
Kein Leerzeichen nach einem Punkt einfügen	`Console.Write("Hallo");`	`Console. Write("Hallo");`

Regel	Positivbeispiel	Negativbeispiel

Kein Leerzeichen vor einem Semikolon in einer `for`-Anweisung einfügen	`for (int i = 0; i < 3; i++)`
	`{`
	`  // …`
	`}`
		`for (int i = 0 ; i < 3 ; i++)`
		`{`
		`  // …`
		`}`

Leerzeichen nach einem Semikolon in einer `for`-Anweisung einfügen	`for (int i = 0; i < 3; i++)`
	`{`
	`  // …`
	`}`
		`for (int i = 0;i < 3;i++)`
		`{`
		`  // …`
		`}`

3.4.5 Operatoren

Regel	Positivbeispiel	Negativbeispiel
Leerzeichen vor und nach einem binären Operator einfügen	`result = 1 + 2 * 3;`	`result = 1+2*3;`

3.4.6 Sonstige Leerzeichenregeln

Regel	Positivbeispiel	Negativbeispiel
Leerzeichen nach Schlüsselwort in Kontrollflussanweisungen einfügen	`if (a > b)` `  return 0;`	`if(a > b)` `  return 0;`
Kein Leerzeichen innerhalb der Klammern von Ausdrücken einfügen	`return a * (b - a);`	`return a * ( b - a );`
Kein Leerzeichen innerhalb der Klammern von Typumwandlungen einfügen	`int number = (int)list[0];`	`int number = ( int )list[0];`
Kein Leerzeichen innerhalb der Klammern von Kontrollflussanweisungen einfügen	`if (a > b)` `  return 0;`	`if ( a > b )` `  return 0;`
Kein Leerzeichen nach einer Typumwandlung einfügen	`int number = (int)list[0];`	`int number = (int) list[0];`

3.5 Kommentare

3.5.1 Kommentare in der Implementierung

Implementierungen sind im Quellcode in einem vernünftigen Maß zu kommentieren, so dass jeder eingewiesene Entwickler eine beliebige Codestelle selbständig verstehen kann.

3.5.2 Kommentare in der Schnittstelle

Schnittstellenbeschreibungen sind mit C#-Dokumentationskommentaren zu versehen. Aus diesen Kommentaren kann vom C#-Compiler eine XML-Dokumentationsdatei generiert werden, die mit weiteren Tools in verschiedene Formate umgewandelt werden kann (CHM, HTML, PDF), die dann wiederum Kunden weitergegeben werden können. Dabei sind die Richtlinien zur Erstellung von C#-Dokumentationskommentaren zu beachten.

Wichtig ist, dass Auflistungen nicht einfach durch Zeilenumbrüche formuliert werden, da diese nicht in die Ausgabe übernommen werden. Stattdessen muss das <list>-Tag verwendet werden, wie es im nachfolgenden Beispiel zu sehen ist:

```
<list type="bullet" | "number" | "table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>term</term>
        <description>description</description>
    </item>
</list>
```

3.5.3 Header-Kommentare

Header-Kommentare, also Kommentare, die am Beginn einer Quellcode-Datei stehen und eine generelle Übersicht bieten sollen, sind grundsätzlich erlaubt. Jedoch sollen keine Schlüsselwörter verwendet werden, die von der Quellcodeverwaltung automatisch ersetzt werden (z.B. `$Date$`, `$Revision$` oder `$Author$`), da sie bei der Quellcode-Integration zu unnötigen Konflikten führen.

4 Namenskonventionen

Konsistente Namenskonventionen gehören zu den wichtigsten Grundsteinen in Bezug auf Vorhersagbarkeit und Wartbarkeit von Code. Eine durchgängige Verwendung dieser Konventionen führt dazu, dass viele Unklarheiten bei der Betrachtung von Code von vornherein ausgeschlossen werden. Dieses Kapitel beschreibt die Namenskonventionen für Klassen innerhalb der *CoDeSys Automation Platform*. Zu jeder Namensart werden Regeln bezüglich Notation, Groß-/Kleinschreibung und Wortwahl aufgestellt.

4.1 Notationen

4.1.1 Pascal-Notation

Der erste Buchstabe eines Bezeichers sowie der erste Buchstabe eines jeden angehängten Wortes werden groß geschrieben.

Beispiel:

```
BackColor
```

Falls ein Wort innerhalb eines Bezeichers eine aus zwei Zeichen bestehende Abkürzung ist, wird diese in Großbuchstaben geschrieben.

Beispiel:

```
System.IO
```
(aber: `GuiHelper`)

4.1.2 Großschreibungsnotation

Alle Buchstaben eines Bezeichners sind groß geschrieben. Einzelne Wörter innerhalb des Bezeichners werden durch Unterstriche voneinander getrennt.

Beispiel:

```
WM_SIZE
MAXIMUM_COUNT
```

4.1.3 Ungarische Notation

Die ungarische Notation ist nach dem Erfinder der Notation, Charles Simonyi, der ein Ungar ist, benannt. Es gibt viele verschiedene Interpretationen dieser Notation, deshalb soll hier die angegeben werden, die sich firmenintern etabliert hat.

Hierbei wird einem Bezeichner ein oder mehrere Präfixe vorangestellt, die über den Datentyp der Variablen Auskunft geben sollen. Der Teil nach dem Präfix ist gemäß der Pascal-Notation formatiert. Unterstriche sind grundsätzlich zu unterlassen, außer gemäß den Regeln für Feldnamen (siehe Kap. 4.11).

Die nachfolgende Tabelle listet die gültigen Präfixe auf:

Präfix	Datentyp	Beispiel
b	System.Boolean	bSuccess
c oder ch	System.Character	cNext
st	System.String	stName
sb	System.StringBuilder	sbErrorMessage
by	System.Byte	byWritten
n	beliebiger ganzzahliger Datentyp	nCount
s	System.Int16	sValue

Präfix	Datentyp	Beispiel
us	System.UInt16	usValue
i	System.Int32	iValue
ui	System.UInt32	uiValue
l	System.Int64	lValue
ul	System.UInt64	ulValue
f	System.Single	fReal
d	System.Double	dImag
a	System.Array (optional, oft in Verbindung mit weiteren Präfixen)	astNames
al	System.ArrayList (optional)	alObjects
ht	System.Hashtable (optional)	htRelationships

4.2 Übersicht über Notationsregeln

Die nachfolgende Tabelle fasst die Großschreibungsregeln für die wichtigsten Bezeichnerarten zusammen.

Bezeichnerart	Notationsregel	Beispiel
Allgemeine Klasse	Pascal-Notation	AppDomain
Enumerationstyp	Pascal-Notation	ErrorLevel
Enumerationswert	Pascal-Notation	FatalError
Ereignis	Pascal-Notation	ValueChanged
Delegat	Pascal-Notation	ClosedEventHandler
Ausnahmeklasse	Pascal-Notation	WebException (Beachte: Endet stets mit dem Suffix Exception)
Attributklasse	Pascal-Notation	TypeGuidAttribute (Beachte: Endet stets mit dem Suffix Attribute)
Statisches readonly-Feld	Großschreibungsnotation	MAXIMUM_COUNT
Konstantes Feld	Großschreibungsnotation	MAXIMUM_COUNT
Schnittstelle	Pascal-Notation	IDisposable (Beachte: Beginnt stets mit dem Präfix I)
Methode	Pascal-Notation	ToString
Namensraum	Pascal-Notation	System.Windows.Forms
Parameter	Ungarische Notation	stName
Eigenschaft	Pascal-Notation	BackColor
Geschütztes oder privates Instanzfeld	Ungarische Notation	_stName (Beachte: Beginnt stets mit dem Präfix _ oder m_)
Öffentliches Instanzfeld	Pascal-Notation	BackColor

Bezeichnerart	Notationsregel	Beispiel
Geschütztes oder privates statisches Feld	Ungarische Notation	s_nClassCount (Beachte: Beginnt stets mit dem Präfix s_ oder ms_)
Öffentliches statisches Feld	Pascal-Notation	ClassCount
Lokale Variable	Ungarische Notation	nTemp

4.3 Groß-/Kleinschreibung

Um Verwechslungen zu vermeiden und um die Interoperabilität zwischen verschiedenen Programmiersprachen sicherzustellen, sind folgende Regeln bezüglich Groß-/Kleinschreibung einzuhalten:

- Komponenten müssen sowohl für Programmiersprachen verwendbar sein, für die Groß-/Kleinschreibung relevant ist, als auch für solche, in denen dies keine Rolle spielt. Letztere können zwei Namen nicht unterscheiden, die sich ausschließlich in ihrer Groß-/Kleinschreibung unterscheiden. Daher sind solche Situationen unbedingt zu vermeiden.
- Erstelle keine zwei Namensräume, deren Name sich nur durch Groß-/Kleinschreibung unterscheiden. Beispielsweise können manche Programmiersprachen nicht zwischen folgenden beiden Deklarationen unterscheiden:

```
namespace ee.cummings;
namespace Ee.Cummings;
```

- Erstelle keine Methode, in der sich zwei Parameternamen nur durch Groß-/Kleinschreibung unterschieden. Beispielsweise lässt sich folgendes Beispiel in manchen Programmiersprachen nicht übersetzen:

```
void MyMethod(string a, string A)
```

- Erstelle innerhalb eines Namensraums keine zwei Typen, die sich nur durch Groß-/Kleinschreibung unterscheiden. Folgendes Beispiel ist auf jeden Fall zu vermeiden:

```
namespace Foo
{
   public class POINT {}
   public class Point {}
}
```

- Erstelle innerhalb eines Typs keine zwei Member, der sich nur durch Groß-/Kleinschreibung unterscheiden. Folgendes Beispiel ist auf jeden Fall zu vermeiden:

```
public class GraphicObject
{
   public int Color { get { … } set { … } }
   public int color { get { … } set { … } }
}
```

4.4 Abkürzungen

Um Verwirrungen zu vermeiden, sind folgende Regeln bezüglich Abkürzungen einzuhalten:

- Verwende keine Abkürzungen von Wortbestandteilen innerhalb von Bezeichnern. Beispielsweise soll GetWindow anstelle von GetWin verwendet werden.
- Verwende keine Akronyme, die nicht in der Informatik oder der Automatisierungsbranche allgemein anerkannt sind.
- Wenn möglich, können wohlbekannte Akronyme verwendet werden, wenn der Bezeichner ansonsten unnötig lang würde. Beispiele hierfür sind UI für User Interface oder POU für Program Organization Unit.

- Akronyme, die zwei Buchstaben lang sind, werden in Großbuchstaben formuliert. Längere Akronyme werden in Pascal-Notation formuliert. Beispiele: `System.IO`, `GuiHelper`.

4.5 Wortwahl

Vermeide die Verwendung von Klassennamen, die bereits innerhalb des .NET-Frameworks verwendet wurden. Verzichte beispielsweise auf Namen wie `System`, `Collections`, `Forms` oder `UI`.

Des weiteren dürfen keine Bezeichner aus der folgenden Liste verwendet werden, da sie in manchen Programmiersprachen (insbesondere Visual Basic.NET) reserviert sind.

AddHandler	AddressOf	Alias	And	Ansi
As	Assembly	Auto	Base	Boolean
ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate
CDec	CDbl	Char	CInt	Class
CLng	CObj	Const	CShort	CSng
CStr	CType	Date	Decimal	Declare
Default	Delegate	Dim	Do	Double
Each	Else	ElseIf	End	Enum
Erase	Error	Event	Exit	ExternalSource
False	Finalize	Finally	Float	For
Friend	Function	Get	GetType	Goto
Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let
Lib	Like	Long	Loop	Me
Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object	On
Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly	ReDim
Region	REM	RemoveHandler	Resume	Return
Select	Set	Shadows	Shared	Short
Single	Static	Step	Stop	String
Structure	Sub	SyncLock	Then	Throw
To	True	Try	TypeOf	Unicode
Until	Volatile	When	While	With
WithEvents	WriteOnly	Xor	eval	extends
instanceof	package	var		

4.6 Richtlinien bei der Verwendung von Basisdatentypen

Unterschiedliche Programmiersprachen haben auch unterschiedliche Bezeichner für die grundlegenden Basisdatentypen. Bei der Entwicklung von Softwarekomponenten muss auf jeden Fall vermieden werden, solche sprachspezifischen Bezeichner zu verwenden.

Für Parameter gilt:

Verwende Bezeichner, die die semantische Bedeutung eines Parameters beschreiben und nicht dessen Datentyp. In den seltenen Fällen, in denen ein Parameter keine weitere semantische Bedeutung hat, soll ein generischer Name verwendet werden. Als Beispiel sei hier eine Klasse aufgeführt, die eine Reihe von Datentypen in einen Stream schreiben kann.

Positivbeispiel	**Negativbeispiel**

`void Write(double value);`
`void Write(float value);`
`void Write(long value);`
`void Write(int value);`
`void Write(short value);`
	`void Write(double doubleValue);`
	`void Write(float floatValue);`
	`void Write(long longValue);`
	`void Write(int intValue);`
	`void Write(short shortValue);`

Für Methoden gilt:

In den sehr seltenen Fällen, in denen es nötig ist, den Typnamen in einen Methodenbezeichner aufzunehmen, muss der universelle Typname verwendet werden, nicht der sprachspezifische. Die folgende Tabelle enthält den universellen Typnamen für jeden eingebauten C#-Datentyp.

C#-Typname	**Universeller Typname**
`sbyte`	`SByte`
`byte`	`Byte`
`short`	`Int16`
`ushort`	`UInt16`
`int`	`Int32`
`uint`	`UInt32`
`long`	`Int64`
`ulong`	`UInt64`
`float`	`Single`
`double`	`Double`
`bool`	`Boolean`
`char`	`Char`
`string`	`String`
`object`	`Object`

Beispielsweise könnte eine Klasse, die verschiedene Datentypen aus einem Stream lesen kann, folgende Methoden haben:

Positivbeispiel	**Negativbeispiel**

`double ReadDouble();`
`float ReadSingle();`
`long ReadInt64();`
`int ReadInt32();`
`short ReadInt16();`
	`double ReadDouble();`
	`float ReadFloat();`
	`long ReadLong();`
	`int ReadInt();`
	`short ReadShort();`

4.7 Namesräume

Die allgemeine Regel bei der Benennung von Namensräumen lautet: Firmenname, gefolgt vom Produktnamen, gefolgt vom Komponentennamen und optional gefolgt vom Namen eines Komponentenbestandteils.

```
CompanyName.ProductName.ComponentName[.ComponentPart]
```

Im Rahmen der CoDeSys Automation Platform lautet der Firmenname `_3S` und der Produktname `CoDeSys`. Beispielsweise sind also folgende Namensraumbezeichner möglich:

```
_3S.CoDeSys.POUEditor
_3S.CoDeSys.Core.Objects
```

Firmen- und Produktname werden deshalb in den Namensraum aufgenommen, um zu vermeiden, dass zwei Komponenten von unterschiedlichen Herstellern denselben Namensraum definieren.

Ein Namensraum, der Design-Time-Funktionalität zur Verfügung stellt, ist mit dem Suffix `.Design` zu versehen. So enthält der Namensraum `System.Windows.Forms.Design` Designer-Klassen und dazugehörige Hilfsklassen, die zur Entwicklung von Applikationen mit `System.Windows.Forms` verwendet werden.

Ein geschachtelter Namensraum sollte von Typen im enthaltenden Namensraum abhängig sein, aber nicht umgekehrt. So verwenden Klassen in `System.Web.UI.Design` solche aus `System.Web.UI`, aber keine Klasse in `System.Web.UI` ist von einer Klasse in `System.Web.UI.Design` abhängig.

Namensraumbezeichner sollten in Pascal-Notation vorliegen. Eine Ausnahme von dieser Regel ist dann zulässig, wenn ein Firmen- oder Produktname über eine ungewöhnliche Groß-/Kleinschreibung verfügt, wie z.B. `NeXT`, `CoDeSys`.

Sofern zutreffend sind Namensräume im Plural zu benennen. Verwende beispielsweise `System.Collections` statt `System.Collection`. Ausnahmen von dieser Regel sind Firmen- und Produktnamen oder Abkürzungen wie `IO`.

Für einen Namensraum und ein darin enthaltener Typ darf nicht derselbe Name verwendet werden. Beispielsweise soll innerhalb des Namensraums `Debug` keine Klasse `Debug` definiert werden.

4.8 Klassennamen

Bezeichner für Klassen müssen folgenden Regeln genügen:

- Verwende ein Substantiv, um eine Klasse zu benennen. Dieses Substantiv kann zusammengesetzt sein und gegebenenfalls mit einem Adjektiv versehen sein. Zulässige Beispiele sind: `Stream`, `FileStream`, `UnknownObject`. Unzulässige Beispiele sind: `Operate`, `Calculate`, `PerformAction`.
- Verwende Pascal-Notation.
- Setze Abkürzungen sparsam ein.
- Verwende kein Präfix (wie `C` für Klasse) im Klassennamen. Beispielsweise ist `FileStream` statt `CFileStream` zu verwenden.
- Verwende keinen Unterstrich (_) im Klassennamen.
- Ein Klassenname darf mit einem `I` beginnen (obwohl es sich nicht um eine Schnittstelle handelt), sofern `I` der erste Buchstabe eines ganzen Worts innerhalb des Klassennamens ist. Der Klassenname `IdentityStore` ist beispielsweise zulässig.
- Wenn möglich und sinnvoll, verwende bei abgeleiteten Klassen einen zusammengesetzten Namen, der am Ende den Namen der Basisklasse hat. Beispielsweise sollte eine von `Stream` abgeleitete Klasse vorzugsweise `FileStream` statt nur `File` heißen, da `File` nicht notwendigerweise auf einen Stream schließen lässt. Hingegen ist es vollkommen ausreichend, eine von `Control` abgeleitete Klasse `Button` zu nennen statt `ButtonControl`, da es klar ist, dass ein Button ein Control ist.
  Diese Regel soll also mit Verstand eingesetzt werden.
- Ausnahmeklassen müssen das Suffix `Exception` haben, z.B. `InvalidOperationException`. Nur auf diese Weise ist es möglich, dem Klassennamen auf den ersten Blick anzusehen, dass es sich um eine Ausnahmeklasse handelt.
- Attributklassen müssen das Suffix `Attribute` haben, z.B. `TypeGuidAttribute`. Diese Regel gilt, obwohl es für den C#-Compiler unerheblich ist, ob diese Klasse mit `[TypeGuid]` oder `[TypeGuidAttribute]` verwendet wird. Nur auf diese Weise ist es möglich, dem Klassennamen auf den ersten Blick anzusehen, dass es sich um eine Attributklasse handelt.

- Abstrakte Klassen sollen das Präfix `Abstract` haben, z.B. `AbstractTreeTableModel`. Nur auf diese Weise ist es möglich, der Klasse auf den ersten Blick anzusehen, dass sie nicht instanzierbar ist und über Spezialisierungen verfügt.

4.9 Schnittstellennamen

Bezeichner für Schnittstellen müssen folgenden Regeln genügen:

- Verwende ein Substantiv, ein Adjektiv oder eine Besitzanzeige, um eine Schnittstelle zu benennen. Das Substantiv kann zusammengesetzt sein und gegebenenfalls mit einem Adjektiv versehen sein. Zulässige Beispiele sind: `IComponent`, `IComparable`, `IHasOnlineMode`, `ICustomAttributeProvider`, `IPersistableElement`. Unzulässige Beispiele sind: `IOperate`, `ICalculate`, `IPerformAction`.
- Verwende Pascal-Notation.
- Setze Abkürzungen spärlich ein.
- Verwende das Präfix `I`, um den Typ als Schnittstelle zu kennzeichnen.
- Falls es eine Standardimplementierung zu einer Schnittstelle gibt, sollte sie denselben Namen haben wie die Schnittstelle, jedoch ohne das Präfix `I`. Beispielsweise soll die Standardimplementierung der Schnittstelle `IComponent` den Namen `Component` haben. Suffixe wie `Impl` sind zu vermeiden.
- Verwende keinen Unterstrich (_) im Schnittstellennamen.

4.10 Enumerationsnamen

Bezeichner für Enumerationstypen und Enumerationswerte müssen folgenden Regeln genügen:

- Verwende Pascal-Notation sowohl für den Typ als auch für die Werte einer Enumeration.
- Setze Abkürzungen sparsam ein.
- Verwende kein Suffix wie `Enum` im Typnamen.

Für Enumerationstypen, die keine Bitfelder darstellen, gilt zusätzlich:

- Verwende einen Namen im Singular für den Enumerationstyp, z.B. `ProcessState`.

Für Enumerationstypen, die Bitfelder darstellen, gilt zusätzlich:

- Verwende einen Namen im Plural für den Enumerationstyp, z.B. `VarFlags`.
- Füge stets das Attribut `[Flags]` zum Enumerationstyp hinzu.

4.11 Feldnamen

Bezeichner für Felder müssen in Abhängigkeit von der Lebensdauer, der Sichtbarkeit und des Zugriffs des zugrunde liegenden Feldes folgenden Regeln genügen:

- **Geschützte oder private Instanzfelder, die nicht konstant sind:** Verwende ungarische Notation sowie eines der Präfixe _ oder `m_`.

```
protected int _bEnabled;      // oder m_bEnabled
private int _nElementCount;   // oder m_nElementCount
```

- **Geschützte oder private statische Felder, die nicht konstant sind:** Verwende ungarische Notation sowie eines der Präfixe `s_` oder `ms_`.

```
protected static int s_nClassCount;   // oder ms_nClassCount
private static bool s_bInitialized;   // oder ms_bInitialized
```

- **Geschützte oder private konstante Felder (Instanzfelder und statische Felder):** Großschreibungsnotation ohne Präfix.

```
protected const int COLIDX_NAME = 0;
protected static readonly int COLIDX_VALUE = 1;
private const string NAME = "hugo";
private static readonly Bitmap IMAGE = new Bitmap(16, 16);
```

- **Öffentliche Felder (Instanzfelder und statische Felder, variabel und konstant):** Verwende Pascal-Notation ohne Präfix.

```
public string Name;
public static bool ExistsAnInstance;
public readonly int Value;
public static readonly string DefaultPath;
public const int NameIndex;
```

- Es wird dringend empfohlen, geschützte und öffentliche Felder, gleichgültig welcher Lebensdauer und welchen Zugriffs, durch entsprechende Eigenschaften zu ersetzen.

4.12 Parameternamen

Bezeichner für Parameter müssen folgenden Regeln genügen:

- Verwende ungarische Notation ohne Präfix.
- Verwende sprechende Namen. Parameternamen müssen so aussagekräftig sein, dass der Name und der Typ des Parameters ausreichen, um seine Bedeutung in den allermeisten Fällen erkennen zu können. Moderne Entwicklungsumgebungen stellen Parameternamen bereits beim Eingeben eines Methodenaufrufs dar; daher ist es für den Entwickler sehr hilfreich, wenn er umstandslos die nötigen Parameter angeben kann.
- Achte bei der Wahl des Namens darauf, dass die Bedeutung des Parameters und nicht dessen Datentyp beschrieben wird. Die Informationen über den Datentyp sollen nicht über das Maß der ungarischen Notation hinausgehen.
- Definiere keine reservierten Parameter. Sollte eine Methode in Zukunft zusätzliche Parameter benötigen, ist die Methode zu überladen.

4.13 Lokale Variablennamen

Bezeichner für lokale Variablen müssen folgenden Regeln genügen:

- Verwende ungarische Notation ohne Präfix. Ausnahme: Laufvariablen in Schleifenkonstrukten dürfen aus einem einzelnen kleinen Buchstaben bestehen, z.B. `i`, `j`, `k`.
- Verwende sprechende Namen. Parameternamen müssen so aussagekräftig sein, dass der Name und der Typ des Parameters ausreichen, um seine Bedeutung in den allermeisten Fällen erkennen zu können. Ausnahme: Laufvariablen in Schleifenkonstrukten, die aus einem einzigen Buchstaben bestehen, sind dadurch bereits als Laufvariable gekennzeichnet.
- Achte bei der Wahl des Namens darauf, dass die Bedeutung des Parameters und nicht dessen Datentyp beschrieben wird. Die Informationen über den Datentyp sollen nicht über das Maß der ungarischen Notation hinausgehen. Ausnahme: Laufvariablen in Schleifenkonstrukten, die aus einem einzigen Buchstaben bestehen, sind dadurch bereits als Laufvariable gekennzeichnet.
- Lokale Variablen dürfen im Laufe ihrer Lebensdauer ihre Bedeutung nicht ändern. In einem solchen Fall sind mehrere Variablen zu deklarieren.

4.14 Methodennamen

Bezeichner für Methoden müssen den folgenden Regeln genügen:

- Verwende ein Verb oder einen verbalen Ausdruck, um eine Methode zu benennen. Zulässige Beispiele sind: `RemoveAll`, `GetCharArray`, `Invoke`. Unzulässige Beispiele sind: `Comparable`, `FileAttributes`.

- Verwende Pascal-Notation.
- Setze Abkürzungen sparsam ein.

4.15 Eigenschaftsnamen

Bezeichner für Eigenschaften müssen den folgenden Regeln genügen:

- Verwende ein Substantiv, ein Adjektiv oder eine Besitzanzeige, um eine Eigenschaft zu benennen. Das Substantiv kann zusammengesetzt sein. Zulässige Beispiele sind: `Color`, `Enabled`, `HasElements`, `ForegroundColor`. Unzulässige Beispiele sind: `GetCharArray`, `Invoke`.
- Bei booleschen Eigenschaften darf dem Bezeichner optional ein Präfix `Is` vorangestellt werden. Umgekehrt darf dieses Präfix nicht verwendet werden, wenn die Eigenschaft nicht boolesch ist. Beispiel: Beide Varianten `Enabled` und `IsEnabled` sind zulässig.
- Verwende Pascal-Notation.
- Setze Abkürzungen spärlich ein.
- Eigenschaften dürfen durchaus den zugehörigen Typnamen im Namen führen. So ist es beispielsweise gestattet, eine Eigenschaft `Color` oder `ForeColor` vom Datentyp `Color` zu definieren. Umkehrt ist zu vermeiden, dass in einem Eigenschaftsname ein Typname verwendet wird, der nicht mit dem Datentyp der Eigenschaft übereinstimmt. Es ist also beispielsweise nicht gestattet, eine Eigenschaft `ForeColor` zu definieren, die als Datentyp `int` hat.

4.16 Ereignisnamen

Bezeichner für Ereignisse müssen den folgenden Regeln genügen:

- Verwende Pascal-Notation.
- Verwende ein Verb, um ein Ereignis zu benennen. Beispielsweise sind `Clicked`, `Painting` und `DroppedDown` gültige Bezeichner für Ereignisse.
- Verwende die Gerundform eines Verbs (die „ing-Form“) für Ereignisse, die vor einer bestimmten Aktion ausgelöst werden und die möglicherweise abgebrochen werden können. Das Muster `BeforeXxx` ist nicht gestattet. Beispiel: `WindowClosing`.
- Verwende die erste Vergangenheitsform eines Verbs (die „ed-Form“) für Ereignisse, die nach einer bestimmten Aktion ausgelöst werden. Das Muster `AfterXxx` ist nicht gestattet. Beispiel: `WindowClosed`.
- Für Ereignisnamen sind weder Präfixe noch Suffixe gestattet (dies gilt insbesondere für das Präfix `On`).
- Darüber hinaus soll für jedes definierte Ereignis in einer Klasse eine zugehörige geschützte überladbare Methode namens `OnXxx` zur Verfügung gestellt werden. Diese Methode muss den Ereignisparameter `e` haben; der Auslöser des Ereignisses ist immer die Instanz selbst und darf deshalb nicht als Parameter übergeben werden.

  Beispiel:

```
public class Window
{
   // your definitions here…

   protected virtual void OnClosed(EventArgs e)
   {
      if (Closed != null)
         Closed(this, e);
   }
```

```
    public event EventHandler Closed;
}
```

4.17 Delegatnamen

Bezeichner für Delegate müssen den folgenden Regeln genügen:

- Verwende Pascal-Notation.
- **Wenn der Delegat zur Definition eines Ereignisses verwendet wird:** Füge an den Namen das Suffix `EventHandler` hinzu. Beispiel: `WindowClosedEventHandler`.
- **Wenn der Delegat zur Definition eines Ereignisses verwendet wird:** Der Delegat muss zwei Parameter namens `sender` und `e` haben. Der Parameter `sender` repräsentiert das Objekt, welches das Ereignis ausgelöst hat und ist stets vom Typ `object`, auch dann, wenn ein spezialisierterer Typ angegeben werden könnte. Die zu dem Ereignis gehörigen Daten werden im Parameter `e` übergeben. Hierzu ist eine Klasse zu definieren, deren Name das Suffix `EventArgs` hat und die direkt oder indirekt von `EventArgs` abgeleitet ist.

  Beispiel:

  ```
  public class WindowClosedEventArgs : EventArgs
  {
     // your event data here…
  }

  public delegate void WindowClosedEventHandler(
     object sender,
     WindowClosedEventArgs e);
  ```

- **Wenn der Delegat zur Definition einer Callback-**Methode verwendet wird: Füge an den Namen das Suffix `Callback` hinzu. Beispiel: `AsyncOperationCompletedCallback`.

5 Richtlinien für Klassenmitglieder

Dieses Kapitel die Richtlinien, die bei der Definition von Mitgliedern einer Klasse (Felder, Methoden, Eigenschaften, Ereignisse) befolgt werden müssen.

5.1 Richtlinien für Eigenschaften

Zuerst muss ermittelt werden, ob an der betreffenden Stelle eher eine Eigenschaft oder Methode in Frage kommt. Diese Frage wird in 5.1.3 detailliert beantwortet.

Die Bezeichnung einer Eigenschaft muss den Regeln in 4.16 genügen.

5.1.1 Zustände

Eine Klasse muss dergestalt implementiert werden, dass seine Eigenschaften in beliebiger Reihenfolge gesetzt werden können. Das bedeutet mit anderen Worten, dass jede Eigenschaft gegenüber allen anderen Eigenschaften dieser Klasse nicht zustandsbehaftet sein darf.

Oftmals steht in einer Klasse eine bestimmte Funktionalität erst dann zur Verfügung, wenn der Anwender eine Reihe von Eigenschaften auf geeignete Werte gesetzt hat, oder wenn das Objekt einen bestimmten Zustand erreicht hat. Bevor dies der Fall ist, darf die Funktionalität nicht aktiv sein. Nachdem das Objekt den korrekten Zustand erreicht hat, muss sich die Funktionalität ohne expliziten Aufruf aktivieren. Die Semantik darf sich nicht dadurch ändern, indem die Eigenschaften in einer anderen Reihenfolge gesetzt werden. **Abweichungen von dieser Regel müssen in der betreffenden Klasse umfangreich dokumentiert werden!**

Beispiel:

Die Klasse `TextBox` hat zwei verwandte Eigenschaften namens `DataSource` und `DataField`. `DataSource` spezifiziert den Namen einer Datenbanktabelle und `DataField` spezifiziert den Namen einer Spalte in dieser Datenbanktabelle. Sobald beide Eigenschaften auf einen geeigneten Wert gesetzt wurden, verbindet sich das Objekt automatisch an diese Datenbank und modifiziert seinen Text entsprechend. Wenn keine oder nur eine Eigenschaft gesetzt ist, findet keine Verbindung statt.

Die beiden Varianten

```
TextBox t = new TextBox();
t.DataSource = "Publishers";
t.DataField = "AuthorID";
// Die Funktionalität ist nun aktiviert.
```

und

```
TextBox t = new TextBox();
t.DataField = "AuthorID";
t.DataSource = "Publishers";
// Die Funktionalität ist nun aktiviert.
```

führen zum selben Ergebnis: Die Reihenfolge spielt keine Rolle und es ist kein expliziter Aufruf zum Aktivieren der Funktionalität nötig.

Durch Setzen einer der beiden Eigenschaften auf `null` kann die Funktionalität deaktiviert werden, ohne dass die jeweils andere Eigenschaft ihren Wert verliert.

5.1.2 Ereignisse beim Ändern von Eigenschaften

Komponenten sollen ein Ereignis auslösen, wenn sie Anwender darüber informieren wollen, wenn sich die Eigenschaft der Komponente programmatisch geändert hat. Per Konvention muss ein solches Ereignis den Namen der betreffenden Eigenschaft mit dem Suffix `Changed` haben, z.B. `TextChanged` für die Eigenschaft `Text`. Eine Klasse muss ein solches Ereignis auslösen, indem es im `set`-Teil der Eigenschaft die geschützte `OnXxx`-Methode aufruft, die gemäß 4.16 obligatorisch ist.

Es ist darüber hinaus in manchen Fällen empfehlenswert, vor der Änderung einer Eigenschaft ein Ereignis auszulösen, welches besagt, dass sich die Eigenschaft ändern *wird*. Per Konvention muss ein solches Ereignis den Namen der betreffenden Eigenschaft mit dem Suffix `Changing` haben, z.B.

`TextChanging` für die Eigenschaft `Text`. Eine Klasse muss ein solches Ereignis auslösen, indem es im `set`-Teil der Eigenschaft die geschützte `OnXxx`-Methode aufruft, die gemäß 4.16 obligatorisch ist. Außerdem soll die Argumentklasse für ein solches Ereignis von `CancelEventArgs` abgeleitet sein, so dass ein Empfänger des Ereignisses ein Veto gegen die tatsächliche Änderung des Eigenschaftswerts einlegen kann.

Folgendes Beispiel soll die Zusammenhänge verdeutlichen:

```
class Control : Component
{
   private string _stText;

   public string Text
   {
      get { return _stText; }

      set
      {
         if (value != _stText)
         {
            CancelEventArgs e = new CancelEventArgs();
            OnTextChanging(e);
            if (e.Cancel)
            {
               return;   // Veto, daher Eigenschaft nicht ändern
               // Es ware auch statthaft, hier eine Ausnahme zu werfen,
               // wenn sich der Aufrufer auf die Änderung der Eigenschaft
               // verlässt.
            }

            _stText = value;
            OnTextChanged(EventArgs.Empty);
         }
      }
   }

   protected virtual void OnTextChanged(EventArgs e)
   {
      if (TextChanged != null)
         TextChanged(this, e);
   }

   protected virtual void OnTextChanging(CancelEventArgs e)
   {
      if (TextChanging != null)
         TextChanging(this, e);
   }

   public event EventHandler TextChanged;
   public event CancelEventHandler TextChanging;
}
```

5.1.3 Eigenschaft oder Methode?

Bei der Erstellung einer Klasse stellt in vielen Fällen die Frage, ob eher eine Eigenschaft oder eine Methode erstellt werden soll. Allgemein gilt: Methoden repräsentieren Aktionen und Eigenschaften repräsentieren Daten.

Für den Einsatz einer Eigenschaften spricht:

- Es gibt eine 1-zu-1-Entsprechung zwischen einem internen Feld und der Eigenschaft. Hat ein Control beispielsweise das Feld `_stText`, so macht eine entsprechende Eigenschaft `Text` Sinn.

Wenn einer der folgenden Punkte zutrifft, muss hingegen eine Methode implementiert werden:

- Bei der Operation handelt es sich um eine Konvertierung, wie in `Object.ToString`.

- Die Operation ist so teuer, dass der Aufrufer das Ergebnis zwischenspeichern sollte. Umgekehrt gilt für den Aufrufer: Das Ergebnis einer Methode muss zwischengespeichert werden, das Ergebnis einer Eigenschaft darf verworfen und jedes Mal neu ermittelt werden. Ausnahme: Arrays (siehe unten).
- Die Ausführung des `get`-Teil der Eigenschaft führt zu einem nach außen sichtbaren Seiteneffekt. Ausgenommen sind interne Seiteneffekte, die sich öffentlich nicht auswirken.
- Mehrmaliges Ausführen der Operation führt zu unterschiedlichen Ergebnissen.
- Die Ausführungsreihenfolge ist relevant. Gemäß 5.1.1 müssen Eigenschaften in jeder beliebigen Reihenfolge mit identischem Ergebnis aufrufbar sein.

Darüber hinaus sollte es vermieden werden, Eigenschaften zu definieren, die ein Array zurückliefern, und zwar aus folgenden Gründen:

- Um zu vermeiden, dass der Aufrufer den internen Zustand verändern kann, muss eine Kopie des Arrays zurückgeliefert werden.
- Der Aufrufer könnte sich anhand der Tatsache, dass es sich um eine Eigenschaft handelt, dazu entschließen, das Ergebnis nicht zwischenzuspeichern und jedes Mal das Ergebnis neu zu ermitteln. Dies führt dazu, dass bei jedem Aufruf eine Kopie des Arrays angelegt wird!

Da diese Regel leider in der *CoDeSys Automation Platform* an einigen Stellen verletzt wurde, sollte folgendes beachtet werden:

- Der Aufrufer sollte das Ergebnis einer Eigenschaft zwischenspeichern, wenn es sich beim Rückgabewert um ein Array handelt.
- Bei der Implementierung der Eigenschaft sollte erwogen werden, die Kopie des Arrays zwischenzuspeichern, so dass nur der erste Aufruf nach einer Änderung teuer ist.
- In Zukunft müssen die oben definierten Regeln befolgt werden!

Folgendes Beispiel soll die Grundideen der aufgestellten Regeln beleuchten:

```
class Connection
{
   // Die nachfolgenden drei Mitglieder sind Eigenschaften,
   // da sie in jeder beliebigen Reihenfolge gesetzt werden
   // können.
   public string DnsName { get { } set { } }
   public string UserName { get { } set { } }
   public string Password { get { } set { } }

   // Das folgende Mitglied ist eine Methode, da die
   // Aufrufreihenfolge relevant ist. Die Methode darf
   // erst dann aufgerufen werden, wenn die obigen drei
   // Eigenschaften gesetzt wurden.
   public bool Execute() { }
}
```

5.1.4 Nur-lesbare und nur-schreibbare Eigenschaften

- Eine nur-lesbare Eigenschaft ist zu definieren, wenn der Anwender das entsprechende Feld nicht ändern können darf.
- Eine nur-schreibbare Eigenschaft darf nicht definiert werden.

5.1.5 Indizierte Eigenschaften

Die Regeln für indizierte Eigenschaften sind für die Programmiersprache C# einfach und überschaubar, da die für andere Sprachen geforderten Regeln bereits per Sprachspezifikation erfüllt sind.

- Definiere eine indizierte Eigenschaft mit integralem Argument, wenn die Klasse eine Spezialisierung einer Liste darstellt.

- Definiere eine indizierte Eigenschaft mit beliebigem Argument, wenn die Klasse eine Spezialiserung einer Tabelle darstellt.

5.2 Richtlinien für Ereignisse

Die folgenden Regeln gelten für die Definition von Ereignissen.

- Wähle einen Namen für das Ereignis, der den Regeln gemäß 4.16 entspricht.
- In englischsprachiger Dokumentation ist der Ausdruck „raise an event“ zu verwenden. Ausdrücke wie „fire an event“ oder „trigger an event“ sind zu vermeiden.
- Der Rückgabewert für eine Methode, die ein Ereignis behandelt, muss `void` sein, z.B.

```
public delegate void MouseEventHandler(
    object sender,
    MouseEventArgs e);
```

- Definiere eine Klasse, die die Argumente eines Ereignisses transportiert und leite diese Klasse direkt oder indirekt von `EventArgs` oder `CancelEventArgs` ab. Falls mit einem Ereignis keine Daten transportiert werden müssen, verwende direkt `EventArgs` bzw. `CancelEventArgs` als Argumentklasse. Beispiel:

```
public class MouseEventArgs : EventArgs
{
   private int _nX;
   private int _nY;
   // more data…

   public MouseEventArgs(int x, int y)
   {
      _nX = x;
      _nY = y;
   }

   public int X
   {
      get { return _nX; }
   }

   public int Y
   {
      get { return _nY; }
   }
}
```

- Definiere für jedes Ereignis einer Klasse eine zugehörige geschützte virtuelle Methode, die das Ereignis auslöst. Der Name dieser Methode muss mit dem Ereignisnamen übereinstimmen, wobei das Präfix `On` vorangestellt werden muss. Einziger Parameter dieser Methode ist eine Instanz der oben angesprochenenen Argumentklasse. Der Grund für dieses Vorgehen liegt darin begründet, dass eine abgeleitete Klasse das Ereignis behandeln kann, indem es einfach diese Methode überschreibt. Ohne diese Methode müsste sich eine abgeleitete Klasse auf ein Ereignis der Basisklasse anmelden, was unnötigen Overhead und mangelnde Übersichtlichkeit bedeuten würde. Ein Beispiel für eine solche Methode ist:

```
protected virtual void OnMouseClicked(MouseEventArgs e)
{
   if (MouseClicked != null)
      MouseClicked(this, e);
}

public event MouseEventHandler MouseClicked;
```

  Zu beachten ist, dass möglicherweise in einer abgeleiteten Klasse die Basisimplementierung nicht aufgerufen wird. Daraus folgt, dass sich in dieser Methode kein Code befinden darf,

dessen Ausführung zwingend notwendig ist.

Diese Technik ist nicht geeignet für Klassen, die als `sealed` deklariert sind. In diesen Fällen kann das Ereignis direkt ausgelöst werden.

- Die Implementierung einer Klasse muss darauf vorbereitet sein, dass während der Behandlung eines Ereignisses alle möglichen Operationen aufgerufen werden könnten. Das bedeutet, dass vor Auslösung eines Ereignisses das Objekt sich stets in einem konsistenten Zustand befinden muss.

5.3 Richtlinien für Methoden

5.3.1 Allgemeine Regeln

- Wähle einen Namen für die Methode, der den Regeln gemäß 4.14 entspricht.
- Standardmäßig sind Methoden nicht virtuell. Dieser Standard muss beibehalten werden, wenn die Methode nicht von abgeleiteten Klassen überschrieben werden soll. Im Abschnitt 6.1 werden detaillierte Informationen über Vererbungsrichtlinien angegeben.

5.3.2 Überladung von Methoden

Überladung bedeutet, dass eine Klasse zwei Methoden mit gleichem Namen, aber unterschiedlicher Parametersignatur definiert. Hier werden einige Regeln zu diesem Thema aufgestellt:

- Verwende Methodenüberladung genau dann wenn verschiedene Methoden zur Verfügung gestellt werden sollen, die semantisch den selben Effekt haben.
- Achte auf die korrekte Benennung der Parameter. Bei mehreren überladenen Methoden muss die komplexere Methode Parameter definieren, die den Unterschied zur einfacheren Methode klar anzeigen. Dies lässt sich durch folgendes Beispiel anschaulich erklären: Der folgende Code enthält zwei Methoden: die erste führt eine Suche durch, die Groß-/Kleinschreibung nicht berücksichtigt, bei der zweiten Methode kann dieses Verhalten parametriert werden. Aus diesem Grund heißt der entsprechende Parameter `bIgnoreCase`, um anzuzeigen, dass in der einfacheren Methode „ignore case“ offensichtlich das Standardverhalten ist. Ein Name `bCaseSensitive` wäre hier fehl am Platz und würde beim Anwender zu falschen Schlussfolgerungen führen.

```
// Methode #1: ignoreCase = false
int FindSubstring(string stFindWhat);
// Methode #2: Zeigt an, wie das Standardverhalten bzgl.
// Methode #1 geändert wurde
int FindSubstring(string stFindWhat, bool bIgnoreCase);
```

- Achte auf die konsistente Benennung und Reihenfolge der Parameter. Allgemein ist es üblich, bei mehreren Methodenüberladungen eine wachsende Anzahl von Parametern zu definieren, die es dem Anwender erlauben, der Methode Informationen unterschiedlichen Detaillierungsgrads zu übergeben. Die Reihenfolge der Parameter darf dabei nicht geändert werden. Beispiel:

```
public void Execute()
{
   Execute("defaultForA", "defaultForB", "defaultForC");
}

public void Execute(string a)
{
   Execute(a, "defaultForB", "defaultForC");
}

public void Execute(string a, string b)
{
   Execute(a, b, "defaultForC");
```

```
}

public virtual void Execute(string a, string b, string c)
{
   // Operation hier…
}
```

- Bei mehreren überladenen Methoden soll nur die Methode mit den meisten Parametern als virtuell deklariert werden (und nur dann, wenn dies auch nötig ist). Dies ist auch in obigem Beispiel dargestellt.

5.3.3 Methoden mit variabler Parameteranzahl

Wenn nötig, können Methoden definiert werden, die eine variable Anzahl von Parametern haben. Dies wird durch das Schlüsselwort `params` erreicht. Beispiel:

```
void Format(string stFormat, params object[] args);
```

Falls die Ausführungsgeschwindigkeit für solche Methoden kritisch ist (und nur dann!), können für sinnvolle Spezialfälle Überladungen definiert werden. Beispiel:

```
void Format(string stFormat);
void Format(string stFormat, object arg0);
void Format(string stFormat, object arg0, object arg1);
void Format(string stFormat, object arg0, object arg1, object arg2);
void Format(string stFormat, params object[] args);
```

5.4 Richtlinien für Konstruktoren

Die folgenden Regeln gelten für die Definition von Konstruktoren.

- Definiere einen privaten Standardkonstruktor, wenn die Klasse nur aus statischen Methoden und Eigenschaften besteht. Dies verhindert, dass eine Instanz der Klasse angelegt werden kann. Alternativ kann die Klasse auch als `abstract` oder `static` (seit C# 2.0) deklariert werden.
- Minimiere den Code-Umfang eines Konstruktors. Ein Konstruktor soll möglichst nur die übergebenen Parameter innerhalb der Instanz speichern. Teure Operationen sollen dann durch einen expliziten Methodenaufruf angestoßen werden.
- Definiere für jede Klasse einen Standardkonstruktor (also ein Konstruktor ohne Parameter). Falls die Klasse nicht instanzierbar sein soll, muss der Standardkonstruktur privat oder geschützt (im Falle einer abstrakten Klasse) sein. Falls überhaupt kein Konstruktor angegeben wird, fügt der C#-Compiler automatisch einen Standardkonstruktor hinzu. Es ist aber zu beachten, dass dieser automatisch generierte Standardkonstruktor wieder entfernt wird, sobald später explizit ein parametrierter Konstruktor hinzugefügt wird. Dies könnte in Anwender-Code zu Compile-Fehlern führen.
- Die Verwendung eines parametrierten Konstruktors muss eine Abkürzung gegenüber dem Setzen der entsprechenden Eigenschaften sein. Das bedeutet, dass kein semantischer Unterschied bestehen darf zwischen dem Aufruf eines parametrierten Konstruktors gegenüber dem Aufruf eines Standardkonstruktors, gefolgt vom Setzen der entsprechenden Eigenschaften. Beispiel: Folgende drei Code-Stücke müssen äquivalent sein:

  ```
  Sample sample = new Sample();
  sample.A = "a";
  sample.B = "b";

  Sample sample = new Sample("a");
  sample.B = "b";

  Sample sample = new Sample("a", "b");
  ```

- Achte auf die konsistente Benennung und Reihenfolge der Parameter. Allgemein ist es üblich, bei mehreren Konstruktoren eine wachsende Anzahl von Parametern zu definieren, die es dem Anwender erlauben, dem Konstruktor Informationen unterschiedlichen

Detaillierungsgrads zu übergeben. Die Reihenfolge der Parameter darf dabei nicht geändert werden. Beispiel:

```
public class Sample
{
   public Sample()
      : this ("defaultForA", "defaultForB", "defaultForC")
   {
   }

   public Sample(string a)
      : this (a, "defaultForB", "defaultForC")
   {
   }

   public Sample(string a, string b)
      : this (a, b, "defaultForC")
   {
   }

   public Sample(string a, string b, string c)
      : this (a, b, c)
   {
      // Initialisierungen hier…
   }
}
```

5.5 Richtlinien für Felder

Die folgenden Regeln gelten für die Definition von Feldern:

- Verwende keine Instanzfelder, die `public` oder `protected` sind. Wenn es vermieden wird, Felder direkt zu veröffentlichen, können Klassen besser versioniert werden, da Felder nicht in Eigenschaften umgewandelt werden können, ohne die binäre Kompatibilität zu verlieren. Anstatt Felder öffentlich zu machen, sollten Eigenschaften mit `get`- und `set`-Zugriff zur Verfügung gestellt werden. Dadurch können spätere Verbesserungen wie Objekterzeugung bei Bedarf oder Ereignisse bei Wertänderung gemacht werden. Das folgende Beispiel zeigt die korrekte Verwendung von privaten Instanzfeldern in Verbindung mit öffentlichen Eigenschaften:

```
public class Point
{
   private int _xValue;
   private int _yValue;

   public Point(int xValue, int yValue)
   {
      _xValue = xValue;
      _yValue = yValue;
   }

   public int X
   {
      get { return _xValue; }
      set { _xValue = value; }
   }

   public int Y
   {
      get { return _yValue; }
      set { _yValue = value; }
   }
}
```

- Veröffentliche ein Feld gegenüber einer abgeleiteten Klasse mittels einer geschützten Eigenschaft, die den Wert des Feldes zurückliefert. Dies wird durch folgendes Beispiel deutlich:

```
public class Control : Component
{
   private int _handle;

   protected int Handle
   {
      get { return _handle; }
   }
}
```

- Verwende das Schlüsselwort `const`, um konstante Felder zu deklarieren, deren Wert sich nicht ändern wird. Verwende hierbei Großschreibungsnotation. Compiler ersetzen diesen Wert direkt im Code.
- Verwende öffentliche statische Nur-Lese-Felder für vordefinierte Objektinstanzen. Solche Felder sind in Großschreibungsnotation zu benennen; Pascal-Notation ist ebenfalls zulässig. Beispiel:

```
public class Color
{
   public static readonly Color RED = new Color(0x0000FF);
   public static readonly Color GREEN = new Color(0x00FF00);
   public static readonly Color BLUE = new Color(0xFF0000);
   public static readonly Color BLACK = new Color(0x000000);
   public static readonly Color WHITE = new Color(0xFFFFFF);

   public Color(int rgb)
   {
      // ...
  }

   public Color(byte r, byte g, byte b)
   {
      // ...
   }

   // ...
}
```

- Schreibe alle Wörter eines Feldes grundsätzlich aus. Abkürzungen sind nur dort zu verwenden, wo sie verstanden werden. Verwende ungarische Notation mit Präfix _ oder `m_`, z.B. `_url` oder `m_destinationUrl`.

5.6 Richtlinien für Parameter

Die folgenden Regeln gelten für die Definition und Verwendung von Parametern:

- Überprüfe die Korrektheit von Parametern in jeder öffentlichen oder geschützten Methode bzw. `set`-Zugriff von Eigenschaften. Werfe eine aussagekräftige Ausnahme, wenn ein Parameterargument falsch ist. Dazu sollte die Klasse `System.ArgumentException` oder eine davon abgeleitete Klasse verwendet werden. Das folgende Beispiel überprüft Parameterargumente auf ihre Korrektheit und löst gegebenenfalls aussagekräftige Ausnahmen aus:

```
public class SampleClass
{
   public int Count
   {
      get { return _count; }

      set
```

```
        {
           if (value < 0 || value >= MaxValue)
              throw new ArgumentOutOfRangeException("value");
           _count = value;
        }
     }

     public void Select(int nStart, int nEnd)
     {
        if (nStart < 0)
           throw new ArgumentException("nStart", nStart, "nStart must
be non-negative.");
        if (nEnd < nStart)
           throw new ArgumentException("nEnd", nEnd, "nEnd must be
greater or equal than nStart.");
     // ...
     }
}
```

Beachte, dass die tatsächliche Überprüfung nicht ungedingt in der öffentlichen oder geschützten Methode selbst durchgeführt werden muss. Sie kann auch in einer aufgerufenen privaten Routine erfolgen. Entscheidend ist, dass die gesamte öffentliche Schnittstelle nach aussen hin diesbezüglich abgesichert ist.

- Bedenke alle Implikationen, die mit Parameterübergabe per Wert oder per Referenz zusammenhängen. Wertübergabe eines Parameters kopiert den übergebenen Wert und hat keinen Einfluss auf den Originalwert, z.B.

  ```
  public bool IsPrime(int nValue);
  public void Add(object item);
  ```

  Referenzübergabe eines Parameters übergibt den Speicherort des Werts. Daher können Änderungen am übergebenen Wert vorgenommen werden. Beispiel:

  ```
  public void Exchange(ref int nValue1, ref int nValue2);
  public void CreateOrUse(ref StringBuilder sb);
  ```

6 Richtlinien für Typen

Typen sind die Kapselungseinheit der Common Language Runtime. Eine detaillierte Beschreibung unterstützter Datentypen kann in [2] nachgelesen werden.

6.1 Richtlinien für Basisklassen

Eine Klasse ist die am meisten verbreitete Typart. Eine Klasse kann sowohl abstrakt als auch versiegelt sein. Eine abstrakte Klasse verlangt von jeder abgeleiteten Klasse eine Implementierung. Von einer versiegelten Klasse kann nicht abgeleitet werden.

Basisklassen sind nützlich, um Klassen zu gruppieren, die eine gemeinsame Menge an Funktionalität besitzen. Basisklassen können einen Standardumfang zur Verfügung stellen und erlauben Anpassung durch Ableitung.

Für eine Klasse sollte ein Konstruktor explizit implementiert werden. Typischerweise fügen Compiler automatisch einen öffentlichen Standardkonstruktor zu Klassen hinzu, die selbst keinen Konstruktor definieren. Dies kann aber für Verwender einer solchen Klasse irreführend sein, insbesondere dann, wenn eigentlich keine Instanzen dieser Klasse erzeugt werden dürften. Daher ist es guter Stil, stets mindestens einen Konstruktor für eine Klasse zu erstellen. Falls eine Klasse nicht instanzierbar sein soll, muss dieser Konstruktor privat sein.

6.1.1 Basisklasse oder Schnittstelle?

Diese Frage wird in vielen Codierrichtlinien ausführlich behandelt. Im Rahmen der *CoDeSys Automation Platform* spielen diese Richtlinien aber eine eher untergeordnete Rolle, da aufgrund der Architektur Vererbung über Komponentengrenzen hinweg nur in Ausnahmefällen möglich ist. Aus diesem Grund werden in diesem Kapitel keine weiteren Richtlinien behandelt.

6.1.2 Geschützte Methoden und Konstruktoren

Klassenanpassungen sind durch geschützte Methoden zur Verfügung zu stellen. Die öffentliche Schnittstelle einer Basisklasse soll dem Verwender ein großer Funktionsumfang zur Verfügung stellen. Jedoch ist es meist so, dass der Implementierer einer abgeleiteten Klasse möglichst wenige Methoden implementieren möchte, um diesen großen Funktionsumfang weiterzugeben. Um dieses Ziel zu erreichen, sollte eine Reihe nicht-virtueller Methoden definiert werden, die jeweils eine einzelne geschützte Methode aufrufen, die wiederum die eigentliche Implementierung beinhaltet. Diese geschützte Methode sollte ein angehängtes `Impl` im Namen haben. Dieses Design-Pattern wird häufig als Template-Methode bezeichnet. Folgendes Beispiel demonstriert die Vorgehensweise:

```
public class MyClass
{
   private int _x;
   private int _y;
   private int _nWidth;
   private int _nHeight;
   private BoundsSpecified _specified;

   public void SetBounds(int x, int y, int nWidth, int nHeight)
   {
      SetBoundsImpl(x, y, nWidth, nHeight, _specified);
   }

   public void SetBounds(int x, int y, int nWidth, int nHeight,
BoundsSpecified specified)
   {
      SetBoundsImpl(x, y, nWidth, nHeight, specified);
   }

   protected virtual void SetBoundsImpl(int x, int y, int nWidth, int
nHeight, BoundsSpecified specified)
   {
      _x = x;
      _y = y;
      _nWidth = nWidth;
      _nHeight = nHeight;
      _specified = specified;
   }
}
```

C#-Compiler fügen automatisch einen parameterlosen öffentlichen Konstruktor zu einer Klasse hinzu, für die kein Konstruktor explizit definiert wurde. Für bessere Dokumentation und bessere Lesbarkeit des Quellcodes sollte daher explizit ein geschützter Konstruktor für alle abstrakten Klassen definiert werden.

6.1.3 Statische Klassen

Klassen, die ausschließlich statische Methoden und Eigenschaften beinhalten, sollten ihrerseits als `static` deklariert sein (seit C# 2.0).

6.2 Richtlinien für Werttypen

Ein Werttyp beschreibt einen Wert, der in Form einer Bitsequenz auf dem Stack gespeichert wird. Eine Beschreibung aller im .NET-Framework eingebauten Werttypen befindet sich in [3]. Dieses Kapitel gibt Richtlinien für die Verwendung von Strukturen (`struct`) und Enumerationen (`enum`) vor.

6.2.1 Richtlinien für Strukturen

Es wird empfohlen, für Typen, die die folgenden Kriterien erfüllen, `struct` zu verwenden:

- Verhalten wie primitive Datentypen
- Instanzgöße unter 16 Bytes
- Unveränderlichkeit
- Wertsemantik ist wünschenswert

Das folgende Beispiel zeigt eine korrekt definierte Struktur:

```
public struct Complex : IFormattable
{
   private double _dReal;
   private double _dImag;

   public string ToString(string stFormat, IFormatProvider
formatProvider)
   {
      // ...
   }

   public override ToString()
   {
      // ...
   }

   public double Real
   {
      get { return _dReal; }
   }

   public double Imag
   {
      get { return _dImag; }
   }

   public override int GetHashCode()
   {
      // ...
   }

   public override bool Equals(object obj)
   {
      // ...
   }

   public static Complex Parse(string st)
   {
      // ...
   }
}
```

In C# ist es nicht gestattet, dass eine Struktur einen expliziten Standardkonstruktor besitzt. Das Laufzeitsystem generiert automatischen einen Standardkonstruktor, der jedes Mitglied zu Null initialisiert. Dadurch ist es möglich, ein Array von Strukturen zu erzeugen, ohne pro Element einen Konstruktor aufrufen zu müssen. Es ist daher zu vermeiden, dass eine Struktur von einem initialen Konstruktoraufruf abhängt. Ausserdem muss die Struktur derart gestaltet sein, dass es einen gültigen Zustand darstellt, wenn jedes Feld mit Null initialisiert ist.

6.2.2 Richtlinien für Enumerationen

Die folgenden Regeln gelten für die Verwendung von Enumerationstypen:

- Verwende kein `Enum`-Suffix im Namen.
- Verwende eine Enumeration, um Parameter, Eigenschaften und Rückgabewerte zu typisieren. Dadurch ist sichergestellt, dass Entwicklungswerkzeuge die möglichen Werte für Eigenschaften und Parameter kennen. Das folgende Beispiel zeigt, wie ein Enumerationstyp definiert wird:

```
public enum FileMode
{
   Append,
   Create,
   CreateNew,
   Open,
```

```
   OpenOrCreate,
   Truncate
}
```

Ein weiteres Beispiel, wie dieser Enumerationstyp in einem Konstruktor verwendet werden kann:

```
public FileStream(string stPath, FileMode mode);
```

- Verwende eine Enumeration statt einer Reihe von Konstanten.
- Verwende keine Enumeration für offene Mengen (z.B. die Version des Betriebssystems).
- Verwende das Attribute `System.FlagsAttribute` nur dann, wenn eine bitweise Oder-Operation auf den numerischen Werten sinnvoll ist. In diesem Fall sollen die Enumerationswerte Zweierpotenzen sein, um einfach kombiniert werden zu können. Dieses Vorgehen soll durch folgendes Beispiel verdeutlicht werden:

```
[Flags]
public enum WatcherChangeTypes
{
   Created = 0x01,
   Deleted = 0x02,
   Changed = 0x04,
   Renamed = 0x08,
   All = Created | Deleted | Changed | Renamed;
}
```

- Stelle benannte Konstanten für häufig verwendete Flag-Kombinationen zur Verfügung (siehe `All` in vorherigem Beispiel).
- Verwende stets `int` (`System.Int32`) als Basistyp für eine Enumeration, außer einer der folgenden beiden Bedingungen ist erfüllt:
  - Die Enumeration repräsentiert Flags und es gibt mehr als 32 davon (oder wird in Zukunft mehr als 32 davon geben).
  - Der Typ muss aus Gründen der Rückwärtskompatibilität von `int` verschieden sein.
- Überprüfe auch bei Enumerationsargumenten die Gültigkeit. Es ist erlaubt, jede beliebige Ganzzahl in eine Enumeration umzuwandeln, selbst dann, wenn dieser Wert nicht in der Enumeration definiert ist. Die Validierung kann wie folgt vorgenommen werden:

```
public enum Color
{
   Red,
   Yellow,
   Green
}

public class MyClass
{
   public void SetColor(Color color)
   {
      if (!Enum.IsDefinied(typeof(Color), color)
         throw new ArgumentOutOfRangeException("color");
      // ...
   }
}
```


6.3 Richtlinien für Delegate

Ein Delegat ist ein mächtiges Werkzeug, um Methodenaufrufe zu kapseln. Es ist nützlich für Ereignisbenachrichtigungen und Rückruffunktionen.

6.3.1 Ereignisbenachrichtigungen

Verwende ein geeignetes Design Pattern auch für solche Ereignisse, die nicht mit der Benutzeroberfläche zusammenhängen. Nähere Informationen zur Verwendung von Ereignissen befinden sich in Kapitel 5.2.

6.3.2 Rückruffunktionen

Eine Rückruffunktion wird an eine Methode übergeben, um während deren Ausführung mehrmals aufgerufen zu werden. Das klassische Beispiel hierfür ist eine Rückruffunktion `Compare`, die an eine Sortierroutine übergeben wird. Solche Methoden sollten die Richtlinien in Kapitel 13.3 befolgen.

Benenne Delegate, die für Rückruffunktionen verwendet werden, mit dem Suffix `Callback`.

6.4 Richtlinien für Attribute

Das .NET-Framework erlaubt Entwicklern, neue Arten deklarativer Informationen zu erfinden, diese Informationen an verschiedenen Programmentitäten zu verwenden und sie zur Laufzeit abzufragen. Beispielsweise könnte eine Klassenbibliothek das Attribute `HelpAttribute` definieren, welches zu Elementen wie Klassen oder Methoden platziert werden könnte, um eine Abbildung vom Element zur jeweiligen Dokumentation zu schaffen.

Die folgenden Regeln gelten für Attributklassen:

- Benenne Attributklassen mit einem `Attribute`-Suffix, z.B. `ObsoleteAttribute`.
- Spezifiziere `AttributeUsage` für die Attributklasse, um ihre Verwendung detailliert festzulegen. Beispiel:

```
[AttributeUsage(AttributeTargets.All, Inherited = false,
AllowMultiple = true)]
public class ObsoleteAttribute : Attribute
{
   // ...
}
```

- Wenn möglich sollten Attributklassen versiegelt sein, so dass andere Klassen nicht davon ableiten können.
- Verwende Konstruktorparameter für erforderliche Parameter. Stelle für jeden Konstruktorparameter eine entsprechende Nur-Lese-Eigenschaft zur Verfügung, so dass das Argument zur Laufzeit abgefragt werden kann.
- Verwende benannte Argumente für optionale Parameter. Hierbei muss für jedes benannte Argument eine Lese-Schreib-Eigenschaft definiert werden.
- Definiere keinen Parameter, der sowohl über den Konstruktor als auch über den Namen gesetzt werden kann.

Folgendes Beispiel verdeutlicht das Design-Muster:

```
[AttributeUsage(AttributeTargets.All)]
public class NameAttribute : Attribute
{
   private string _stUserName;
   private int _nAge;

   // Konstruktorparameter (erforderlicher Parameter)
   public NameAttribute(string stUserName)
   {
      // ...
   }

   public string UserName
   {
      get { return _stUserName; }
   }

   // Benannter Parameter (optionaler Parameter)
   public int Age
   {
      get { return _nAge; }
      set { _nAge = value; }
   }
}
```

6.5 Richtlinien für geschachtelte Typen

Ein geschachtelter Typ ist ein Typ, der innerhalb eines anderen Typs definiert wird. Geschachtelte Typen sind sehr nützlich, um private Implementationsdetails eines Typs zu kapseln, da sie Zugriff auf die privaten Daten haben. Beispiel hierfür ist ein Enumerator über eine Datensammlung.

Öffentliche geschachtelte Typen sollten nur dann verwendet werden, wenn der innere Typ logisch zum äußeren Typ gehört.

Verwende geschachtelte Typen nicht, falls folgendes gilt:

- Der Typ muss vom Anwendercode instanziert werden. Falls der Typ einen öffentlichen Konstruktor hat, sollte er typischerweise nicht geschachtelt sein. Begründung: Wenn ein geschachtelter Typ instanziert werden kann, ist dies ein Zeichen dafür, dass er ohne den äußeren Typ verwendet werden kann, was darauf hinweist, dass er keine enge Beziehung zum äußeren Typ hat.
- Im Anwendercode befinden sich viele Referenzen auf den Typ.

7 Richtlinien für Anweisungen

Dieses Kapitel beschreibt Richtlinien, die für verschiedene Anweisungen (`if`, `switch`, `while` etc.) zu beachten sind.

7.1 Bedingungen

- Bedingte Anweisungen (`if`, `switch`, `while`, `do`, `for`) dürfen in ihren Bedingungsteilen nicht ausschließlich Zuweisungen enthalten, da sonst Verwechslungsgefahr zwischen dem Zuweisungsoperator = und dem Vergleichsoperator == besteht.

  Negativbeispiel:

  ```
  if (bCondition = true)
  {
     …
  }
  ```

  Positivbeispiel:

  ```
  bCondition = true;
  if (bCondition)
  {
     …
  }
  ```

  Aber: Folgendes Konstrukt ist erlaubt, da es nicht *ausschließlich* Zuweisungen enthält:

  ```
  string stLine;
  while ((stLine = textReader.ReadLine()) != null)
  {
     …
  }
  ```

- In booleschen Ausdrücken sollen Klammern verwendet werden, wenn es der Übersichtlichkeit dient, auch wenn es gemäß der Vorrangregeln nicht nötig wäre.

  ```
  (A && B) || ((A || B) && C)
  ```
  statt
  ```
  A && B || (A || B) && C
  ```

7.2 `goto`-Anweisung

Die Verwendung der goto-Anweisung ist zu vermeiden.

8 Auslösung und Behandlung von Ausnahmen

Dieses Kapitel beschreibt Richtlinien, die für die Auslösung und Behandlung von Ausnahmen gelten.

8.1 Allgemeine Regeln

- Für alle Codepfade, die in einer Ausnahme münden, sollte eine Methode zur Verfügung stehen, die auf Erfolg prüfen, ohne eine Ausnahme zu werden. Beispielsweise kann mittels einem Aufruf von `File.Exists` eine `FileNotFoundException` vermieden werden. Möglicherweise ist dies nicht immer möglich; dennoch muss es das Ziel sein, dass unter normalen Umständen keine Ausnahmen ausgelöst werden.
- Benenne Ausnahmeklassen mit einem Exception-Suffix, z.B.

```
public class FileNotFoundException : ApplicationException
{
   // ...
}
```

- Verwende bei der Entwicklung von Ausnahmeklassen die folgenden Konstruktoren:

```
public class XxxException : ApplicationException
{
   public XxxException()
   {
      // ...
   }

   public XxxException(string stMessage)
   {
      // ...
   }

   public XxxException(string stMessage, Exception innerException)
   {
      // ...
   }

   public XxxException(SerializationInfo info, StreamingContext
context)
   {
      // ...
   }
}
```

- Verwende wenn möglich vordefinierte Ausnahmeklassen. Es sollte nur dann eine neue Ausnahmeklasse definiert werden, wenn erwartet wird, dass Anwendercode diese spezielle Ausnahme abfängt und daraufhin spezielle Aktionen ausführt. Andernfalls müsste der Anwendercode den Text der Ausnahme parsen, was schlechte Performance und Wartbarkeit zur Folge hat. Beispielsweise macht es Sinn, eine `FileNotFoundException` zu definieren, da der Anwendercode als Reaktion die fehlende Datei erzeugen könnte, während eine allgemeinere `FileIOException` höchstwahrscheinlich zu keiner expliziten Reaktion im Anwendercode führt.
- Leite neue Ausnahmeklassen von `ApplicationException` oder spezielleren Klassen ab, nicht von `SystemException` oder gar `Exception`.
- Gruppiere neue Ausnahmeklassen nach Namensraum. Beispielsweise leiten alle Ausnahmeklassen im Namensraum `System.IO` von `System.IO.IOException` ab, alle in `Microsoft.Media` von `Microsoft.Media.MediaException`.
- Verwende in allen Ausnahmen lokalisierte Beschreibungen. Der Anwender wird typischerweise diese Beschreibung als Fehlermeldung sehen.

- Verwende grammatikalisch korrekte Fehlermeldungen mit Interpunktion. Jeder Satz in der Meldung sollte mit einem Punkt enden. Der Code, der Fehlermeldungen zur Anzeige bringt, braucht sich nicht um eventuell fehlende Interpunktion kümmern.
- Stelle Eigenschaften für programmatischen Zugriff zur Verfügung, wenn es ein Szenario gibt, in dem diese zusätzliche Information für den Anwendercode nützlich ist.
- Vermeide es, privilegierte Informationen wie z.B. Pfade im lokalen Dateisystem in Fehlermeldungen aufzunehmen. Bösartiger Code könnte hierdurch an private Informationen gelangen.
- Verwende keine Ausnahmen für normale oder erwartete Fehler oder zur Steuerung des Kontrollflusses.
- Für ganz häufige Fehler sollte `null` als Rückgabewert verwendet werden, anstatt eine Ausnahme auszulösen. Beispielsweise liefert die Methode `File.Open` eine `null`-Referenz zurück, wenn die Datei nicht gefunden werden konnte, und löst eine Ausnahme aus, wenn die Datei gesperrt ist.
- Gestalte Klassen derart, dass bei normaler Verwendung niemals eine Ausnahme ausgelöst wird. Im folgenden Beispiel stellt die Klasse `FileStream` eine Möglichkeit zur Verfügung, das Ende des Datenstroms festzustellen, ohne dass eine Ausnahme behandelt werden muss.

```
public void Open()
{
   FileStream stream = File.Open("myfile.txt", FileMode.Open);
   byte b;

   while ((b = stream.ReadByte()) >= 0)
   {
      // do something.
   }
}
```

- Löse eine `InvalidOperationException` immer dann aus, wenn der Aufruf einer Methode bzw. eines set-Zugriffs auf eine Eigenschaft bezüglich des Objektstatus nicht passend ist.
- Löse eine `ArgumentException` oder eine davon abgeleitete Ausnahme immer dann aus, wenn einer Methode oder einer Eigenschaft ein ungültiges Argument übergeben wird.
- Beachte, dass der Stacktrace einer Ausnahme immer dort beginnt, wo die Ausnahme ausgelöst wird, und nicht dort, wo sie mit `new` erzeugt wird.
- Erstelle Methoden zur Erzeugung von Ausnahmen. Typischerweise wird ein und dieselbe Ausnahme an verschiedenen Stellen in der Implementierung ausgelöst. Um Code-Duplikate zu vermeiden, sollten Hilfsroutinen verwendet werden, die die Ausnahme erzeugen und zurückliefern. Beispiel:

```
class File
{
   private string _stFileName;

   public byte[] Read(int nBytes)
   {
      if (!ReadFile(_stFileName, nBytes))
         throw NewFileIOException();
      // ...
   }

   private FileException NewFileIOException()
   {
      // build localized string including _stFileName
      string stDescription = ...;
      return new FileException(stDescription);
   }
}
```

- Löse Ausnahmen aus, anstatt Fehlercodes oder `HRESULT`s zurückzuliefern.

- Löse immer die speziellste Ausnahme aus.
- Setze aller Felder in den verwendeten Ausnahmen.
- Verwende innere Ausnahmen (verkette Ausnahmen, `InnerException`).
- Vermeide, Ausnahmen zu fangen und erneut auszulösen, wenn nicht zusätzliche Informationen hinzugefügt werden oder sich der Typ der Ausnahme ändert.
- Löse keine Ausnahmen vom Typ `NullReferenceException` oder `IndexOutOfRangeException` aus.
- Prüfe Argumente in geschützten und internen Methoden und Eigenschaften. Falls Argumente nicht geprüft werden, ist dies in der Dokumentation ausdrücklich zu formulieren. Falls die Dokumentation keine diesbezüglichen Informationen beinhaltet, kann davon ausgegangen werden, dass Argumentprüfungen vorgenommen werden. Möglicherweise wird jedoch aus Performance-Gründen auf solche Prüfungen verzichtet.
- Bereinige sämtliche Seiteneffekte beim Auslösen von Ausnahmen. Anwendercode sollte davon ausgehen können, dass beim Auftreten einer Ausnahme keine Seiteneffekte stattgefunden haben. Falls beispielsweise die Methode `Hashtable.Insert` eine Ausnahme auslöst, kann der Aufrufer davon ausgehen, dass das Element nicht eingefügt wurde und sich die Hash-Tabelle nach wie vor in einem konsistenten Zustand befindet.

8.2 Standard-Ausnahmeklassen

Die nachfolgende Tabelle listet die Standardausnahmen aus, die von der Laufzeitumgebung zur Verfügung gestellt werden, sowie die Bedingungen, unter welchen eine abgeleitete Ausnahmeklasse verwendet werden soll.

Ausnahmeklasse	Basisklasse	Beschreibung	Beispiel
Exception	Object	Basisklasse für alle Ausnahmen.	Keines. Verwende eine abgeleitete Klasse.
SystemException	Exception	Basisklasse für alle von der Laufzeitumgebung ausgelösten Ausnahmen.	Keines. Verwende eine abgeleitete Klasse.
IndexOutOfRangeException	SystemException	Wird nur von der Laufzeitumgebung ausgelöst, wenn ein Array ungültig indiziert wird.	arr[arr.Length+1]
NullReferenceException	SystemException	Wird nur von der Laufzeitumgebung ausgelöst, wenn ein null-Objekt referenziert wird.	object o = null; o.ToString();
InvalidOperationException	SystemException	Wird von Methoden ausgelöst, wenn sie sich in einem unpassenden Zustand befinden.	Enumerator.GetNext(), nachdem eine Element aus der Sammlung entfernt wurde.
ArgumentException	SystemException	Basisklasse für alle Argumentausnahmen.	Keines. Verwende eine abgeleitete Klasse.
ArgumentNullException	ArgumentException	Wird von Methoden ausgelöst, wenn ein Argument fälschlicherweise null ist.	string s = null; "otto".IndexOf(s);
ArgumentOutOfRangeException	ArgumentException	Wird von Methoden ausgelöst, wenn sich ein Argument nicht im gütligen Wertbereich befindet.	string s = "abc"; char c = s[9];
ExternalException	SystemException	Basisklasse für Ausnahmen, die mit der Welt außerhalb der verwalteten Laufzeitumgebung zusammenhängen.	Keines. Verwende eine abgeleitete Klasse.
COMException	ExternalException	Ausnahme, die HRESULT-Informationen bezüglich COM kapselt.	In COM-Interop verwendet.

Ausnahmeklasse	Basisklasse	Beschreibung	Beispiel
SEHException	ExternalException	Ausnahme, die die strukturierte Win32-Ausnahmebehandlung kapselt.	In nicht verwaltetem Code-Interop verwendet.

8.3 Ausnahmeverkettung

Fehler, die auf der Komponentenebene auftreten, sollen zur Auslösung einer Ausnahme führen, die für den Anwendercode bedeutsam ist. Im folgenden Beispiel ist die Fehlermeldung für Anwendercode, der versucht, Daten aus einem Strom zu lesen, bedeutsam.

```
public class TextReader
{
   public string ReadLine()
   {
      try
      {
         // read a line from the stream
      }
      catch (Exception ex)
      {
         throw new IOException("Could not read from stream.", ex);
      }
   }
}
```

9 Richtlinien für Arrays

9.1 Vergleich Arrays ← → Sammlungen

Bei der Entwicklung von Klassenbibliotheken sind oft schwierige Entscheidungen zu treffen, wann Arrays und wann Sammlungen zum Einsatz kommen sollten. Obwohl beiden Arten ähnliche Anwendungsmodelle zugrundeliegen, haben sie unterschiedliche Charakteristika bezüglich der Ausführungskomplexität. Generell gilt: Wenn die Manipulation der Datensammlung unterstützt wird, ist die Sammlung dem Array vorzuziehen.

9.2 Verwendung von Arrays

Gebe keine interne Instanz eines Arrays zurück, da der Anwendercode dadurch in der Lage ist, die Elemente im Array zu ändern. Das folgende Code-Beispiel zeigt, wie der Anwendercode interne Daten modifizieren kann, ohne dass es hierfür eine schreibbare Eigenschaft gibt.

```
public class ExampleClass
{
   public class Path
   {
      private static char[] s_badChars = new string[] { '\"', '<', '>' };

      public static char[] GetInvalidPathChars()
      {
         return s_badChars;
      }
   }

   public static void Main()
   {
      // Wie erwartet:
      foreach (char c in Path.GetInvalidPathChars())
         Console.Write(c);
      Console.WriteLine();

      // Internes Array manipulieren:
      Path.GetInvalidPathChars()[0] = 'A';
      Path.GetInvalidPathChars()[1] = 'A';
      Path.GetInvalidPathChars()[2] = 'A';

      // Entgegen allen Erwartungen:
      foreach (char c in Path.GetInvalidPathChars())
         Console.Write(c);
      Console.WriteLine();
   }
}
```

Das Problem muss dadurch behoben werden, indem das Array vor der Rückgabe dupliziert wird.

```
public static char[] GetInvalidPathChars()
{
   return (char[])s_badChars.Clone();
}
```

Die Deklaration von `s_badChars` als `readonly` ist nicht wirksam, da dadurch zwar das Array schreibgeschützt ist, die Elemente aber dennoch veränderbar sind.

9.3 Inidizierte Eigenschaften in Sammlungen

Indizierte Eigenschaften sollten nur in Sammlungsklassen oder –schnittstellen definiert werden. Ebenso sollten Methodenmuster wie `Add`, `Remove`, `Count` nur in solchen Klassen verwendet werden, da sie ein Signal für den Anwender sind, dass es sich hierbei um eine Sammlung handelt.

9.4 Eigenschaften mit Array-Rückgabe

Vermeide es, Eigenschaften zu definieren, die ein Array zurückgeben, da sie möglicherweise extrem ineffizient sind. Wenn die oben beschriebenen Regeln bezüglich Duplizierung vor Rückgabe eingehalten werden, wird folgender Code 2^n+1 Kopien des Arrays erzeugen, ohne dass dies dem Anwender der Eigenschaft klar ist:

```
for (int i = 0; i < Path.InvalidPathChars.Count; i++)
   DoSomething(Path.InvalidPathChars[i]);
```

In einem solchen Fall ist die Verwendung einer Sammlung unbedingt vorzuziehen. Alternativ kann die Eigenschaft auch durch eine Methode ersetzt werden, was dem Anwender signalisiert, dass es sich hierbei möglicherweise um eine speicherteure Operation handelt.

9.5 Rückgabe leerer Arrays

Eigenschaften, die `String` oder `Array` zurückliefern, sollten niemals null zurückgeben, da dies für den Anwendercode erhöhten Prüfungsaufwand bedeutet. Die Rückgabe eines leeren Strings (`""`) oder eines Arrays mit null Elementen ist in jedem Fall vorzuziehen.

10 Richtlinien für Operatorüberladung

Dieses Kapitel beschreibt Richtlinien im Zusammenhang mit der Überladung von Operatoren.

- Verwende Operatorenüberladung nur dann, wenn das Resultat der Operation offensichtlich ist. Beispielsweise macht es Sinn, den Subtraktionsoperator für zwei `Time`-Werte, resultierend in einem `TimeSpan`-Wert, zu überladen. Hingegen ist nicht angemessen, den Oder-Operator zu überladen, um die Vereinigung zweier Datenbankabfragen zu erzeugen, oder den Schiebe-Operator, um in einem Datenstrom zu schreiben.
- Überlade Operatoren stets symmetrisch. Wenn beispielsweise der Gleichheitsoperator überladen wird, muss auch der Ungleichheitsoperator überladen werden.
- Stelle stets alternative Methoden zur Verfügung, da bei weitem nicht alle .NET-fähigen Programmiersprachen Operatorüberladung unterstützen. Beispiel:

  ```
  public struct DateTime
  {
     public static TimeSpan operator-(DateTime t1, DateTime t2) { }
     public static TimeSpan Subtract(DateTime t1, DateTime t2) { }
  }
  ```

  Diese Regel existiert deshalb, da die in der Automation Platform befindlichen Schnittstellen kundenseitig auch von anderen .NET-Programmiersprachen bedient werden können, die ihrerseits keine Operatorüberladung unterstützen.
- Die Deklaration impliziter Konvertierungsoperatoren ist zu vermeiden.

11 Richtlinien für die Implementierung von Equals und dem Gleichheitsoperator

Die folgenden Regeln gelten für die Implementierung von `Equals` und dem Gleichheitsoperator (==).

11.1 Allgemeine Regeln

- Implementiere stets `GetHashCode`, wenn `Equals` implementiert wird.
- Überschreibe stets `Equals`, wenn der Gleichheitsoperator == überladen ist, und sorge dafür, dass beide Implementierung dasselbe tun.
- Überschreibe stets `Equals`, wenn die Klasse `IComparable` implementiert.
- Bedenke die Überladung aller Vergleichsoperatoren (==, !=, <, >, <=, >=), wenn die Klasse `IComparable` implementiert.
- Löse innerhalb von `Equals`, `GetHashCode` und dem Gleichheitsoperator == niemals eine Ausnahme aus.

11.2 Implementierung des Gleichheitsoperators bei Werttypen

- In den meisten Programmiersprachen gibt es keine Standardimplementierung des Gleichheitsoperators für Werttypen. Deshalb sollte == stets überladen werden, wenn Gleichheit bedeutsam ist.
- Typischerweise sollte `Equals` für Werttypen implementiert werden, da die geerbte Implementierung in `System.ValueType` unzureichend ist.
- Implementiere == immer dann, wenn `Equals` implementiert ist.

11.3 Implementierung des Gleichheitsoperators bei Referenztypen

- C# bietet eine Standardimplementierung des Gleichheitsoperators für Referenztypen. Aus diesem Grunde sollte == typischerweise nicht implementiert werden, auch dann, wenn `Equals` implementiert ist.
- Implementiere ==, wenn es sich bei dem Typ und einen Datentyp wie `Point`, `String`, `BigNumber` o.ä. handelt. Insbesondere wenn der Additions- und Subtraktionsoperator überladen wird, sollte auch der Gleichheitsoperator überladen werden.

12 Richtlinien für Typumwandlungen

Dieses Kapitel beschreibt Richtlinien dafür, dass eine Klasse explizit implementierte Typumwandlungsroutinen zur Verfügung stellt.

- Erlaube keine impliziten Typumwandlungen, die verlustbehaftet sind. Beispielsweise sollte es keine implizite Typumwandlung von `Double` nach `Int32` geben. Die implizite Typumwandlung von `Int32` nach `Double` ist statthaft.
- Löse in impliziten Typumwandlungen keine Ausnahmen aus, da diese sehr schwer zu diagnostizieren sind.
- Stelle nur Typumwandlungen zur Verfügung, die auf dem gesamten Objekt arbeiten. Der umgewandelte Werte soll das gesamte Objekt repräsentieren, nicht nur ein einzelnes Feld darin. Beispielsweise ist es nicht gestattet, für eine Klasse `Button` eine Umwandlung nach `String` zur Verfügung zu stellen, die den Text des Knopfes zurückliefert.
- Erzeuge bei Typumwandlungen keinen semantisch verschiedenen Wert. Einerseits ist es statthaft, eine `TimeSpan` in einen `Int32` umzuwandelt, da auch dieser eine Zeitspanne repräsentiert, andererseits macht es keinen Sinn, einen Dateinamen wie `"c:\mybitmap.gif"` in ein `Bitmap`-Objekt umzuwandeln.
- Wandle nicht zwischen unterschiedlichen Wertdomänen um. Beispielsweise sind Zahlen und Strings unterschiedliche Domänen. Es ist sinnvoll, dass ein `Int32` in ein `Double` umgewandelt werden kann, es ist aber nicht sinnvoll, ein `Int32` in einen `String` umzuwandeln, da es sich hierbei um verschiedene Wertdomänen handelt.

13 Gängige Design-Muster

Dieses Kapitel beschreibt häufig anzutreffende Design-Muster, wie sie in Klassenbibliotheken verwendet werden sollen.

13.1 Implementierung von Finalize und Dispose zur Freigabe nicht-verwalteter Ressourcen

13.1.1 Allgemeines Vorgehen

Objekte kapseln häufig Ressourcen, die nicht von der .NET-Laufzeitumgebung verwaltet werden, wie beispielsweise Fenster-Handles (`HWND`) und Datenbankverbindungen. Aus diesem Grund soll sowohl eine explizite als auch eine implizite Möglichkeit bestehen, diese Ressourcen freizugeben. Implementiere hierzu die geschützte `Finalize`-Methode (in C# über die Destruktor-Syntax). Der Garbage Collector ruft diese Methode auf, wenn es keine Referenz mehr auf das betreffende Objekt gibt.

In manchen Fällen ist es sinnvoll, dem Anwender auch eine explizite Möglichkeit zu geben, solche externen Ressourcen freizugeben, ehe der Garbage Collector das Objekt freigibt. Dies ist immer dann der Fall, wenn eine solche externe Ressource knapp verfügbar oder teuer ist. Hierzu wird die Methode `Dispose` der `IDisposable`-Schnittstelle implementiert. Der Anwendercode soll diese Methode aufrufen, wenn er das Objekt nicht mehr benötigt; dieser Aufruf kann auch stattfinden, wenn es noch andere Referenzen auf dasselbe Objekt gibt.

Beachte, dass selbst wenn eine `Dispose`-Methode implementiert wird, zusätzlich auch die `Finalize`-Methode (in C# über die Destruktor-Syntax) implementiert werden muss, die auch dann automatisch aufgerufen wird, wenn es der Anwendercode versäumt, `Dispose` explizit aufzurufen. Auf diese Weise können Ressourcenlecks vermieden werden.

```
public class Base : IDisposable
{
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   protected virtual void Dispose(bool bDisposing)
   {
      if (bDisposing)
      {
         // Verwaltete Ressourcen freigeben.
      }
      // Nicht verwaltete Ressourcen freigeben.
   }

   ~Base()
   {
      Dispose(false);
   }
}

public class Derived : Base
{
   protected override void Dispose(bool bDisposing)
   {
      if (bDisposing)
      {
         // Verwaltete Ressourcen freigeben.
      }
      // Nicht verwaltete Ressourcen freigeben.
      base.Dispose(bDisposing);
   }

   // Sowohl Dispose() als auch Destruktor werden geerbt.
}
```

13.1.2 Umbenennung von Dispose

In manchen Szenarien ist es möglicherweise sinnvoll, dass ein von `Dispose` verschiedener Name besser verständlich ist. Ein klassisches Beispiel ist eine Klasse, die Dateifunktionalität kapselt; hier macht der Name `Close` mehr Sinn wie `Dispose`. In diesem Fall kann eine Methode mit dem gewüschten Namen implementiert werden, die einfach `Dispose` aufruft. Diese Methode soll nicht-virtuell sein, so dass abgeleitete Klassen sie nicht überschreiben können.

```
public void Close()
{
   Dispose();
}
```

13.1.3 Finalize (in C# Destruktoren)

- Implementiere einen Destruktor nur für Klassen, die ihn auch wirklich benötigen. Finalisierung ist eine teure Operation.
- Falls ein Destruktor implementiert wird, sollte die Klasse auch `IDisposable` implementieren. Dadurch ist es dem Anwendercode möglich, die teure Finalisierung durch expliziten Aufruf von `Dispose` zu vermeiden.
- Der Destruktor einer Klasse soll alle externen Ressourcen freigeben, die das Objekt besitzt. Es darf keine anderen Objekte referenzieren.

13.1.4 Dispose

- Implementiere das Dispose-Design-Muster für Klassen, welche Ressourcen kapseln, die explizit freigegeben werden müssen. Anwendercode kann diese Ressourcen freigeben, indem er die öffentliche Methode `Dispose` aufruft.
- Gib alle Ressourcen in der `Dispose`-Methode frei, die das Objekt besitzt.
- Nachdem `Dispose` für ein Objekt aufgerufen wurde, sollte für diese Instanz mittels der Methode `GC.SuppressFinalize` verhindert werden, dass der Destruktor aufgerufen wird.
- Rufe, falls vorhanden, die Methode `Dispose` der Basisklasse auf.
- Gehe nicht davon aus, dass der Anwendercode `Dispose` aufruft. Nicht verwaltete Ressourcen müssen auch im Destruktor freigegeben werden, für den Fall, dass `Dispose` nicht explizit aufgerufen wird.
- Instanzmethoden einer Klasse mit Ausnahme der Methode `Dispose` sollen eine `ObjectDisposedException` auslösen, wenn sie nach einem `Dispose` aufgerufen werden. Die `Dispose`-Methode selbst muss mehrmals aufrufbar sein, ohne dass eine Ausnahme ausgelöst wird.
- Rufe in der `Dispose`-Methode auch die `Dispose`-Methoden aller referenzierten Objekte auf, falls vorhanden. Beispiel: Wenn man ein Objekt vom Typ `TextReader` erzeugt, werden implizit ohne Wissen des Anwenders auch Objekte der Typen `Stream` und `Encoding` erzeugt, die ihrerseits wieder nicht-verwaltete Ressourcen belegen. Wird nun `Dispose` auf dem `TextReader` aufgerufen, muss diese Methode selbst auch die `Dispose`-Methoden von `Stream` und `Encoding` aufrufen.
- Ein Objekt sollte nach einem Aufruf von `Dispose` nicht mehr verwendbar sein. Das automatische Wiederherstellen eines gültigen Zustands ist in der Regel eine schwierige Aufgabe.
- Sorge dafür, dass `Dispose` mehrmals auf der selben Instanz aufgerufen werden kann. Nach dem ersten Aufruf sollte die Implementierung nichts mehr tun.

13.2 Implementierung der Methode Equals

Detaillierte Informationen über den Gleichheitsoperator befinden sich in Kapitel 11.

13.2.1 Allgemeines Vorgehen

- Überschreibe die Methode `GetHashCode`, so dass Objekte der Klasse korrekt in einer Hash-Tabelle verwendet werden können.
- Löse in der Implementierung einer `Equals`-Methode keine Ausnahme aus. Gib `false` zurück, falls ein `null`-Argument übergeben wird.
- Folgende Regeln gelten stets:
  - `x.Equals(x)` gibt `true` zurück.
  - `x.Equals(y)` gibt denselben Wert wie `y.Equals(x)` zurück.
  - `x.Equals(y) && y.Equals(z)` ist dann und nur dann `true`, falls `x.Equals(z)` gleich `true` ist.
  - Wiederholte Aufrufe von `x.Equals(y)` liefern stets denselben Wert, solange `x` und `y` nicht modifiziert werden (Seiteneffektfreiheit!).
  - `x.Equals(null)` gibt `false` zurück.
- Für einige Arten von Objekten ist es wünschenswert, dass `Equals` auf Wertgleichheit statt auf Referenzgleichheit testet. Solche Implementation von `Equals` liefern `true` zurück, wenn die zwei Objekte denselben Wert haben, selbst dann, wenn sie nicht dieselbe Instanz sind. Die Definition, was den Wert eines Objekts repräsentiert, liegt beim Implementierer der Klasse, ist

aber typischerweise alle oder einige Instanzfelder des Objekts. Beispielsweise ist der Wert eines Strings die Zeichen innerhalb des Strings; die zugehörige `Equals`-Methode liefert dann `true`, wenn zwei beliebige String-Instanzen dieselben Zeichen in derselben Reihenfolge beinhalten.

- Wenn die Basisklasse bereits eine `Equals`-Methode im Sinne von Wertgleichheit implementiert, sollen abgeleitete Klassen auch die Basisklassenmethode aufrufen.
- Falls der Gleichheitsoperator (`==`) für eine bestimmte Klasse überladen ist, soll auch eine `Equals`-Methode implementiert werden, die dasselbe Resultat liefert.
- Bei Werttypen macht es möglicherweise Sinn, `Equals` zu überschreiben, um eine bessere Performance im Vergleich zur Standardimplementierung in `System.ValueType` zu erzielen. In diesem Fall sollte auch der Gleichheitsoperator (`==`) überladen werden.
- Bei Referenztypen kann Equals überschrieben werden, wenn die Klasse einem Basistyp ähnlich ist, wie z.B. `Point`, `String`, `BigNumber` etc. Typischerweise soll der Gleichheitsoperator nicht überladen werden, auch wenn `Equals` überschrieben ist. Eine Ausnahme dieser Regel ist dann gegeben, wenn der Referenztyp sich wie ein Werttyp verhalten soll.
- Wenn ein Typ `IComparable` implementiert, soll `Equals` auf diesem Typ überschrieben werden.

13.2.2 Beispiel 1

Das folgende Beispiel enthält zwei Aufrufe der Standardimplementierung von `Equals`.

```
class SampleClass
{
   public static void Main()
   {
      Object obj1 = new Object();
      Object obj2 = new Object();
      Console.WriteLine(obj1.Equals(obj2));
      obj1 = obj2;
      Console.WriteLine(obj1.Equals(obj2));
   }
}
```

Die Ausgabe dieses Codes ist wie folgt:

```
false
true
```

13.2.3 Beispiel 2

Das folgende Beispiel zeigt eine Klasse `Point`, die die `Equals`-Methode überschreibt, um auf Wertgleichheit zu prüfen, und eine davon abgeleitete Klasse `Point3D`. `Point` ruft die Basisklassenvariante von `Equals` (`Object.Equals`) nicht auf, da diese keine Wertsemantik hat. `Point3D.Equals` hingegen ruft `Point.Equals` auf, da hier Wertsemantik realisiert ist.

```
class Point
{
   private int _x;
   private int _y;

   public override bool Equals(object obj)
   {
      if (obj == null || GetType() != obj.GetType())
         return false;
      Point p = (Point)obj;
      return _x == p._x && _y == p._y;
   }

   public override int GetHashCode()
   {
      return _x ^ _y;
   }
}

class Point3D : Point
{
   private int _z;

   public override bool Equals(object obj)
   {
      return base.Equals(obj) && _z == ((Point3D)obj)._z;
   }

   public override int GetHashCode()
   {
      return base.GetHashCode() ^ _z;
   }
}
```

Die Methde `Point.Equals` prüft, ob das Argument nicht null und vom gleichen Typ wie dieses Objekt ist. Falls einer dieser Prüfungen fehlschlägt, gibt die Methode `false` zurück. Die Methode verwendet die `Object.GetType`-Methode um den Laufzeittyp eines Objekts festzustellen. Beachte, dass hier nicht `typeof` verwendet werden kann, da dieser den statischen Typ (zur Compilezeit, in diesem Fall also `Object`) zurückgibt. Falls die Methode eine Prüfung der Art `obj is Point` verwenden würde, wäre die Überprüfung auch dann `true`, wenn `obj` von einem abgeleiteten Typ (in diesem möglicherweise vom Typ `Point3D`) wäre. Nachdem die Typgleichheit festgestellt wurde, wandelt die Methode `obj` in den Typ `Point` um und vergleicht die Instanzvariablen der beiden Objekte.

In `Point3D.Equals` wird zu allererst die geerbe `Equals`-Method aufgerufen. Diese geerbte `Equals`-Methode prüft, dass `obj` nicht `null` und vom selben Laufzeittyp ist und dass die geerbten Instanzvariablen gleich sind. Nur wenn hieraus `true` resultiert, wird die eigene Instanzvariable verglichen. Die Typumwandlung auf `Point3D` ist hier gefahrlos möglich, da der Code nur durchlaufen wird, wenn die geerbte Methode `Equals` `true` zurückliefert, was bedeutet, dass `obj` exakt vom Typ `Point3D` sein muss.

13.2.4 Beispiel 3

Im vorherigen Beispiel wurde der Gleichheitsoperator (`==`) verwendet, um einzelne Instanzvariablen miteinander zu vergleichen. In manchen Fällen ist es jedoch nötig, innerhalb von `Equals` auch die Instanzvariablen mit `Equals` zu vergleichen, wie im folgenden Beispiel:

```
class Rectangle
{
   Point _a;
   Point _b;

   public override bool Equals(object obj)
   {
      if (obj == null || GetType() != obj.GetType())
         return false;
      Rectangle r = (Rectangle)obj;
      return a.Equals(r.a) && b.Equals(r.b);
   }

   public override int GetHashCode()
   {
      return a.GetHashCode() ^ b.GetHashCode();
   }
}
```

13.2.5 Beispiel 4

Wenn ein Typ den Gleichheitsoperator (`==`) überlädt, sollte er auch die `Equals`-Methode überschreiben, um die gleiche Funktionalität zur Verfügung zu stellen. Typischerweise wird dies dadurch erreicht, indem die `Equals`-Methode den überladenen Gleichheitsoperator verwendet, wie im folgenden Beispiel:

```
struct Complex
{
   double _dReal;
   double _dImag;

   public override bool Equals(object obj)
   {
      return obj is Complex && this == (Complex)obj;
   }

   public override int GetHashCode()
   {
      return _dReal.GetHashCode() ^ _dImag.GetHashCode();
   }

   public static bool operator==(Complex x, Complex y)
   {
      return x._dReal == y._dReal && x._dImag == y._dImag;
   }

   public static bool operator!=(Complex x, Complex y)
   {
      return !(x == y);
   }
}
```

Da `Complex` ein Werttyp ist, kann keine Klasse von `Complex` abgeleitet werden. Aus diesem Grunde ist die Typprüfung mit dem `is`-Operator zulässig.

13.3 Verwendung von Rückruffunktionen

Delegate, Schnittstellen und Ereignisse bieten Rückruffunktionalität. Jeder dieser Arten hat auf die ein oder andere Weise Vorteile gegenüber den anderen, je nach Einsatzzweck.

13.3.1 Ereignisse

Verwende ein Ereignis, wenn folgendes gilt:

- Eine Methode meldet sich gleich zu Beginn der Lebenszeit eines Objekts auf dessen Rückruf an.
- Mehrere Objekte möchten durch den Rückruf benachrichtigt werden.
- Anwender sollen im visuellen Designer eine einfache Möglichkeit haben, sich auf den Rückruf anzumelden.

13.3.2 Delegate

Verwende ein Delegat, wenn folgendes gilt:

- Ein Konstrukt im Stile eines klassischen C-Funktionszeigers ist erwünscht.
- Es soll eine einzelne Rückruffunktion geben.
- Die Registration soll direkt beim Aufruf der rückrufenden Methode erfolgen.

13.3.3 Schnittstellen

Wenn die Rückruffunktionalität komplexer wird, ist eine Schnittstelle anzuraten.

14 Richtlinien für Multithreading

Die wichtigste Regel lautet:

Vermeide Multithreading, wann immer möglich. Es ist insbesondere darauf beachten, dass Funktionalität, die mit der Benutzeroberfläche zusammenhängt, prinzipbedingt nur im Haupt-Thread ablaufen darf. Auch in allen anderen Fällen ist sorgfältig abzuwägen, ob die extrem erschwerte Wartbarkeit von Code den Einsatz von Multithreading rechtfertigt.

Sollte Multithreading dennoch nötig sein, sind folgende Regeln zu beachten:

- Vermeide es, statische Methoden zur Verfügung zu stellen, die den statischen Zustand verändern. In einem klassischen Server-Szenario wird der statische Zustand zwischen verschiedenen Anfragen geteilt, was bedeutet, dass dieser Code von vielen Threads gleichzeitig durchlaufen wird. Dies öffnet ein weites Feld möglicher Fehler. Es ist daher anzuraten, ein Design-Muster zu wählen, in dem die Daten in Instanzen gekapselt werden, die nicht zwischen verschiedenen Anfragen geteilt werden.
- Der statische Zustand muss thread-sicher sein.
- Der Instanzzustand braucht nicht thread-sicher zu sein. Standardmäßig sollen Klassenbibliotheken nicht thread-sicher sein, da die Einführung von Locks zu verminderter Performance, erhöhter Lock-Konkurrenz und der Möglichkeit von Deadlocks führt. In typischen Applikationsmodellen wird Code von einem einzelnen Thread ausgeführt, so dass kaum Bedarf für Thread-Sicherheit besteht. Aus diesem Grunde sind sowohl das .NET-Framework als auch die CoDeSys Automation Alliance in weiten Teilen nicht thread-sicher. In den Fällen, in denen eine thread-sichere Variante zur Verfügung gestellt werden soll, sollte eine statische `Synchronized`-Methode implementiert werden, die eine thread-sichere Instanz eines Typs zurückliefert. Als Beispiel seien die Methoden `System.Collections.ArrayList.Synchronized` und `System.Collections.ArrayList.IsSynchronized`.
- Vorsicht bei Methodenaufrufen in einem gelockten Bereich! Deadlocks können auftreten, wenn eine statische Methode in Klasse A eine statische Methode in Klasse B und umgekehrt aufruft. Falls sowohl A und B ihre statischen Methoden synchronisieren, ist ein Deadlock sehr wahrscheinlich. Dennoch tritt dieser möglicherweise nur unter hoher Belastung auf, ist also schwer zu diagnostizieren.
- Mit Performance-Problemen ist zu rechnen, wenn eine statische Methode in einer Klasse eine statische Methode in derselben Klasse aufruft. Hier besteht die Gefahr, dass eine große Menge an redundanten Synchronisationen stattfinden.
- Beachte, dass die `System.Threading.Interlocked`-Klasse in vielen Szenarien dem `lock`-Block vorzuziehen ist, da die Performance hier meist deutlich besser ist.

  Beispiel 1: Schlecht:
  ```
  lock(this)
  {
     _value++;
  }
  ```

  Gut:
  ```
  System.Treading.Interlocked.Increment(_value);
  ```

  Beispiel 2: Schlecht:
  ```
  if (x == null)
  {
     lock(this)
     {
        if (x == null)
           x = y;
     }
  }
  ```

  Gut:
  ```
  System.Threading.Interlocked.CompareExchange(ref x, y, null);
  ```

- Vermeide die Notwendigkeit für Synchronisation soweit wie möglich. Oftmals kann ein Algorithmus dergestalt angepasst werden, dass er Race Conditions tolerieren kann, anstatt sie zu verhindern.

15 Richtlinien zur Oberflächenerstellung

Dieses Kapitel enthält Richtlinien, die bei der Erstellung von Oberflächenelementen verbindlich gelten. Grundlage für diese Regeln ist die Verwendung des in Visual Studio 2005 integrierten visuellen Oberflächendesigners.

15.1 Erstellung von modalen Dialogen

15.1.1 Dialogeigenschaften

Für einen modalen Dialog sind folgende Einstellungen zu setzen, um ein konformes Aussehen und Verhalten zu gewährleisten (es werden hier nur die Eigenschaften aufgezählt, die vom Default-Wert abweichen):

Eigenschaft	Wert	Bemerkung
Font	Tahoma; 8,25pt	
FormBorderStyle	FixedDialog	Es kann auch Sizable gewählt werden. In diesem Fall ist jedoch unbedingt darauf zu achten, dass sich die im Dialog enthaltenen Elemente entsprechend anpassen.
Text	Nach Bedarf	
(Name)	Nach Bedarf	
Language	(Default)	Hier niemals eine andere Einstellung machen, da sonst unser Lokalisierungsprozess nicht mehr funktioniert!
Localizable	True	Hier niemals eine andere Einstellung machen, da sonst der Dialog nicht in andere Sprachen übersetzt werden wird!
Size	Nach Bedarf, aber nicht größer als 800;600	
StartPosition	CenterParent	
AcceptButton	Diesen Wert auf den „OK“- oder „Schließen“-Button des Dialogs setzen, sofern vorhanden	Andernfalls funktioniert das Betätigen der „Return“-Taste zum Bestätigen des Dialogs nicht.
CancelButton	Diesen Wert auf den „Cancel“- oder „Schließen“-Button des Dialogs setzen, sofern vorhanden	Andernfalls funktioniert das Betätigen der „ESC“-Taste zum Abbrechen des Dialogs nicht.
MinimizeBox	False	
MaximizeBox	False	
ShowIcon	False	
ShowInTaskBar	False	

15.1.2 Positionierung von Kontrollelementen

Folgende grundsätzliche Regeln gelten bezüglich der Positionierung von Kontrollelementen:

- Verwende die vom visuellen Designer zur Verfügung gestellten Snap-Lines. Diese sorgen dafür, dass der Abstand zum Rand sowie die Abstände untereinander den gängigen Windows-Richtlinien entsprechen.
- Ordne die Buttons „OK“ und „Cancel“ horizontal nebeneinander an. Sie sind grundsätzlich rechtsbündig an unteren Fensterrand zu setzen und an der Snap-Line einzurasten. Eine

vertikale Anordnung in der oberen rechten Fensterecke ist nicht gestattet. Sollte ein Dialog nur über einen „Close“ Button verfügen, so ist auch dieser in der unteren rechten Fensterecke zu platzieren.

- Buttons haben grundsätzlich die Größe 75; 23, es sei denn, der Text würde nicht hineinpassen. In diesem Fall ist eine horizontale Vergrößerung erlaubt.
- Verwende die AutoSize-Option, wenn es ein Kontrollelement gestattet. Auf diese Weise kann sichergestellt werden, dass sich das Element auch an anderssprachige Texte anpasst.
- Prüfe nach Fertigstellung bzw. nach Veränderung des Oberflächendesigns mittels des Menüpunktes *View* | *Tab Order*, ob die Tabulatorreihenfolge für die Elemente sinnvoll ist.

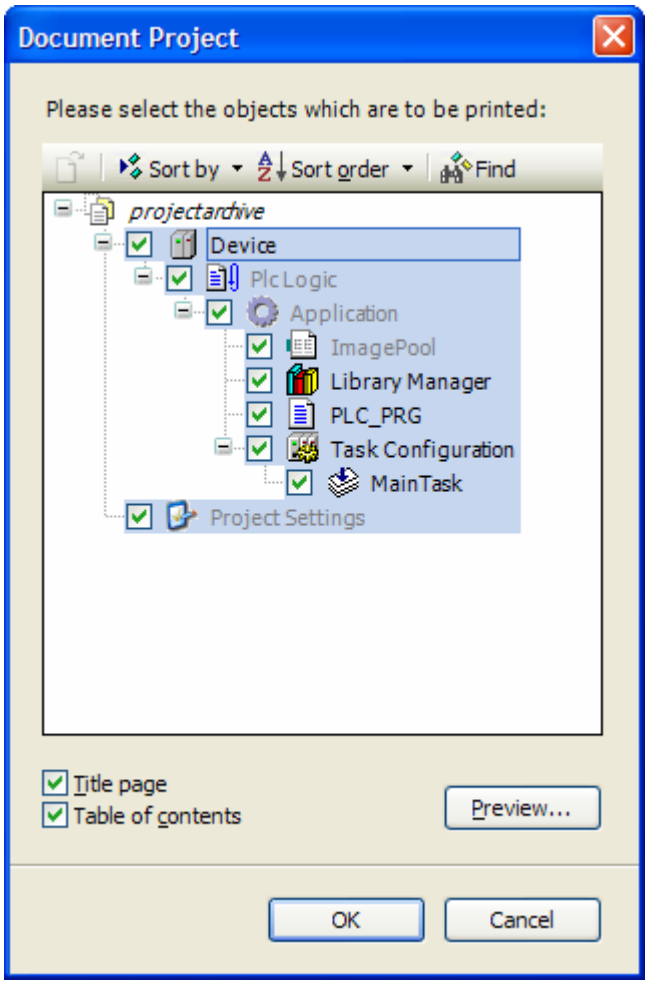
15.1.3 Schreibweise von Texten

- Verwende keine Titlecase-Notation. Also “Library repository location” statt “Library Repository Location”.
- Schreibe “OK” statt “Ok”.
- Verwende einen abschließenden Doppelpunkt für Bezeichnungstexte.

15.1.4 Sonstiges

- Verwende zur Darstellung von tabellarischen, listenartigen und baumartigen Datentypen wenn möglich das Control `TreeTableView`, zu finden im GAC-Assembly `Controls` im Namespace `_3S.CoDeSys.Controls.Controls`.
- Bei Comboboxen, die über einen festen Satz an auswählbaren Einträgen verfügen, ist die Eigenschaft `DropDownStyle` auf `DropDownList` zu setzen.

Beispiel für einen Dialog, der gemäß diesen Normen erstellt wurde:



15.2 Erstellung andersartiger Oberflächenelemente

Die Regeln aus Kapitel 15.1 sind – falls anwendbar – sinngemäß zu verwenden.

15.3 Meldungstexte

Für die Anzeige von Meldungstexten gelten besondere Regeln bezüglich der aufzurufenden Methoden sowie der Formulierung der Texte:

- Verwende die Methoden aus dem Interface `IMessageServiceX`. Die Instanz dieser Schnittstelle ist über `SystemInstances.Engine.MessageService` zu erreichen. Verwende nicht die Methodengruppe `MessageBox.Show()` aus Windows Forms, da die Automation Platform jederzeit in einem oberflächenlosen Kontext funktionieren muss (automatischer Test, Konsolenapplikationen etc.). Solche speziellen Tools arbeiten mit einer passenden Implementierung der `IMessageServiceX`-Schnittstelle.
- Eine Fehlermeldung (Icon: rotes Kreuz) ist zu verwenden, wenn ein Vorgang nicht abgeschlossen werden kann. Eine Warnungsmeldung (Icon: gelbes Ausrufungszeichen) ist zu verwenden, wenn der Vorgang fortgesetzt werden kann, evtl. mit Einschränkungen, die im Meldungstext zu formulieren ist. Eine Informationsmeldung (Icon: blaue Sprechblase) ist zu verwenden, um nach einem erfolgreichen Vorgang den Benutzer über die Beendigung zu informieren.
- Verwende ganzen Sätze. Statt „No name!“ besser „Please specify a name.” oder auch “You must enter a name.”
- Gehe sparsam mit Ausrufungszeichen um. Meldungen wie „Download failed!!!“ sind nicht erlaubt. Hier besser: „The download failed due to the following reason: …”.

Literaturverzeichnis

[1] .NET Framework Developer's Guide – Design Guidelines for Class Library Developers
http://msdn2.microsoft.com/en-us/library/czefa0ke(VS.71).aspx

[2] .NET Framework Developer's Guide – Common Type System
http://msdn2.microsoft.com/en-us/library/aa720708(VS.71).aspx

[3] .NET Framework Developer's Guide – Value Types
http://msdn2.microsoft.com/en-us/library/aa735672(VS.71).aspx

Appendix A: Änderungshistorie

Version	Beschreibung	Bearbeiter	Datum
0.1	Erstellung	KeK	26.07.2007
1.0	Review und Freigabe	DS	27.07.2007
1.1	• Kapitel 2: Überarbeitung für „`case`-Marken einrücken“ und „`break`-Statements nicht ausrücken“ • Kapitel 3.1.3: Klarere Formulierung: Unterstriche sind nur dann zu unterlassen, wenn die Regeln für Feldnamen gemäß Kap. 3.11 nicht zutreffen (Feldnamen verfügen immer über einen Unterstrich). • Kapitel 4.2: Beispielklasse `MouseEventArgs` überarbeitet, um den Codierrichtlinien bezüglich Feldnamen gerecht zu werden. • Kapitel 9 (vormals Kapitel 8): Begründung für die Regel „Stelle stets alternative Methoden zur Verfügung,…“ eingefügt. • Kapitel 9 (vormals Kapitel 8): Neue Regel eingefügt, dass die Deklaration impliziter Konvertierungsoperatoren zu vermeiden ist. • Neues Kapitel 6 eingeführt: Richtlinien für Anweisungen. Alle nachfolgenden Kapitelnummern verschieben sich entsprechend.	KeK	10.07.2008
1.1	Review und Freigabe	DS	10.07.2008
1.2	Neues Kapitel 14: Richtlinien zur Oberflächenerstellung	KeK	05.09.2008
1.2	Review und Freigabe	DH	05.09.2008
1.3	• Neues Kapitel 2: Compilerfehler und Compilerwarnungen • Neues Kapitel 3.5: Kommentare	KeK	13.08.2010
1.3	Review und Freigabe	DH	13.08.2010