



FAKULTÄT  
FÜR INFORMATIK  
Faculty of Informatics

TECHNISCHE UNIVERSITÄT WIEN

SEMINAR MIT BACHELORARBEIT

---

Vergleich des Konzepts  
*Design by Contract*  
in Eiffel und C#

---

*Autoren:*

Christoph ERKINGER  
Thomas PANI

*Betreuer:*

Ao.Univ.Prof. Dipl.-Ing. Dr.  
Franz PUNTIGAM

27. Oktober 2010

## Zusammenfassung

*Design by Contract* (DbC) wurde von Meyer in Eiffel eingeführt, um die Entwicklung robusterer und zuverlässigerer Software zu ermöglichen. Im .NET Framework 4.0 ist unter dem Namen *Code Contracts* (CC) eine ähnliche Art von Zusicherungen vorhanden.

Im Rahmen dieser Arbeit werden die Umsetzungen des DbC-Prinzips in Eiffel und C#/CC verglichen. Insbesondere betrachten wir im Zusammenhang mit DbC bekannte Spezialfälle, und beurteilen deren Vorkommen und Auswirkungen in den beiden Implementierungen. Außerdem wird ein Performancevergleich angestellt, und die „gemischte“ Verwendung von CC in Eiffel untersucht.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
1.1	Planung . . . . .	3
1.1.1	Motivation . . . . .	3
1.1.2	Methode . . . . .	3
1.1.3	Erwartete Ergebnisse . . . . .	4
1.2	Kapitelzuordnung . . . . .	4
1.2.1	Christoph Erkinge . . . . .	4
1.2.2	Thomas Pani . . . . .	4
<b>2</b>	<b>Literaturüberblick</b>	<b>5</b>
<b>3</b>	<b>Einführung in Design by Contract</b>	<b>6</b>
<b>4</b>	<b>Anwendungsbeispiel: Bankkonto</b>	<b>7</b>
4.1	Allgemeine Funktionen . . . . .	7
4.1.1	Zusicherungen . . . . .	7
<b>5</b>	<b>Spezialfälle</b>	<b>9</b>
5.1	Toolchains . . . . .	9
5.1.1	Eiffel for .NET . . . . .	9
5.1.2	C# und Code Contracts . . . . .	9
5.2	Prestate-Values . . . . .	10
5.2.1	C# . . . . .	10
5.2.2	Eiffel . . . . .	11
5.2.3	Zusammenfassung . . . . .	11
5.3	Rückgabewerte . . . . .	12
5.3.1	C# . . . . .	12
5.3.2	Eiffel . . . . .	12
5.4	Hilfsprozeduren . . . . .	13
5.5	Hilfsprozeduren: Überschreiben . . . . .	13
5.5.1	C# . . . . .	14
5.5.2	Eiffel . . . . .	15
5.6	Hilfsprozeduren: Seiteneffekte . . . . .	15
5.6.1	C# . . . . .	15
5.6.2	Eiffel . . . . .	16
5.7	Hilfsprozeduren: Quantoren . . . . .	16
5.7.1	C# . . . . .	17
5.7.2	Eiffel . . . . .	17
5.8	Hilfsprozeduren: Zusammenfassung und Vergleich . . . . .	18
5.9	Statisches Binden . . . . .	18
5.9.1	C# . . . . .	19
5.9.2	Eiffel . . . . .	20
5.9.3	Zusammenfassung und Vergleich . . . . .	20
5.10	Überprüfung von Invarianten . . . . .	21
5.10.1	C# . . . . .	21

5.10.2 Eiffel . . . . .	22
5.10.3 Zusammenfassung und Vergleich . . . . .	22
5.11 Rekursive und Reflexive Zusicherungen . . . . .	22
5.11.1 Umsetzung in Eiffel . . . . .	22
5.11.2 Unendliche Rekursion . . . . .	23
5.11.3 Technische Umsetzung . . . . .	25
5.11.4 Zusammenfassung und Vergleich . . . . .	26
5.12 Generics . . . . .	26
5.12.1 C# . . . . .	26
5.12.2 Eiffel . . . . .	28
5.12.3 Zusammenfassung und Vergleich . . . . .	29
<b>6 Interoperabilität Eiffel – Code Contracts</b>	<b>30</b>
6.1 Idee . . . . .	30
6.2 Umsetzung . . . . .	30
6.3 Ergebnis . . . . .	31
<b>7 Performancevergleich</b>	<b>32</b>
7.1 Ausgangslage . . . . .	32
7.1.1 Testszenario . . . . .	32
7.2 Ergebnisse . . . . .	34
7.3 Analyse der Ergebnisse . . . . .	36
7.3.1 Inkonsistente Performance in C# . . . . .	36
7.3.2 Schlechte Performance von <i>Eiffel for .NET</i> . . . . .	38
7.4 Fazit . . . . .	38
<b>8 Schlussbemerkungen</b>	<b>39</b>
<b>9 Verwandte Themen &amp; Ausblick</b>	<b>40</b>

# Kapitel 1

## Einleitung

### 1.1 Planung

#### 1.1.1 Motivation

Seit der Einführung von *Design by Contract* (DbC) in Eiffel wird das Konzept auch von einigen anderen Sprachen nativ unterstützt. C# ist allerdings die erste industriell weit verbreitete general-purpose Sprache, die dieses Konzept offiziell unterstützt. Die Vielzahl an verfügbaren externen Tools für ähnlich verwendete Sprachen (z.B. C, C++, Java, Python) zeigt ein grundsätzliches Interesse; unter den Java RFEs<sup>1</sup> ist DbC-Unterstützung an erster Stelle gereiht. [25]

#### 1.1.2 Methode

Wir betrachten bei Eiffel die Toolchain *Eiffel for .NET* von *Eiffel Software* und bei C# den von *Microsoft* bereitgestellten dynamischen Checker (`ccrewrite.exe`) in den folgenden Bereichen:

**DbC-Spezialfälle** Dazu implementieren wir kleine, abgeschlossene Softwareartefakte in jeweils beiden Programmiersprachen und betrachten das Verhalten der Zusicherungen zur Laufzeit. Wir wollen dabei auf folgende Spezialfälle genauer eingehen:

- Generics
- Rekursion
- Statisches Binden
- Zugriff auf Rückgabewerte (im Speziellen `out`-Parameter in C#)
- Zugriff auf Variablenbelegung zum Eintritt in die Prozedur
- Hilfsprozeduren (Überschreiben, Klasseninvarianten-Methode in C#/CC, Seiteneffekte)
- Quantifizierung über mengen-/listenartige Datentypen

Lässt das Laufzeitverhalten nicht eindeutig auf die Implementierung schließen, ziehen wir zusätzliche Tools wie *Ildasm* (ein CIL<sup>2</sup>-Disassembler) oder *.NET Reflector* (ein CIL-Decompiler) heran.

**Performance-Vergleich** Außerdem stellen wir einen Performancevergleich an. Dazu implementieren wir in beiden Sprachen eine Klasse, in der möglichst alle DbC-Eigenschaften aber nur Operationen von konstantem Aufwand vorhanden sind. Anschließend bestimmen wir den Overhead durch oftmaliges Ausführen der Routine und Messen der Ausführungszeit (z.B. mit `System.Diagnostics.Stopwatch`) und vergleichen dabei auch Performanceunterschiede die sich durch das Deaktivieren einzelner Zusicherungsarten soll ergeben könnten.

---

<sup>1</sup>Requests for Enhancements

<sup>2</sup>Common Intermediate Language, früher MSIL

**Eiffel-CC-Interoperabilität** Ein Design-Ziel der *Code Contracts* war, eine sprachagnostische Lösung für alle CLR<sup>3</sup>-Sprachen zu schaffen. Daher möchten wir abschließend evaluieren, ob die Zusicherungen in *Eiffel for .NET* auch mit der neuen CC-Library annotiert und mit `crewrite.exe` überprüft werden können. Falls dies möglich ist, werden wir die Kombination Eiffel/CC in den Performancevergleich aufnehmen.

### 1.1.3 Erwartete Ergebnisse

Aufgrund der unterschiedlichen technischen Umsetzung von DbC (siehe auch [5.1 Toolchains](#)) ist zu erwarten, dass sich C# in einigen Fällen etwas flexibler verhält als Eiffel. Außerdem sind wahrscheinlich Erkenntnisse aus dem Einsatz von DbC in anderen Sprachen in das Design von C#/CC eingeflossen. Unterschiede sind nur im Detail bzw. in der praktischen Anwendbarkeit zu erwarten, da viele der genannten Probleme einerseits in den imperativen und objektorientierten Paradigmen und andererseits in der relativ geringen Ausdrucksstärke der Zusicherungen begründet sind, welche für beide Kandidaten gleichermaßen zutreffen.

Im Performancevergleich erwarten wir, dass C# etwas besser abschneidet, da es als .NET-„Haussprache“ speziell für diese Plattform und die Ausgabe in CIL designet wurde.

Ob die Kombination Eiffel/CC funktioniert, liegt hauptsächlich an der Kompatibilität zwischen CIL-Ausgabe des Eiffel-Compilers und der vom Bytecode-Rewriter `crewrite.exe` akzeptierten Eingabe.

## 1.2 Kapitelzuordnung

### 1.2.1 Christoph Erking

- [4 Anwendungsbeispiel: Bankkonto](#)
- [5.2 Prestate-Values](#)
- [5.6 Hilfsprozeduren: Seiteneffekte](#)
- [5.7 Hilfsprozeduren: Quantoren](#)
- [5.11 Rekursive und Reflexive Zusicherungen](#)
- [6 Interoperabilität Eiffel – Code Contracts](#)
- [7 Performancevergleich](#)

### 1.2.2 Thomas Pani

- [2 Literaturüberblick](#)
- [3 Einführung in Design by Contract](#)
- [5.1 Toolchains](#)
- [5.3 Rückgabewerte](#)
- [5.5 Hilfsprozeduren: Überschreiben](#)
- [5.9 Statisches Binden](#)
- [5.10 Überprüfung von Invarianten](#)
- [5.12 Generics](#)
- [7 Performancevergleich](#)
- [8 Schlussbemerkungen](#)

---

<sup>3</sup>Common Language Runtime

# Kapitel 2

## Literaturüberblick

*Design by Contract* wurde als objektorientiertes Konzept erstmals 1988 von Bertrand Meyer in seinem Buch *Object-Oriented Software Construction* [21] vorgestellt. Meyer beschreibt Prinzipien zur Erstellung qualitativ hochwertiger Software, Methoden um diese zu erreichen, sowie deren Anwendung im objektorientierten Paradigma. Als eine solche Methode wird „programming by contract“ beschrieben, ein formales Übereinkommen zwischen einer Klasse und ihren Aufrufern, durch das für beide Seiten Rechte und Pflichten entstehen.

Meyers Artikel *Applying ‘design by contract’* [19] gibt einen praxisorientierteren Überblick über *Design by Contract*, *Design by contract: the lessons of Ariane* [15] eine populärwissenschaftliche Motivation.

Als Grundlage bezieht sich Meyer auf wesentlich ältere Texte: Floyd beschreibt im Konferenzbeitrag *Assigning meanings to programs* [12] ein Zusicherungskonzept für eine auf Flussdiagrammen aufsetzende Sprache. Darauf aufbauend bindet der Artikel *An Axiomatic Basis for Computer Programming* [13] von C. A. R. Hoare die Zusicherungen in ein Deduktionssystem („Hoare logic“) ein. Zentrales Element hierbei ist das „Hoare-Tripel“ der Form  $P\{Q\}R$ , wobei  $P$  und  $R$  Zusicherungen und  $\{Q\}$  ein Programm beschreiben. Das Tripel sagt aus, dass falls  $P$  vor Ausführung von  $\{Q\}$  wahr ist,  $R$  nach Terminierung von  $\{Q\}$  ebenfalls wahr ist. Das Hoare-Tripel kann damit als formale Grundlage einer mit Zusicherungen bewehrten Prozedur gesehen werden. Beachtet werden muss allerdings, dass in dieser Sichtweise nicht auf Objektorientierung eingegangen wird. Zudem beschreibt Hoare die Zusicherungen in prädikatenlogischen Formeln, während *Design by Contract* üblicherweise mit Mitteln der Programmiersprache auskommen muss.

Der Gedanke eines Vertrags mit den drei Zusicherungsarten (Invariante, Vor- und Nachbedingung) findet sich auch später im „Liskov substitution principle“ (Ersetzbarkeitsprinzip) von Liskov und Wing [17, 18] wieder.

Die Anwendung von *Design by Contract* in Eiffel wird von Meyer umfassend in *Eiffel: The Language* [20, Kapitel 9 – Correctness] beschrieben. Eine Erläuterung von Eiffel for .NET findet sich in *Full Eiffel on the .NET Framework* [23].

*Design by Contract* mit *Code Contracts* in den .NET-Sprachen wird im Artikel *Embedded contract languages* [11] von Fähndrich, Barnett und Logozzo (alle Microsoft Research) beschrieben. Als praktische Referenz dient das *Code Contracts User Manual* [5].

Vor der Auswahl von Eiffel als Referenz wurde zudem Material über andere Sprachen gesichtet. Einen Überblick über DbC in Java gibt [16], Details finden sich in u.a. in [24, 1, 4]. Material zu Python findet sich in [26] und [27].

# Kapitel 3

## Einführung in Design by Contract

*Design by Contract* wurde von Bertrand Meyer als Methodik zur Erhöhung der Zuverlässigkeit objektorientierter Software eingeführt. Zuverlässigkeit beschreibt er dabei als Kombination von Korrektheit und Robustheit der erstellten Software.

### Software als Implementierung von Schnittstellen

Meyer argumentiert, dass die damals vorherrschende Technik der *defensiven Programmierung* durch die eingesetzten umfassenden Überprüfungen zu weiterer Komplexität beiträgt, und dadurch dem Ziel zuverlässiger Software entgegensteht. Stattdessen schlägt er vor, Software als Implementierung „wohlverstandener Spezifikationen“ zu sehen.

Zentral ist dabei der Gedanke, dass diese Spezifikationen nicht allein durch das Typsystem, sondern als komplexe Schnittstellen gegeben sind. Eine solche Schnittstelle, auch *Contract* genannt, beschreibt das Verhalten für die beteiligten Seiten, den Aufrufer (Client) und den aufgerufenen Code (Server). Insbesondere schützt der Contract beide Seiten, indem er definiert:

- welches Resultat der Client zumindest erwarten darf.
- außerhalb welcher Bedingungen der Server scheitern darf.

### Notation der Schnittstelle mit Zusicherungen

Meyer schlägt vor, die Schnittstelle mit *Assertions* (Zusicherungen) zu notieren, und auf deren Einhaltung zu prüfen. Dazu unterscheidet er drei spezielle Arten der Zusicherung:

**Preconditions** (Vorbedingungen) beziehen sich auf einzelne Routinen und geben Bedingungen an, unter denen ein Aufruf korrekt ist.

**Postconditions** (Nachbedingungen) beziehen sich auf einzelne Routinen und geben Bedingungen an, die nach Rückgabe des Kontrollflusses zutreffen.

**(Klassen-)Invarianten** treffen auf alle Instanzen einer Klasse zu, und berühren alle ihre Routinen. Sie geben also generelle Bedingungen an.

### Prüfung der Zusicherungen

Letztendlich soll die Einhaltung der Zusicherungen überprüft werden, da jede Verletzung einer Schnittstelle einen Bug im Programm darstellt. Meyer beschreibt dazu die dynamische Prüfung der Zusicherungen zur Laufzeit: bei jedem Aufruf einer Routine werden die zugehörigen Vor- und Nachbedingungen, sowie anzuwendende Invarianten von der Laufzeitumgebung überprüft. Fehler bei der Prüfung können einen Fehlerfall (Exception) oder andere qualitätssichernde Maßnahmen (z.B. Logging) verursachen.

# Kapitel 4

## Anwendungsbeispiel: Bankkonto

Zum besseren Verständnis der in dieser Arbeit behandelten Thematik sind Codebeispiele angeführt, welche die erklärten Features und Funktionen von Design by Contract (DbC) genauer veranschaulichen sollen. Ein Bankkonto als Anwendungsbeispiel kann die meisten Eigenschaften von DbC in einem realitätsnahen Kontext vermitteln.

### 4.1 Allgemeine Funktionen

Ein Bankkonto ermöglicht zwei grundlegende Funktionen, nämlich das Einzahlen und das Abheben eines bestimmten Geldbetrages auf ein Konto. Das Bankkonto speichert außerdem seinen derzeitigen Saldo und enthält jeweils eine Liste über alle Einzahlungen und Abhebungen.

#### Bewegung

Elemente dieser beiden Listen sind vom Typ `Bewegung`. Eine `Bewegung` enthält Informationen über die Höhe des Geldbetrages der abgehoben oder eingezahlt wurde, sowie den genauen Zeitpunkt zu dem diese Bewegung durchgeführt wurde.

#### 4.1.1 Zusicherungen

Bei der Erstellung unserer Bankkontoapplikation müssen wir selbstverständlich darauf achten, sämtliche Funktionen zuverlässig und korrekt zu implementieren, sodass keine unrechtmäßigen Manipulationen durchgeführt werden können. Aus diesem Grund wollen wir alle Funktionen sowie den globalen Objektzustand mit Zusicherungen davor schützen, dass eine inkorrekte Eingabe oder ein Fehler in unserem Programmcode zum Geldverlust eines Kunden oder einem Programmabsturz führt.

#### Einzahlen

Beim Einzahlen eines Betrags auf das Bankkonto muss sichergestellt werden, dass die übergebene Dezimalzahl positiv ist und nicht versucht wird einen negativen Betrag auf das Konto zu überweisen. Nach der Ausführung von `Einzahlen()` muss der globale Saldo des Kontos um exakt den eingezahlten Betrag erhöht worden sein. Außerdem muss ein neues Bewegungsobjekt mit der aktuellen Zeit und dem eingezahlten Betrag erzeugt und der Liste der Einzahlungen hinzugefügt worden sein.

#### Abheben

Beim Abheben gelten ähnliche Bedingungen. Zuerst muss überprüft werden, ob ein positiver Betrag abgehoben werden soll. Nach Durchführung der Auszahlung muss der Saldo um den abgehobenen Betrag reduziert und ein neues Bewegungsobjekt in die Liste der Abhebungen eingefügt werden. Zusätzlich gibt es bei vielen Bankkonten die Möglichkeit des Überziehens, also das Abheben eines Geldbetrags der größer als der aktuelle Saldo ist. Da nicht bei jedem Konto

ein Überziehungsrahmen konfiguriert sein muss, existieren zwei Varianten der Methode `Abheben`. Die Methode `Abheben(decimal betrag, out bool ueberzogen)` (C# Notation) ermöglicht es einen größeren Betrag abzuheben, als den aktuelle Saldo des Bankkontos. In diesem Fall wird der out-Parameter `bool ueberzogen` gesetzt. Ist ein Überziehen des Kontos nicht gestattet, wird die Methode `Abheben(decimal betrag)` verwendet. Diese ruft für das eigentliche Abheben des Betrags zwar die erste Variante von `Abheben()` auf, jedoch überprüft sie in einer zusätzlichen Assertion, ob der out-Parameter von `Abheben(decimal betrag, out bool ueberzogen)` `false` ist.

### Allgemeiner Objektzustand

Die globalen Variablen des Bankkonto-Objekts müssen sich stets in einem gültigen und korrekten Zustand befinden und dürfen nicht unerlaubt über andere Funktionen manipuliert werden. Deshalb definieren wir eine Invariante, welche die Korrektheit dieser Variablen und Objekte sicherstellt.

Über eine zusätzliche Kontrolle wird die Korrektheit des Saldos überprüft. Dieser muss gleich der Summe aller Einzahlungen abzüglich der Summe aller Abhebungen sein. Desweiteren überprüfen wir in unserer Invariante, ob in den beiden Listen (Einzahlungen und Abhebungen) nur positive Bewegungen enthalten sind, da das Ein - bzw. Auszahlen von negativen Beträgen nicht möglich ist. Außerdem stellt unsere Invariante sicher, dass die beiden Listen aufsteigend sortiert sind, damit die chronologisch letzte Bewegung auch an letzter Stelle der jeweiligen Liste ist.

# Kapitel 5

## Spezialfälle

### 5.1 Toolchains

In diesem Abschnitt werden die Toolchains der untersuchten DbC-Implementierungen *Eiffel for .NET* und *C#+Code Contracts* näher erläutert. In den nachfolgenden Abschnitten werden deren Eigenschaften aufgegriffen und anhand der jeweiligen Spezialfälle diskutiert.

#### 5.1.1 Eiffel for .NET

Für Eiffel wurde die *Eiffel for .NET*-Toolchain von *Eiffel Software* ausgewählt.

Der Compiler erhält dabei als Eingabe Eiffel-Quellcode inklusiver allfälliger – in Eiffel notierter – Zusicherungen. Daraus wird direkt CIL-Code erzeugt, welcher bereits den Code zur Überprüfung der Zusicherungen enthält. Dieser wird anschließend wie für andere .NET-Sprachen üblich assembliert; die erzeugte Assembly ist direkt auf der CLR lauffähig. [23]



Abbildung 5.1: Eiffel-Toolchain. In rot: Module mit vorhandenem Contract-Checking-Code

#### 5.1.2 C# und Code Contracts

Für C# wurde die Kombination C#-Compiler (`csc`) und *Code Contracts*-Tools (`ccrewrite`) von *Microsoft* untersucht. Da CC unabhängig von der verwendeten .NET-Sprache funktionieren soll, existiert auch keine besondere Behandlung dafür im Compiler. Der C#-Quellcode kann also mit einer (was CC betrifft) beliebigen Version des `csc` übersetzt werden. Notiert sind die Zusicherungen mit Marker-Methoden, das sind unäre Methoden mit `void`-Rückgabebetyp und leerem Rumpf, wobei als Parameter die Zusicherung als boolescher Ausdruck übergeben wird. Erst das CC-Tooling analysiert die fertig kompilierte Assembly auf Bytecode-Basis und ersetzt die Marker-Methoden durch Code, der tatsächlich die Einhaltung der Zusicherungen zur Laufzeit prüft. [11, 5]



Abbildung 5.2: C#-Toolchain. In rot: Module mit vorhandenem Contract-Checking-Code

## 5.2 Prestate-Values

Bei der Definition von Nachbedingungen eines *Contracts* ist es sehr oft notwendig, auf den Wert einer Variable zuzugreifen, den diese zum Zeitpunkt des Eintritts in die Methode hatte (*Prestate-Value*). Die Notwendigkeit hierfür beruht darauf, dass Methoden meist Werte von nicht-lokalen Variablen im Laufe ihrer Abarbeitung verändern. Um die Korrektheit dieser Änderungen zu überprüfen, muss auf die Eintrittswerte der Variablen zugegriffen werden können. Im folgenden werden wir das für diesen Aufruf verwendete Sprachkonstrukt allgemein als *Old-Expression* bezeichnen.

### 5.2.1 C#

In C# wird die Marker Methode `T Contract.OldValue<T>(T e)` verwendet, um auf den Wert von `e` zum Eintritt in die Methode zuzugreifen. Die wichtigste Einschränkung für die Verwendung von *Old-Expressions* in C# ist, dass eine Old-Expression sich auf einen Wert beziehen muss, der im Precondition-State der Methode bereits existiert hat. In anderen Worten muss es sich um einen Wert handeln, der immer dann evaluiert werden kann, wenn die Precondition der Methode zutrifft.

Um die Funktionsweise der Marker Methode `Contract.OldValue(e)` besser zu verstehen, betrachten wir unser Bankkonto Beispiel. Ein Bankkonto enthält stets Informationen über den derzeitigen Saldo, sowie je eine Liste aller Einzahlungen und Abhebungen die auf diesem Konto durchgeführt wurden. Die Methode `public Bewegung Einzahlen(decimal betrag)` ermöglicht das Einzahlen eines bestimmten Betrags auf das Bankkonto.

Listing 5.1: Prestate-Values in C#

```
public class Bankkonto
{
    public decimal Saldo { get; protected set; }
    protected readonly IList<Bewegung> _einzahlungen = new List<Bewegung>();
    protected readonly IList<Bewegung> _abhebungen = new List<Bewegung>();
    ...
    public Bewegung Einzahlen(decimal betrag)
    {
        // betrag strictly positive
        Contract.Requires(betrag > 0);
        // saldo correctly updated
        Contract.Ensures(Saldo == Contract.OldValue(Saldo) + betrag);
        // buchung appended to list
        Contract.Ensures(_einzahlungen.Count == Contract.OldValue(_einzahlungen.Count) + 1);
        Contract.Ensures(_einzahlungen.Last().Equals(Contract.Result<Bewegung>()))
        ;

        Saldo += betrag;
        Bewegung result = new Bewegung(DateTime.Now, betrag);
        _einzahlungen.Add(result);
        return result;
    }
    ...
}
```

Als Bedingung für eine korrekte Funktion der Methode `Einzahlen` soll der aktuelle Saldo, dem Saldo vor der Einzahlung plus dem Einzahlungsbetrag entsprechen. Um diese Bedingung zu formulieren wird die Marker Methode `Contract.OldValue(Saldo)` benötigt. Desweiteren soll sichergestellt sein, dass die Einzahlung in Form eines Bewegungsobjekts der Liste `List<Bewegung> _einzahlungen` hinzugefügt wird. Außerdem wird überprüft, ob die Anzahl der Elemente der Liste um eins erhöht wurde, wofür der Zugriff auf den Prestate-Value von `_einzahlungen.Count` notwendig ist.

### Technische Umsetzung

Um die technische Umsetzung von Prestate-Values besser nachvollziehen zu können, betrachten wir den von `ccrewrite` erzeugten Bytecode (nach C# decompiliert) unseres obigen Bankkonto-Beispiels etwas genauer. Statt der Aufrufe der Marker-Methoden `Contract.OldValue(e)` erzeugt

`crewrite` für jede Old-Expression eine lokale Variable, in die der *Prestate-Value* der jeweiligen globalen Variable zwischengespeichert wird.

Listing 5.2: Technische Realisierung durch `crewrite`

```
public Bewegung Einzahlen(decimal betrag)
{
    decimal Contract.Old(Saldo);
    decimal Contract.Old(betrag);
    int Contract.Old(Count);
    ...
    try
    {
        Contract.Old(Saldo) = this.Saldo;
    }
    ...
}
```

### 5.2.2 Eiffel

In Eiffel steht das Keyword `old` zur Verfügung, das einer Variable vorangestellt wird, wenn auf ihren Prestate-Value zugegriffen werden soll. Zur besseren Veranschaulichung zeigt Listing 5.3 einen Ausschnitt der Implementierung des Bankkonto Beispiels in Eiffel.

Listing 5.3: Prestate-Values in Eiffel

```
class
    BANKKONTO
    ...
    feature — Access
        saldo : REAL
        einzahlungen : LINKED_LIST [BEWEGUNG]
        abhebungen : LINKED_LIST [BEWEGUNG]
    feature
        einzahlen (betrag : REAL) : BEWEGUNG
        require
            non_negative: betrag > 0
        do
            saldo := saldo + betrag

            Result := bewegung

            einzahlungen.put(bewegung)
        ensure
            saldo_updated: saldo = old saldo + betrag
            einzahlungen_one_more: einzahlungen.count = old einzahlungen.count + 1
            einzahlungen_last_equals_new: einzahlungen.last.is_equal (Result)
        end
    ...
end
```

#### Technische Umsetzung

Der Eiffel Compiler erzeugt ebenfalls temporäre lokale Variablen, um den Zugriff auf Prestate Values zu ermöglichen.

### 5.2.3 Zusammenfassung

Nach genauer Betrachtung der technischen Realisierung von Prestate-Values in den beiden Programmiersprachen konnten keine wesentlichen Unterschiede festgestellt werden. Dies ist darauf zurückzuführen, dass die naheliegendste Implementierung für den Zugriff auf Prestate-Values in Postconditions eine Zwischenspeicherung der Variablenwerte bei Methodeneintritt ist.

## 5.3 Rückgabewerte

Rückgabewerte sind die primären Ergebnisse eines Unterprogrammaufrufs (außer der Objektstatus wird zusätzlich geändert), daher müssen diese in Nachbedingungen zugreifbar sein.

Im nachfolgenden Abschnitt werden die jeweils zur Verfügung stehenden Konstrukte für den Zugriff auf Rückgabewerte diskutiert und anschließend in Bezug auf ihre Nutzung in Nachbedingungen betrachtet.

### 5.3.1 C#

In C# werden Rückgabewerte per **return**-Statement auf den Stack gepusht und anschließend die Kontrolle an den Aufrufer zurückgegeben. Zusätzlich stehen **ref**- und **out**-Parameter zur Verfügung, deren Werte beim Return auf die beim Methodenaufruf angegebenen Felder zurückgeschrieben werden. **out**-Parameter stellen eine für Rückgabewerte übliche Verallgemeinerung der **ref**-Parameter dar. Deshalb wird auf letztere nicht gesondert eingegangen. [3, 10.5.1.2 Reference parameters und 10.5.1.3 Output parameters]

**out**-Parameter müssen bei der Übergabe nicht zugewiesen sein, dadurch ist ein direkter Zugriff über den Variablennamen nicht möglich: Zwar könnte der Parameter von **ccrewrite** im Bytecode erkannt werden, ein Zugriff auf nichtzugewiesene Variablen wird aber vom C#-Compiler unterbunden. Da Contracts gesammelt als Block am Beginn der Methode notiert werden müssen, wäre das immer der Fall. Allerdings darf der Parameter einer weiteren Methode als **out**-Parameter übergeben werden. Genau das tut die Marker-Methode `T Contract.ValueAtReturn<T>(out T value)`.

Der reguläre **return**-Wert verfügt über keinen Namen, weshalb auf ihn mit der Marker-Methode `T Contract.Result<T>()` verwiesen wird.

Die folgende Methode zur Ganzzahl-Division illustriert den Zugriff auf Rückgabewerte in C#:

Listing 5.4: Rückgabewerte in C#

```
int IntDivide(int a, int b, out int remainder) {
    Contract.Ensures(Math.Sign(a) == Math.Sign(Contract.ValueAtReturn(out
        remainder)));
    Contract.Ensures(Math.Abs(Contract.Result<int>()) <= Math.Abs(a));

    remainder = a % b;
    return a / b;
}
```

### Technische Umsetzung

Gelöst wird der Zugriff auf den **return**-Wert über Zwischenspeichern in einer von **ccrewrite** zusätzlich eingeführten, lokalen Variable. Auf die **out**-Parameter kann direkt zugegriffen werden, da die Prüfung der Nachbedingungen am Ende der Methode stattfindet (im Gegensatz zu ihrer Notation am Beginn des Methoden-Rumpfes).

### 5.3.2 Eiffel

In Eiffel steht eine implizit vorhandene Variable **Result** vom deklarierten Rückgabebetyp zur Verfügung, deren Wert beim Austritt aus der Funktion den Rückgabewert repräsentiert. Folglich kann zum Zugriff auf den Rückgabewert direkt diese Variable angesprochen werden:

Listing 5.5: Rückgabewerte in Eiffel

```
int_divide(a, b: INTEGER): INTEGER
do
    Result := (a / b).floor
ensure
    Result.abs <= a.abs
end
```

## Technische Umsetzung

Gelöst ist der Zugriff auch hier über eine lokale Variable. Diese muss aber nicht zusätzlich angelegt werden; der Eiffel-Compiler erzeugt unabhängig von DbC eine lokale Variable `Result`, die schließlich mit einem CIL-üblichen `Return` zurückgegeben wird.

## Simulation von `out`-Parametern in Eiffel

Eiffel besitzt kein den C#-`out`-Parametern ähnliches Konstrukt. Soll eine Funktion trotzdem ad-hoc (ohne zusätzliche Klassen-Deklaration) mehrelementige Ergebnisse liefern, kann der generische Typ `TUPLE [T1 ... Tn]` verwendet werden. (`TUPLE` ist, da die Anzahl der Typparameter variabel ist, keine Klasse, sondern als Keyword implementiert.)

Die oben in C# vorgestellte Methode mit `out`-Parametern könnte also derart simuliert werden:

Listing 5.6: `TUPLE` als Workaround für `out`-Parameter in Eiffel

```
int_divide(a, b: INTEGER): TUPLE [INTEGER, INTEGER]
do
  Result := [(a / b).floor, a // b]
end
```

Es sollte auf eine Benennung jener `TUPLE`-Attribute, auf die in Zusicherungen zugegriffen werden soll, geachtet werden (`TUPLE [name1: T1; name2: T2]` im Gegensatz zu `TUPLE [T1, T2]`). Nur so ist ein typsicheres Auslesen der Elemente (`t.name1: T1`) möglich; das Ergebnis der Funktion `item (i: INTEGER): ANY` müsste per „Assignment attempt“ (Operator `?=`) in einer Hilfsprozedur umgewandelt werden ( $n$  Hilfsprozeduren bei  $n$  unterschiedlichen Tupel-Attribut-Typen). Dadurch können weitere Probleme auftreten, auf die wir im Weiteren eingehen.

## 5.4 Hilfsprozeduren

Um in Zusicherungen komplexe Bedingungen zu überprüfen, können in beiden Sprachen Hilfsprozeduren aufgerufen werden. (Wir verwenden den Namen *Hilfsprozedur* in diesem Dokument für jede Routine, die in einer Zusicherung zu Hilfszwecken aufgerufen wird, unabhängig von der Nomenklatur der jeweiligen Programmiersprache.) In den folgenden Abschnitten wird erläutert, welche Probleme dabei auftreten können.

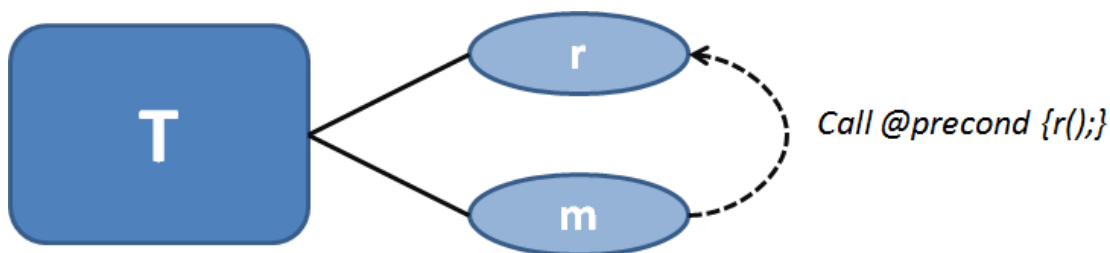


Abbildung 5.3: Typ `T` mit Methode `m` und Hilfsprozedur `r`. `r` wird in der Vorbedingung von `m` aufgerufen.

## 5.5 Hilfsprozeduren: Überschreiben

Sei `U` ein Untertyp eines mit Zusicherungen versehenen Typs `T`, und `T.r` eine in einer Zusicherung aufgerufene Hilfsprozedur. Dann kann in `U` diese Zusicherung effektiv umgangen werden, indem `r` mit einer geeigneten Implementierung `r+` überschrieben wird. Es sollten also Mechanismen zur Verfügung stehen, die ein solches Aushebeln des Vertrags verhindern, oder zumindest vor unbeabsichtigtem Überschreiben warnen.

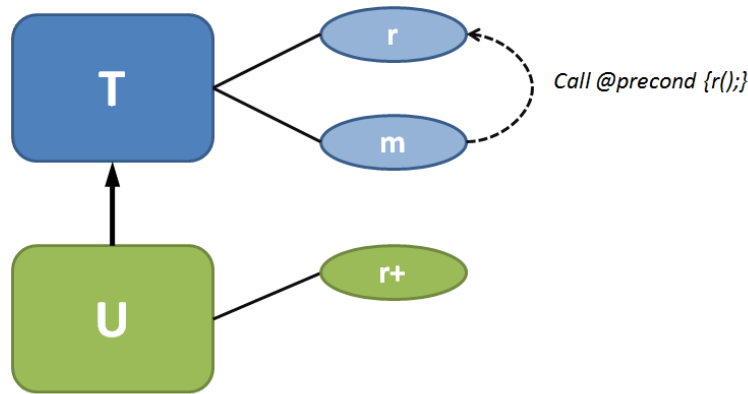


Abbildung 5.4: Untertyp  $U$  überschreibt  $T.r$  mit  $U.r^+$ .

### 5.5.1 C#

#### Sichtbarkeit

Eine Möglichkeit Überschreiben zu verhindern, ist die Hilfsmethode im Obertyp als **private** zu deklarieren.

Diese Variante ist ausschließlich für Invarianten geeignet! In Vor- und Nachbedingungen werden Hilfsmethoden immer direkt aufgerufen. Die notwendigen Aufrufe werden auf Bytecode/CIL-Ebene einfach in den Untertyp kopiert. Da in CIL alle Aufrufe absolut (als `Package.Klasse::Member()`) qualifiziert werden (nicht etwa wie in C# als `base.Member()`), ist das ohne Probleme möglich. Fehlen aber im Untertyp entsprechende Zugriffsrechte, weil die Methode als **private** gekennzeichnet ist, so schlägt der Aufruf der Hilfsmethode aus dem Untertyp zur Laufzeit mit einer `MethodAccessException` fehl.

Für Invarianten ist dies kein Problem: alle Invarianten werden in einer speziellen Methode `void $InvariantMethod$()` im Obertyp zusammengefasst und diese als **protected** für Untertypen sichtbar gemacht, wodurch sie ohne Probleme aufgerufen werden kann.

Außerdem ist zu beachten, dass durch die Einschränkung der Sichtbarkeit möglicherweise die Wiederverwendbarkeit des Codes reduziert wird.

#### Statisches Binden

Zur Erläuterung dieses Lösungsansatzes betrachten wir zuerst die beiden Möglichkeiten, in C# Aufrufe zu binden:

**„Echtes“ Überschreiben** Um in C# eine Methode überschreiben zu dürfen, muss die zugrundeliegende Methode im Obertyp als virtuell deklariert sein. Die überschreibende Methode spezialisiert die als virtuell eingeführte Methode, indem sie sie neu implementiert. Syntaktisch impliziert eine als **override** gekennzeichnete Methode in der Kindklasse eine Markierung als **virtual** in der Basisklasse, andernfalls liefert der Compiler einen Fehler. **virtual** bewirkt, dass alle Aufrufe dieser Methode dynamisch gebunden werden. [3, 10.5.3 Virtual methods und 10.5.4 Override methods]

Listing 5.7: Überschreiben in C#

```

public class T {
    public virtual void DoSomething() { /* Funktion */ }
}
public class U : T {
    public override void DoSomething() { /* spezialisierte Funktion */ }
}
  
```

**Method Hiding** Zusätzlich ist – unabhängig vom Konstrukt des Überschreibens, aber leicht damit zu verwechseln – das Verstecken eines Members möglich. Konkret wird dabei ein Member mit gleichem Namen im Untertyp deklariert. [3, 3.7.1.2 Hiding through inheritance] Die auftretende Compiler-Warnung kann durch den `new`-Modifier unterdrückt werden:

Listing 5.8: Method-Hiding in C#

```
public class T {
    public void DoSomething() { /* Funktion */ }
}
public class U : T {
    new public void DoSomething() { /* andere Funktion */ }
}
```

Beachte: `U::DoSomething()` stellt eine vollkommen andere Methode als `T::DoSomething()` dar; beide Member teilen sich nur „zufällig“ den selben Namen. Daher erfolgt das Binden nicht dynamisch; die korrekte Methode wird als Kombination aus statischem Typ und Methodenname ermittelt.

**Statisches Binden als Lösungsansatz** Überschreiben (mit `override`) ist in C# nur bei gleichzeitigem dynamischen Binden möglich. Eine weitere Möglichkeit das Überschreiben zu verhindern, besteht also darin, die Hilfsmethode im Obertyp nicht als `virtual` zu definieren. Dann ist nur mehr ein Verstecken der Methode möglich (mit `new`). Die korrekte Methode im Obertyp wird aufgerufen, da lediglich eine neue Methode mit gleichem Namen im Untertyp entsteht.

Bemerkenswert ist, dass auf dieses Problem und zugehörige Lösungsansätze in der Literatur und der Dokumentation zu *Code Contracts* überhaupt nicht eingegangen wird.

## 5.5.2 Eiffel

**Sichtbarkeit** Die Sichtbarkeit von Features kann in Eiffel durch eine Clients-Liste in der `feature`-Klausel eingeschränkt werden (`feature {CLIENT_1, CLIENT_2, ..., CLIENT_N}`). Unqualifizierte Aufrufe innerhalb eines Typs sind aber immer erlaubt; die Sichtbarkeit wirkt sich nur auf die Client-Server-Beziehung, nicht aber auf die Vererbungsbeziehung aus. Ein Verhindern des Überschreibens wie mit einem C#-`private` ist folglich nicht möglich.

**Statisches Binden** Aufrufe werden in Eiffel immer dynamisch gebunden, d.h. es existieren keine Mechanismen wie in C#, die das Überschreiben von statisch gebundenen Methoden verhindern.

Andere, Eiffel-spezifische Möglichkeiten das Überschreiben zu verhindern, konnten wir nicht finden. Auch hier muss angemerkt werden, dass in der Literatur und der Dokumentation zu Eiffel nicht auf dieses Problem eingegangen wird.

## 5.6 Hilfsprozeduren: Seiteneffekte

Bei der Verwendung von Hilfsprozeduren in Zusicherungen dürfen diese Funktionen den Prestate der aufrufenden Methode nicht verändern. Andernfalls können bereits evaluierte Zusicherungen des Aufrufers nachträglich verletzt werden.

### 5.6.1 C#

In C# müssen Funktionen, die als Hilfsprozeduren für Zusicherungen dienen, mit dem Attribut `[Pure]` gekennzeichnet werden. Das `PureAttribute` bedeutet, dass die damit gekennzeichnete Methode keine Änderungen am bereits vorhandenen Objektzustand vornehmen darf. Es dürfen lediglich Objekte verändert werden, die nach Eintritt in diese Methode erzeugt werden.

```

[ContractInvariantMethod]
private void ObjectInvariant()
{
    Contract.Invariant(CheckSum()); //call Helperfunction CheckSum()
    ...
}

//Helperfunction to check if Saldo is correct
[Pure]
private bool CheckSum()
{
    return Saldo == _einzahlungen.Sum(b => b.Betrag) - _abhebungen.Sum(b => b.
        Betrag);
}

```

In unserem Bankkonto-Beispiel wird die Hilfsfunktion `CheckSum()` in der Invariante verwendet, um die Korrektheit und Konsistenz des Saldos und der Bewegungssummen zu gewährleisten. Die Hilfsfunktion besitzt das Attribut `[Pure]` und „verspricht“ damit, keine Änderungen am *Prestate* des Objekts vorzunehmen.

### SideEffect-Verification

In der derzeitigen Version des .NET-Frameworks erkennt der Compiler, wenn eine Hilfsprozedur innerhalb einer Zusicherung aufgerufen wird und signalisiert mit einer Warnung, dass diese Hilfsprozedur mit dem `PureAttribute` gekennzeichnet werden muss. Dem Entwickler sollte jedoch bewusst sein: Weder zur Compilezeit, noch zur Laufzeit wird überprüft, ob eine solche Prozedur auch wirklich keine Seiteneffekte aufweist.

#### 5.6.2 Eiffel

In Eiffel gibt es kein dem `PureAttribute` entsprechenden Sprachkonstrukt. Als guter Eiffel-Stil hat sich jedoch das *Command-Query Separation Principle (C-Q-SP)* durchgesetzt: [21]

The features that characterize a class are divided into commands and queries. A command serves to modify objects, a query to return information about objects. A command is implemented as a procedure. A query may be implemented either as an attribute [...] or as a function [...].

Command-Query Separation principle: Functions should not produce abstract side effects. An abstract side effect is a concrete side effect<sup>1</sup> that can change the value of a non-secret query.

Wird durch den Entwickler das C-Q-SP berücksichtigt und werden dadurch nur Queries als Hilfsprozeduren aufgerufen, ist sichergestellt, dass diese keine Seiteneffekte aufweisen.

## 5.7 Hilfsprozeduren: Quantoren

In vielen Vor- bzw. Nachbedingungen werden Zusicherungen auf Datenstrukturen mit mehreren Elementen, wie zum Beispiel Listen, Arrays, usw. definiert. Dabei soll eine bestimmte Bedingung auf alle Elemente der Datenstruktur zutreffen, damit die Zusicherung gültig ist. Um dieses Problem in einer Zusicherung formulieren zu können, bietet die Programmiersprache diese Funktionen entweder bereits implizit in Form von Quantoren an, oder eine selbstdefinierte Hilfsprozedur, die eine Iteration über alle Elemente der Datenstruktur durchführt, ist notwendig.

ist entweder eine selbstdefinierte Hilfsprozedur notwendig, die eine Iteration über alle Elemente der Datenstruktur durchführt, oder die Programmiersprache bietet diese Prozeduren bereits implizit in Form von Quantoren an.

<sup>1</sup>concrete side effect: assignment, assignment attempt or creation instruction targeting an attribute; a procedure call.

### 5.7.1 C#

Die `Contract`-Klasse in `C#` stellt zwei Funktionen zur Verfügung, die Quantifizierung auf Elemente einer Datenstruktur ermöglichen. Die Funktion

```
Contract.ForAll<T>(IEnumerable<T> collection, Predicate<T> predicate)
```

hat die Bedeutung des aus der Prädikatenlogik bekannten All-Quantors, während die Funktion

```
Contract.Exists<T>(IEnumerable<T> collection, Predicate<T> predicate)
```

den Existenz-Quantor repräsentiert.

Listing 5.10: Quantifizierung

```
[ContractInvariantMethod]
private void ObjectInvariant()
{
    Contract.Invariant(Contract.ForAll(_einzahlungen, e => e.Betrag > 0));
}
```

Listing 5.10 zeigt den Einsatz des All-Quantors mit Hilfe der `ForAll`-Funktion. Der `Contract` stellt sicher, dass für alle Einzahlungen die auf das Bankkonto getätigt wurden, der überwiesene Betrag größer als Null ist.

### LINQ - Language Integrated Queries

Als zweite Variante für die Verwendung von Quantoren in `C#` steht `LINQ` zur Verfügung. `LINQ` ist eine Erweiterung des `.NET` Framework um sprachintegrierte Abfrage-, Festlegungs- und Transformationsoperationen. Es erweitert die beiden `.NET`-Sprachen `C#` und `VisualBasic` um eine eigene Sprachsyntax zur Formulierung von *Queries*. [7] `LINQ` stellt neben den beiden Quantifizierungsfunktionen

```
All<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)
```

und

```
Any<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)
```

auch viele Aggregatsfunktionen zur Verfügung, wie man sie aus anderen Abfragesprachen, wie zum Beispiel `SQL`, kennt.

### 5.7.2 Eiffel

In `Eiffel` ist der Einsatz von Quantoren zur Iteration von Elementen einer Datenstruktur ebenfalls möglich. Dafür sind in der Klasse `Traversable` in `EiffelBase` eine Reihe von Iterationsfunktionen definiert, unter anderem die Funktionen `for_all`, `do_all`, `do_if`. Alle Klassen die von `Traversable` abgeleitet sind, implementieren diese Iterationsfunktionen, so zum Beispiel auch die Klasse `Collection` und alle ihre Unterklassen, wodurch die Iteration über alle Elemente einer Liste sehr einfach möglich ist.

### Agents

Die Iterationsfunktionen nehmen als Argument sogenannte *Agents* an. Der Agent-Mechanismus in `Eiffel` ermöglicht es, Operationen in Form von Agent-Objekten darzustellen, und diese Objekte als Argumente an beliebige andere Softwareelemente zu übergeben, sodass diese die Operationen zu einem beliebigen Zeitpunkt ausführen können.

Listing 5.11: Agents

```
intlist: LINKED_LIST [INTEGER]
integer_property (i: INTEGER): BOOLEAN
intlist.for_all( agent integer_property(?) )
```

Listing 5.11 zeigt die essentiellen Codezeilen zur Verwendung von `Agents`. Es handelt sich hierbei um einen `Call-Agent`. `Call-Agents` sind Operationen die in Form von *Features* in einer Klasse definiert werden. Sie werden bei der Definition eines `Agents` wie ein Funktionsaufruf nach dem Schlüsselwort `agent` notiert. Das Fragezeichen symbolisiert dabei, dass hier ein beliebiger Parameter der Funktion `integer_property` übergeben wird.

Call-Agents scheinen nicht die optimale Strategie für die Umsetzung von Quantifizierung zu sein, da es sich bei ihnen ebenfalls nur um Hilfsprozeduren handelt, die nicht vor Seiteneffekten geschützt sind. Es ist aber auch möglich, die Operationen gleich bei der Erzeugung eines Agents zu definieren. Bei diesen sogenannten Inline-Agents, wird die gesamte Funktion des Agents direkt an Stelle seines Aufrufes gesetzt und daher nicht als *Feature* der Klasse deklariert. Der Einsatz von sogenannten Inline-Agents wird in Listing 5.12 veranschaulicht.

Listing 5.12: Inline-Agents

```

invariant
  einzahlungen_positiv :
    einzahlungen.for_all (agent (x: BEWEGUNG): BOOLEAN do Result := x.betrag >
      0 end)
  ...
end

```

Inline-Agents in Eiffel sind somit den in C# vorgestellten LINQ-Queries sehr ähnlich, da die von ihnen bereitgestellte Funktionalität nicht in einer eigenen Funktion ausgelagert werden muss.

## 5.8 Hilfsprozeduren: Zusammenfassung und Vergleich

In C# steht eine effektive Methode zur Verfügung, um das Problem des Überschreibens in Hinblick auf Hilfsprozeduren zu unterbinden. Anzumerken ist allerdings, dass diese nicht automatisch eingesetzt wird, sondern ausschließlich in die Verantwortung und Sorgfalt des Programmierers fällt. In Eiffel konnte keine Möglichkeit gefunden werden, das Problem einzuschränken.

Seiteneffekte stellen ein großes Problem beim Einsatz von Hilfsprozeduren zusammen mit Zusicherungen dar. In beiden Programmiersprachen liegt es auch hier allein in der Verantwortung des Programmierers, Seiteneffekte zu vermeiden. C# ermöglicht mit der Einführung des [Pure] Attributs eine erste zusätzliche Kontrolle durch den Compiler, wobei dieser noch keine wirkliche Sideeffect-Verification durchführen kann.

Quantoren sind ein nützliches Hilfsmittel, um Zusicherungen für eine mehrelementige Datenstruktur zu definieren, ohne explizite Hilfsprozeduren implementieren zu müssen. In beiden Sprachen stehen dafür geeignete Methoden zur Verfügung. Durch den Einsatz dieser vordefinierten Quantifizierungsmethoden wird das Problem des Überschreibens verhindert, jedoch nicht das Auftreten von Seiteneffekten im Allgemeinen.

## 5.9 Statisches Binden

Zur Beschreibung des Spezialfalls „Statisches Binden“ soll folgende Typhierarchie betrachtet werden: Sei  $U$  ein Untertyp eines mit einer Invariante versehenen Typs  $T$ . Sei weiters  $T.r$  eine Prozedur in  $T$  und als  $U.r^+$  in  $U$  überschrieben.

Invarianten werden als Zuständigkeit des aufgerufenen Objekts betrachtet: Wenn die Invariante zum Eintritt in eine Prozedur erfüllt ist, muss sie auch zum Austritt erfüllt sein; diese Zusicherung kann sinnvollerweise nur der aufgerufene Code geben. In der Folge ergibt sich, dass Invarianten kovariant verändert werden dürfen (also in Richtung der Untertypbeziehung stärker werden dürfen). Es gilt:  $INV_U \geq INV_T$ . In der oben genannten Typhierarchie muss demnach  $T.r$  die Invariante  $INV_T$  erfüllen, die überschriebene Prozedur  $U.r^+$  die im Allgemeinen stärkere Invariante  $INV_U$ . (Streng genommen könnte man auch die gemeinsame Zuständigkeit von Client und Server für die Erfüllung der Invariante fordern. In den untersuchten Implementierungen ist aber die zuvor beschriebene kovariante Veränderung erlaubt.)

Werden nun Aufrufe statisch gebunden, ergibt sich folgendes Problem: Wir betrachten eine Instanz  $u$  mit dem dynamischen Typ  $U$  und dem statischen Typ  $T$ . Wird nun  $u.r$  aufgerufen, und dieser Aufruf statisch gebunden, ist  $T.r$  die aufgerufene Prozedur.  $T.r$  sichert aber nur  $INV_T$  zu, die übrigen Zusicherungen  $INV_T \setminus INV_U$  nicht. Da  $u$  aber an anderer Stelle im Programm mit einem spezielleren statischen Typ angesprochen werden kann, erfüllt  $u$  nach einem statisch

gebundenen Aufruf im Allgemeinen nicht mehr alle seine Zusicherungen  $INV_U$ , obwohl der Aufrufer davon ausgehen dürfte.

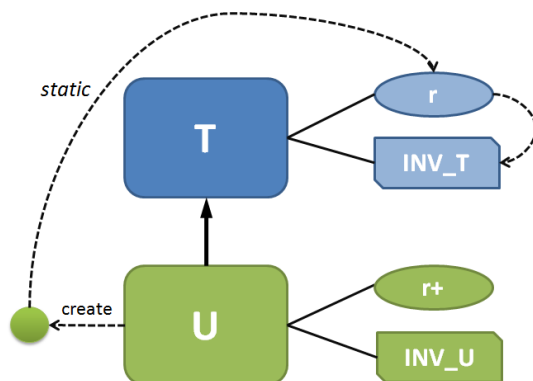


Abbildung 5.5: Statisch über  $T$  gebundener Aufruf von  $U.r$  führt in die Invariante  $INV_T$ .

### 5.9.1 C#

Wie bereits in 5.5 **Hilfsprozeduren: Überschreiben** beschrieben, können Methoden in C# in einem Untertyp entweder überschrieben oder versteckt werden. Diese Möglichkeiten implizieren entsprechend dynamisches oder statisches Binden von Aufrufen der betroffenen Methode.

#### Überschreiben

Da bei mit `virtual/override` überschriebenen Methoden Aufrufe ausschließlich dynamisch gebunden werden, kann der oben beschriebene Spezialfall in C# so nicht herbeigeführt werden.

#### Method Hiding

Mittels Method Hiding und dem hierbei angewendeten statischen Binden, ergibt sich eine Möglichkeit, die Invariante des Untertyps auszuhebeln:

Angenommen  $T$  verlangt in seiner Invariante, dass das Feld  $i$  nichtnegativ ist.  $U$  verstärkt diese Invariante, indem verlangt wird, dass  $i$  positiv ist. Ein Aufruf von `SetToValidValue()` setzt  $i$  auf einen in diesem Typ gültigen Wert; `T::SetToValidValue()` setzt ihn aber im Speziellen auf den einzigen in  $T$  gültigen und in  $U$  ungültigen Wert 0.

Listing 5.13: Statisches Binden mit Method-Hiding in C#

```
class Program {
    static void Main(string[] args) {
        T uAsT = new U();
        uAsT.SetToValidValue();
        uAsT.DynamicCallToForceInvariantCheck(); // Invariant check of U fails.
    }
}

public class T {
    protected int i = 100;

    [ContractInvariantMethod]
    private void Invariant() {
        Contract.Invariant(i >= 0); // i non-negative
    }

    public void SetToValidValue() {
        i = 0;
    }

    public virtual void DynamicCallToForceInvariantCheck() { }
}
```

```

}

public class U : T {
    [ContractInvariantMethod]
    private void Invariant() {
        Contract.Invariant(i > 0); // i positive
    }

    new public void SetToValidValue() {
        i = 50;
    }

    public override void DynamicCallToForceInvariantCheck() { }
}

```

Da `uAsT` vom statischen Typ `T` ist, wird die (in `U` durch einen anderen Member versteckte) Methode `T::SetToValidValue()` aufgerufen. Diese setzt `i` auf 0, erfüllt also auch ihre Zusicherung  $INV_T$ . Nach Abschluss des Aufrufs erfüllt das Objekt aber seine eigene Invariante  $i > 0$  nicht mehr. Zu diesem Zeitpunkt wird jedoch kein Fehler durch die Contract-Überprüfung erkannt, sondern erst beim dynamisch gebundenen Aufruf von `DynamicCallToForceInvariantCheck()`.

### 5.9.2 Eiffel

In Eiffel ist ausschließlich dynamisches Binden möglich. Meyer begründet diese Design-Entscheidung explizit damit, dass ein Aufruf immer die Klassen-Invariante erfüllen muss („invariant preservation requirement“). [20, 21.10 The importance of being dynamic]

### 5.9.3 Zusammenfassung und Vergleich

Nach Meyers korrekter Argumentation muss in Eiffel dynamisch gebunden werden, damit die Invariante korrekt geprüft wird. Die Ursache ist aber die Umsetzung in Eiffel: Die Invarianten liegen in einer `invariant`-Klausel vor, die direkt mit der Klasse für die sie definiert ist, korrespondiert. Bei einem Aufruf muss über dynamisches Binden in die korrekte Klasse gesprungen werden.

Liegt die Invariante jedoch wie in `C#` ohnehin als Methode vor, verschiebt sich diese Zuständigkeit: Statisches Binden per se stellt kein Problem dar, solange immer die speziellste Invarianten-Methode aufgerufen wird. Warum das im vorangegangenen Beispiel trotzdem nicht der Fall war, zeigt das folgende Kapitel.

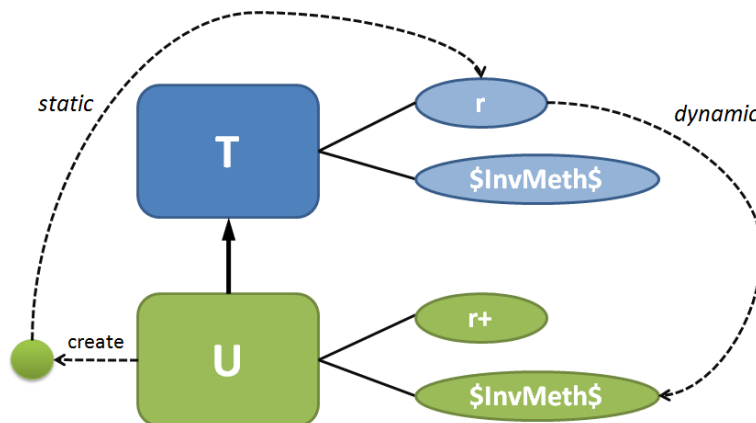


Abbildung 5.6: Statisch über `T` gebundener Aufruf von `U.r` mit dynamisch gebundenen Invarianten-Methoden führt korrekt nach  $INV_U$ .

Selbst im dargestellten Szenario mit Invarianten-Methoden besteht aber folgendes Problem: `T.r` kann nicht wissen, welche Bedingungen in der Invariante  $INV_U$  eines beliebigen Untertyps gefordert werden, sichert diese im Allgemeinen also auch nicht zu. Der Entwickler muss deshalb

selbst entscheiden, ob ein statisch gebundener Aufruf von  $T.r$  tatsächlich die Invariante des dynamischen Typs erfüllen kann. Insbesondere ist dies zu beachten, wenn eine Invariante im Untertyp nachträglich verändert wird; alle statisch aufrufbaren Methoden der Obertypen müssen dann erneut auf Einhaltung der speziellsten Invariante überprüft werden.

Insgesamt scheint eine Beschränkung auf dynamisches Binden im Zusammenhang mit Design by Contract sinnvoll.

## 5.10 Überprüfung von Invarianten

### 5.10.1 C#

Aufgrund des im vorherigen Abschnitt thematisierten statischen Aufrufs der Invarianten-Methode in CIL ergibt sich ein weiterer Problemfall, der am nachfolenden Beispiel illustriert werden soll:

Listing 5.14: Prüfung von Invarianten in C#

```

1  class Program {
2      static void Main(string[] args) {
3          U u = new U();
4          u.SetToValidValue();
5          // Invariante von U ist hier nicht erfuehlt,
6          // dies wird aber zu diesem Zeitpunkt nicht erkannt.
7      }
8  }
9
10 public class T {
11     protected int i = 100;
12
13     [ContractInvariantMethod]
14     private void Invariant() {
15         Contract.Invariant(i >= 0); // i non-negative
16     }
17
18     public void SetToValidValue() {
19         i = 0;
20     }
21 }
22
23 public class U : T
24 {
25     [ContractInvariantMethod]
26     private void Invariant() {
27         Contract.Invariant(i > 0); // i positive
28     }
29 }

```

In Zeile 4 wird `u.SetToValidValue()` aufgerufen. Diese Methode wird in `U` nicht überschrieben, daher lautet der vollständig qualifizierte, statische Aufruf in CIL `T::SetToValidValue()`.

Nun muss die Invariante überprüft, also `$InvariantMethod$()` aufgerufen werden. Der Aufruf im CIL-Code lautet `call instance void T::$InvariantMethod$()`. Damit wird lediglich statisch gebunden – also nur  $INV_T$  überprüft – obwohl `$InvariantMethod$()` als virtual definiert ist und mit der CIL-Instruktion `callvirt` dynamisch gebunden werden könnte.

Der tiefste Call-Stack zum Aufruf von `u.SetToValidValue()` lautet demnach:

1. `T.$InvariantMethod$()`
2. `T.SetToValidValue()`
3. `Program.Main(string[] args)`

Der vorgestellte Code stellt demzufolge eine Verallgemeinerung des Problems der unzureichenden Invarianten-Prüfung im vorherigen Kapitel dar: `U::$InvariantMethod$()` wird nie aufgerufen, die Verletzung der Invariante  $INV_U$  wird daher nicht erkannt, bis eine Methode aufgerufen wird, die entweder mit `U::` qualifiziert werden muss (also eine nur in `U` vorhandene) oder mittels

Überschreiben und dynamischem Binden in den Code von `U` führt. Im vorherigen Kapitel wurde dies durch Aufruf der „Pseudo“-Methode `DynamicCallToForceInvariantCheck()` erreicht.

Im schlimmsten Fall terminiert das Programm, ohne dass die Verletzung der Invariante erkannt und angezeigt wird. Selbst wenn die Verletzung schließlich durch einen Aufruf in `U` erkannt wird, ist die Rückverfolgbarkeit zur Operation, die die Invarianten-Verletzung hervorgerufen hat, beschränkt, da zwischenzeitlich mehrere Aufrufe in den Obertyp stattgefunden haben können.

Warum hier auf dynamisches Binden mit `callvirt` verzichtet wurde, lässt sich nicht nachvollziehen. Zu Testzwecken haben wir das obige Codebeispiel übersetzt, und anschließend der Bytecode mit `ildasm` disassembliert. Im erhaltenen CIL-Code haben wir die `call`-Operation durch `callvirt` ersetzt, und diesen mit `ilasm` wieder assembliert. In der so erzeugten Assembly wird die Verletzung der Invariante korrekt erkannt und eine `ContractException` geworfen.

## 5.10.2 Eiffel

Im von *Eiffel for .NET* generierten Code wird diesem Problem mit dynamischem Binden begegnet:

Die Wurzel der Eiffel-Typhierarchie, `ANY`, wird in CIL als Interface `Any` abgebildet. Dieses implementiert ein Interface `EIFFEL_TYPE_INFO`, welches eine Methode `_invariant()` deklariert. Diese Methode wird vom Eiffel-Compiler gegebenenfalls in den konkreten Klassen überschrieben, und mit den jeweiligen Invarianten befüllt.

Zur Überprüfung einer Invariante wird vor und hinter jedem Aufruf einer Methode `x.m` vom Compiler `ISE_RUNTIME.check_invariant(object o, bool entry)` eingefügt, wobei für `o` das betroffene Objekt `x` übergeben wird. Darin wird dynamisch über `o` gebunden die Methode `o._invariant()` und von dieser weiter die statische – vom Compiler aus den Zusicherungen generierte – Methode `void $$invariant([In] U Current)` aufgerufen. Hier wird schließlich für `Current` die Invariante `INVU` geprüft, und anschließend für alle Vaterklassen `T` mit `T.$$invariant(Current)` die Invariante `INVT` des Obertyps weitergeprüft.

## 5.10.3 Zusammenfassung und Vergleich

Das statische Binden des Invarianten-Aufrufs in `C#` stellt ein großes Problem dar, da der Programmierer offensichtlich ein anderes Verhalten erwarten dürfte. Setzt er sich nicht äußerst intensiv mit *Code Contracts* auseinander, könnten Fehler, die er durch `DbC` zu erkennen glaubt, doch im Code erhalten bleiben.

Wir haben das Team bei *Microsoft Research* mit unserer Erkenntnis konfrontiert, und erhielten die Antwort, dieses Implementierungsdetail stelle tatsächlich ein Versehen ihrerseits dar. Es steht also zu erwarten, dass das Verhalten in einer zukünftigen Version mittels dynamischem Binden korrigiert wird, wie dies in *Eiffel for .NET* schon korrekt der Fall ist.

## 5.11 Rekursive und Reflexive Zusicherungen

Im folgenden Abschnitt möchten wir uns mit der Frage beschäftigen, ob und wann es sinnvoll ist, Zusicherungen einer Methode innerhalb der Evaluierung von Zusicherungen einer anderen Methode durchzuführen und welche Probleme dabei auftreten können.

### 5.11.1 Umsetzung in Eiffel

In Eiffel werden Zusicherungen, die innerhalb der Evaluierung einer übergeordneten Zusicherung aufgerufen werden, nicht überprüft. Dies wird durch die *Assertion Evaluation Rule* [21] definiert, die besagt, während der Evaluierung einer Zusicherung zur Laufzeit, müssen Funktionsaufrufe ohne Evaluierung ihrer Zusicherungen ausgeführt werden.

Meyer begründet dies wie folgt:

If a call to  $f$  occurs as part of assertion checking for  $r$ , that is too late to ask whether  $f$  satisfies its assertions. The proper time for such a question is when you decide to use  $f$  in the assertions applicable to  $r$ .

Für das oben genannte Beispiel kann auch folgende Analogie aus der Realität betrachtet werden. Man nehme an,  $f$  sei ein Sicherheitsbeamter am Eingang eines Atomkraftwerkes, dessen Aufgabe es ist, die Personalien der Besucher zu kontrollieren. Natürlich gibt es bestimmte Anforderungen an den Sicherheitsbeamten, welche erfüllt sein müssen, damit sichergestellt ist, dass er seine Aufgabe verantwortungsbewusst erfüllen kann. Diese Anforderungen sollten jedoch bereits im Vorfeld überprüft werden und nicht erst während der Beamte eine Besucherkontrolle durchführt.

Die Strategie, die Meyer vorschlägt, ist extrem, denn durch ein Nichtüberprüfen der Zusicherungen der Methode  $f$  wird implizit angenommen, der Code von  $f$  sei korrekt. Wenn jedoch davon ausgegangen wird, dass  $f$  korrekt ist, warum sollte dann nicht auch angenommen werden, dass auch  $r$  korrekt ist, und somit auf *Design by Contract* verzichtet werden.

Unserer Meinung nach ist die in Eiffel implementierte Lösung des „blinden Vertrauens“ nicht optimal, da es Fälle gibt, an denen die Contracts von Zusicherungen ebenfalls evaluiert werden sollten.

### 5.11.2 Unendliche Rekursion

Nichtsdestotrotz gibt es Situationen, in denen die Evaluierung des Contracts einer Zusicherung offensichtlich der falsche Weg ist. Nämlich genau dann, wenn dies in eine endlose Rekursion führt. Wir unterscheiden zwischen zwei verschiedenen Ursachen für eine endlose Rekursion:

#### Invariant Recursion

Als erste Ursache betrachten wir Invarianten, welche einen Methodenaufruf beinhalten. Invarianten werden per Definition immer zu Beginn und vor Verlassen einer Methode überprüft, um sicherzustellen, dass sich das Objekt in einem gültigen Zustand befindet. Wird nun eine Hilfsprozedur innerhalb einer Invariante aufgerufen, so führt dies offensichtlich zu einer endlosen Rekursion, da vor Ausführung der Prozedur die Invariante abermals überprüft werden muss, was wieder den Aufruf dieser Hilfsprozedur zur Folge hat, usw.

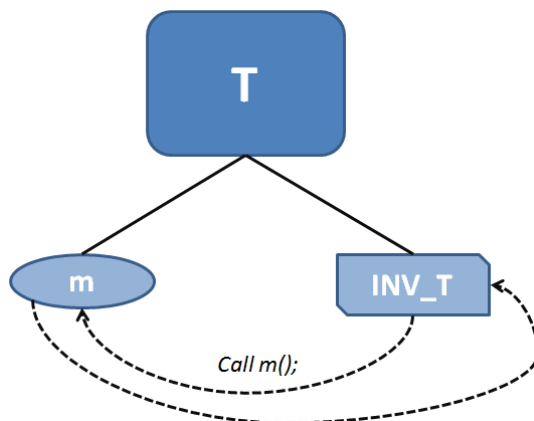


Abbildung 5.7: Invariantenrekursion zwischen der Methode  $T.m$  und der Invariante  $INV_T$ .

Listing 5.15: Invariant Recursion

```

[ContractInvariantMethod]
private void ObjectInvariant() {
    Contract.Invariant(CheckSum()); //call Helperfunction CheckSum()
    ...
}

//Helperfunction to check if Saldo is correct
[Pure]
private bool CheckSum() {
    return Saldo == _einzahlungen.Sum(b => b.Betrag) - _abhebungen.Sum(b => b.
        Betrag);
}
  
```

Der Aufruf der Funktion `Checksum()` in der Invariante würde also zu einer unendlichen Rekursion führen, weil die Invariante vor der Ausführung von `Checksum()` abermals evaluiert werden muss, was einen weiteren Aufruf von `Checksum()` bedeutet.

### Mutual Recursion

Wechselseitige oder verschränkte Rekursion tritt auf, wenn eine Funktion  $r$  eine Funktion  $f$  aufruft, welche dann wieder auf  $r$  verweist. Diese Art der Rekursion ist selbst für einen erfahrenen Programmierer oft schwer zu überblicken und kann leicht zu Fehlern und nichtterminierendem Verhalten führen. Wesentlich schwieriger ist es jedoch, verschränkte Rekursion zu erkennen, wenn die wechselseitigen Aufrufe nicht ausschließlich im Rumpf der Methode, sondern in Vor- bzw. Nachbedingungen eines Contracts geschehen.

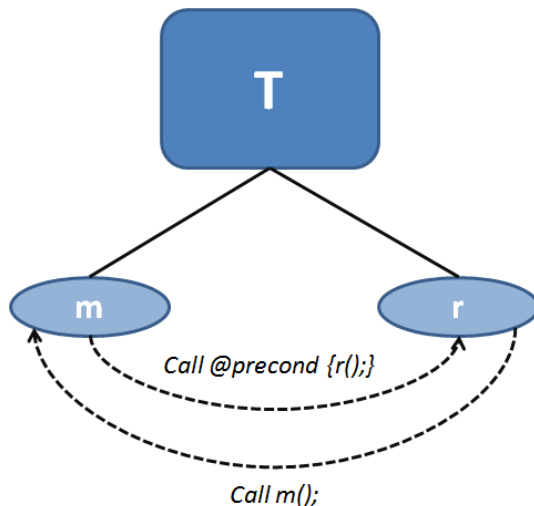


Abbildung 5.8: Indirekte Rekursion zwischen der Methode  $T.m$  und der Hilfsprozedur  $T.r$ .

Betrachten wir als einfaches Beispiel einen Stack, der die beiden Methoden `HasElements()` und `IsEmpty()` bereitstellt. Beide Methoden sind zueinander dual, demnach macht es Sinn, nur in einer der beiden Methoden die eigentliche Logik zu implementieren und in der anderen diese Methode aufzurufen und ihr negiertes Resultat zurückzuliefern. In unserem Beispiel ermittelt die Methode `HasElements()`, ob sich Elemente am Stack befinden und sichert in ihrer Postcondition zu, dass wenn sich Elemente am Stack befinden, dieser nicht leer ist. Die Methode `IsEmpty()` beschränkt sich auf einen Aufruf von `!HasElements()`. Das Verhalten der beiden Methoden führt zu einer endlosen Rekursion und somit zu einem nichtterminierendem Verhalten unseres Programms. Listing 5.16 zeigt den Programmcode in C# Notation:

Listing 5.16: Mutual Recursion

```

private LinkedList<T> elements = new LinkedList<T>();

public void Push(T element) { /* push element onto the stack */ }
public T Pop() { /* pop element from the stack */ return default(T); }

public bool HasElements()
{
    Contract.Ensures(!IsEmpty() || Contract.Result<bool>() == false);
    Contract.Ensures(IsEmpty() || Contract.Result<bool>() == true);
    return elements.Count != 0;
}

public bool IsEmpty()
{
    return !HasElements();
}
  
```

### 5.11.3 Technische Umsetzung

Wie bereits in den vorigen Abschnitten erwähnt, gibt es unterschiedliche Strategien, wie mit der Evaluierung von Zusicherungen einer Hilfsprozedur umgegangen werden sollte. Wir möchten uns daher jetzt mit den einzelnen Strategien der beiden Programmiersprachen Eiffel und C# genauer befassen.

#### Eiffel

Eiffel verzichtet komplett auf die Evaluierung der Contracts von Zusicherungen, weil davon ausgegangen wird, dass Methoden, die von Zusicherungen als Hilfsprozeduren verwendet werden, sehr simpel und mit Sicherheit korrekt sind.

**Invariant-Recursion** Die endlose Rekursion durch Invarianten wird mit einem einfachen booleschen Flag verhindert. Dabei existiert in Eiffel eine globale `HashTable`, die für jeden Typ speichert, ob dessen Invariante gerade überprüft wird. Betrachtet man den in CIL übersetzten Code aus Listing 5.17 (hier nach C# decompiliert), ist zu erkennen: Vor der Evaluierung der Zusicherungen wird die Methode `ISE_RUNTIME.is_invariant_checked_for` aufgerufen, die für das Bankkonto-Objekt zurückliefert, ob eine Invariantenüberprüfung stattfinden darf oder nicht.

Listing 5.17: Invariant Recursion

```
public static void _invariant ([In] BankKonto Current)
{
    if (!ISE_RUNTIME.is_invariant_checked_for (typeof (BankKonto).TypeHandle))
    {
        /* Invariant Assertions */
    }
}
```

**Mutual-Recursion** Da Eiffel keine Evaluierung von Zusicherungen durchführt, wenn diese von anderen Zusicherungen aufgerufen worden sind, besteht auch für den Fall der wechselseitigen Rekursion eine ähnliche Lösung. Über die Methode `set_in_assertion(flag : Boolean)` wird eine global gültige boolesche Variable gesetzt, die eine Evaluierung des Contracts einer Methode verhindert, wenn gerade Zusicherungen einer anderen Methode überprüft werden.

#### C#

Bei der Implementierung von *Code Contracts* in .NET 4.0 wurde ein liberalerer Ansatz für den Umgang mit rekursiven und reflexiven Zusicherungen gewählt. Während bei der Überprüfung von Invarianten wie bei Eiffel ein boolesches Flag gesetzt wird um Rekursion zu verhindern, wird das Auftreten einer unendlichen wechselseitigen Rekursion mit Hilfe einer Integer-Konstante kontrolliert, die die maximale Tiefe des Aufrufstacks innerhalb von Zusicherungen definiert.

Listing 5.18: Recursion Guard in C#

```
public bool HasElements ()
{
    bool CS$1$0000 = this.elements.Count != 0;
    bool Contract.Result<Boolean>() = CS$1$0000;
    if (--ContractsRuntime.insideContractEvaluation <= 4)
    {
        try
        {
            --ContractsRuntime.insideContractEvaluation++;
            --ContractsRuntime.Ensures (!this.IsEmpty () || !Contract.Result<
                Boolean>(),
                null, "IsEmpty () || _Contract.Result<bool>()_==_false");
            --ContractsRuntime.Ensures (this.IsEmpty () || Contract.Result<
                Boolean>(),
                null, "IsEmpty () || _Contract.Result<bool>()_==_true");
        }
        finally
    }
```

```

    {
        __ContractsRuntime.insideContractEvaluation--;
    }
}
return Contract.Result<Boolean>();
}

```

Listing 5.18 zeigt die Funktion `HasElements` unseres Stack-Beispiels.

Die Variable `__ContractsRuntime.insideContractEvaluation` gibt an, in welcher Tiefe des Aufrufstacks wir uns gerade befinden. Was hier auf den ersten Blick etwas merkwürdig wirkt, ist die Bedingung, dass `insideContractEvaluation <= 4` gelten muss. Die Zahl vier ist per Default von `crewrite` als *Recursion Guard* gewählt, kann jedoch beim Aufruf von `crewrite` selbst mit einem Parameter gewählt werden.

Unserer Meinung nach ist die Vorgehensweise rekursive Zusicherungen nicht komplett zu ignorieren, sondern deren Verschachtelungstiefe mit einer Konstante zu begrenzen, wesentlich besser, da die endlose Rekursion effizient verhindert wird. Für nichtrekursive Aufrufe von Hilfsprozeduren, welche selbst Zusicherungen besitzen, werden diese dadurch trotzdem evaluiert.

#### 5.11.4 Zusammenfassung und Vergleich

Im Umgang mit rekursiven bzw. reflexiven Zusicherungen ist ein deutlicher Unterschied zwischen den Implementierungen in den beiden Sprachen Eiffel und C# auszumachen. Eiffel-Vater Bertrand Meyer hält das Überprüfen von Zusicherungen innerhalb der Evaluierung von anderen Zusicherungen für falsch und unnötig, da dies dem Ansatz widerspricht, dass Zusicherungen auf einer höheren Ebene des Codes einzuordnen sind, weil sie dazu dienen den eigentlichen Sourcecode zu überprüfen und dessen Korrektheit sicherzustellen. Meyer geht deshalb davon aus, dass Code, der von Zusicherungen verwendet wird, einfach und unbestreitbar korrekt ist und daher nicht mit Contracts abgesichert werden muss. Das widerspricht jedoch dem Konzept von *Design by Contract*, denn wenn angenommen wird, der Code von Hilfsprozeduren sei korrekt, stellt sich die Frage, warum dies nicht für den gesamten Programmcode gilt und somit auf den Einsatz von *Design by Contract* verzichtet werden kann.

Die Entwickler des *Code Contracts*-Projekts scheinen dieses Problem ebenfalls erkannt zu haben und wählten daher einen flexibleren Ansatz. Durch den Einsatz einer *Recursion Guard* wird die Evaluierung von Zusicherungen in Hilfsprozeduren auf eine bestimmte Verschachtelungstiefe begrenzt, aber nicht komplett darauf verzichtet. Unserer Ansicht nach ist das der bessere Ansatz mit rekursiven Zusicherungen umzugehen und trotzdem eine endlose Rekursion zu verhindern.

## 5.12 Generics

### 5.12.1 C#

Wird wie bisher auf generische Programmierung verzichtet, kann davon ausgegangen werden, dass der von `crewrite` generierte Bytecode eines Obertyps *T* zur Contract-Prüfung einfach in den erbbenden Untertyp *U* übernommen (kopiert) werden kann. Bei generischen Typen gestaltet sich das Bytecode-Rewriting jedoch komplexer. Wir wollen dies an einem einfachen Not-`null`-Check und einem Hilfsmethoden-Aufruf illustrieren.

#### Nicht-generisch

Betrachten wir zunächst folgendes nicht-generische Beispiel:

Listing 5.19: Contract-Vererbung in nicht-generischen Klassen

```

class Person { public string Name { get; set; } public int Age { get; set; } }

class MyList {
    public virtual void Add(object item) {
        Contract.Requires(item != null);
        Contract.Requires(SomethingFancy(item));
    }
}

```

```

        // Add item...
    }

    [Pure]
    public bool SomethingFancy(object item) { /* Check item for something fancy
        . */ }
}

class PersonList : MyList {
    public int CumulatedAge { get; private set; }

    public override void Add(object person) {
        Person p = person as Person;

        base.Add(p);
        CumulatedAge += p.Age;
    }
}

```

Hier lautet der generierte Bytecode (sowohl in `MyList` als auch in `PersonList`) wie folgt:

Listing 5.20: Contract-Vererbung in nicht-generischen Klassen: Bytecode für den Not-null-Check

```

.method public hidebysig virtual instance void Add(object item) cil managed {
    ...
    L_0017: ldarg.1
    L_0018: ldnull
    L_0019: ceq
    ...
}

```

Der Code lädt das erste Argument `person` auf den Stack (`ldarg.1`), lädt `null` auf den Stack (`ldnull`) und vergleicht die beiden (`ceq`).

Listing 5.21: Contract-Vererbung in nicht-generischen Klassen: Bytecode für den Hilfsmethoden-Aufruf

```

.method public hidebysig virtual instance void Add(object person) cil managed {
    ...
    L_002b: ldarg.1
    L_002c: call instance bool GenericsTestNon.MyList::SomethingFancy(object)
    ...
}

```

Auch hier wird das erste Argument `person` auf den Stack (`ldarg.1`) gepusht (diesmal zur Parameter-Übergabe), und anschließend die Methode `SomethingFancy` aufgerufen.

## Generisch

Nun die generische Variante im Vergleich:

Listing 5.22: Contract-Vererbung in generischen Klassen

```

class Person { public string Name { get; set; } public int Age { get; set; } }

class MyList<T> {
    public virtual void Add(T item) {
        Contract.Requires(item != null);
        Contract.Requires(SomethingFancy(item));

        // Add item...
    }

    [Pure]
    public bool SomethingFancy(T item) { /* Check item for something fancy. */
    }
}

class PersonList : MyList<Person> {
    public int CumulatedAge { get; private set; }
}

```

```

    public override void Add(Person person) {
        base.Add(person);
        CumulatedAge += person.Age;
    }
}

```

Listing 5.23: Contract-Vererbung in generischen Klassen: Bytecode für den Not-null-Check im Obertyp

```

.method public hidebysig newslot virtual instance void Add(!T item) cil managed
{
    ...
    L_0017: ldarg.1
    L_0018: box !T
    L_001d: ldnull
    L_001e: ceq
    ...
}

```

Listing 5.24: Contract-Vererbung in generischen Klassen: Bytecode für den Not-null-Check im Untertyp

```

.method public hidebysig virtual instance void Add(class GenericsTest.Person
    person) cil managed {
    ...
    L_0017: ldarg.1
    L_0018: box GenericsTest.Person
    L_001d: ldnull
    L_001e: ceq
    ...
}

```

Da der Typparameter *T* im Allgemeinen auf einen *value type* verweisen kann, muss hier ein Boxing vorgenommen werden. [28] Weil der Zieltyp dieser Operation in Unter- und Obertyp unterschiedlich referenziert wird (einmal als Typparameter *!T*, einmal als konkreter Typ *GenericsTest.Person*), muss die korrekte Bezeichnung von *ccrewrite* aus der Klassensignatur extrahiert werden. Dies muss – aufgrund der Funktionsweise von *ccrewrite* – auf Bytecode-Basis geschehen.

Listing 5.25: Contract-Vererbung in generischen Klassen: Bytecode für den Hilfsmethoden-Aufruf im Obertyp

```

.method public hidebysig newslot virtual instance void Add(!T item) cil managed
{
    ...
    L_0030: ldarg.1
    L_0031: call instance bool GenericsTest.MyList`1<!T>::SomethingFancy(!0)
    ...
}

```

Listing 5.26: Contract-Vererbung in generischen Klassen: Bytecode für den Hilfsmethoden-Aufruf im Untertyp

```

.method public hidebysig virtual instance void Add(class GenericsTest.Person
    person) cil managed {
    ...
    L_0030: ldarg.1
    L_0031: call instance bool GenericsTest.MyList`1<class GenericsTest.Person>::
        SomethingFancy(!0)
    ...
}

```

Auch hier muss, um die korrekt qualifizierte Methode *SomethingFancy* anzugeben, der Typparameter korrekt eingesetzt werden.

## 5.12.2 Eiffel

Der *Eiffel for .NET*-Compiler wendet auf generische Klassen eine homogene Übersetzung an: eine generische Klasse *MYLIST[T]* wird nach den Regeln der homogenen Übersetzung im CIL-Code auf eine nicht-generische Klasse *MylistReference* abgebildet. (Für Wertetypen werden eigene Klassen – zum Beispiel *MylistInteger\_32* – erzeugt; vermutlich um Boxing/Unboxing zu vermeiden.)

Zwar nutzt Eiffel dadurch nicht die Unterstützung von Generizität in der *Common Language Runtime*, das Problem generische Klassen und Methoden korrekt instanzieren zu müssen, wird dadurch aber umgangen.

### 5.12.3 Zusammenfassung und Vergleich

Konkrete Probleme im Zusammenspiel von DbC und generischer Programmierung konnten nicht gefunden werden. Bezüglich der C#-Implementierung ist hervorzuheben, dass die korrekte Instanzierung der Zusicherungen durch Angabe des Typparameters eine technische Herausforderung darstellt; der generierte Code zur Zusicherungs-Prüfung kann nicht 1:1 aus dem Obertyp kopiert werden. Zur Eiffel-Implementierung ist anzumerken, dass auf CIL-Ebene die Typsicherheit aufgegeben wird und zusätzliche Casting-Operationen notwendig sind.

## Kapitel 6

# Interoperabilität Eiffel – Code Contracts

### 6.1 Idee

In diesem Abschnitt wollen wir die Interoperabilität der *.NET Code Contracts* mit der Programmiersprache Eiffel testen. Da die *Code Contracts* grundsätzlich in allen .NET Sprachen einsetzbar sein sollen und wir für unsere Arbeit die *Eiffel for .NET*-Toolchain verwenden, möchten wir testen, ob das Konzept von *Design by Contract* in Eiffel auch über den Einsatz der `Contract` Klasse und ihrer *Marker-Methoden* möglich ist.

### 6.2 Umsetzung

Die technische Umsetzung zum Testen von Interoperabilität, ist denkbar einfach. Abbildung 6.1 zeigt die Vorgehensweise, um Contract-Checking durch *Code Contracts* in Eiffel zu implementieren.

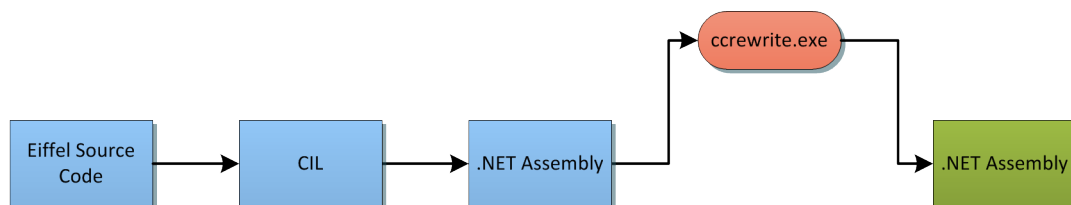


Abbildung 6.1: Ablaufdiagramm

Wir schreiben also ein Eiffel Programm, das anstatt der mitgelieferten Contract-Checking Elemente, die Marker-Methoden der `Contract`-Klasse im .NET Framework verwendet. Dazu müssen wir in unserem Eiffel-Projekt die notwendige Assembly mit dem Namen `Microsoft.Contracts` aus dem GAC<sup>1</sup> importieren. Bevor wir nun die Markermethoden wie in C# verwenden können, verlangt Eiffel, dass wir eine lokale Variable vom Typ `CONTRACT` in jener Methode definieren, in der wir die `Contract`-Klasse verwenden möchten. Listing 6.1 zeigt die Methode `hasElements()` aus unserem Stack-Beispiel mit der Verwendung von Code Contracts.

---

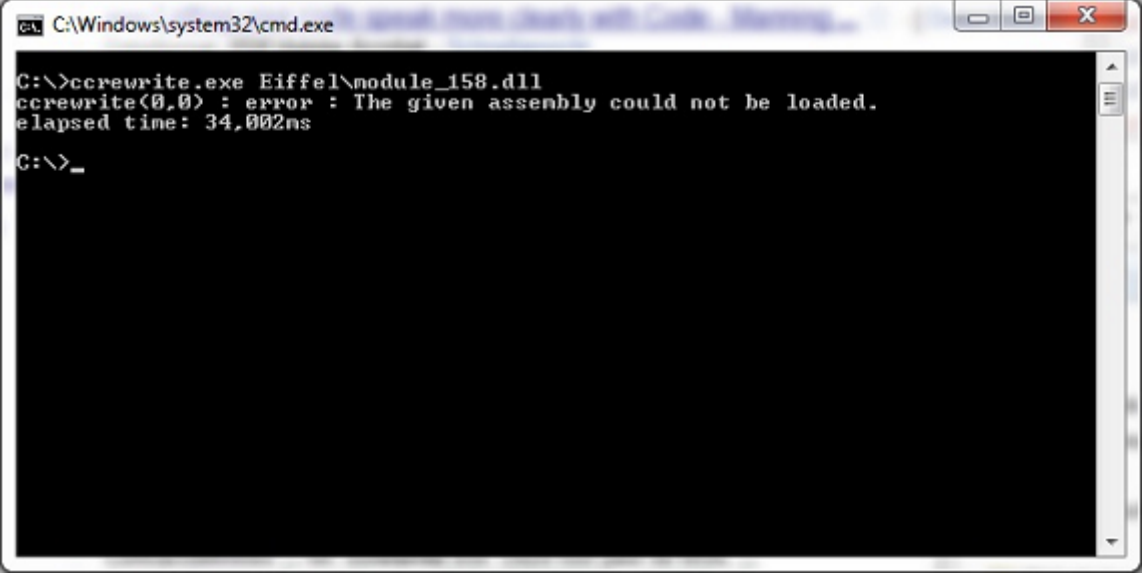
<sup>1</sup>Global Assembly Cache

```

hasElements() : BOOLEAN
local
  contract : CONTRACT
do
  contract.ensures(Result = False or not isEmpty());
  contract.ensures(Result = true or isEmpty());
  Result := (elements.count = 0)
end

```

Das Projekt lässt sich auch mit den Aufrufen der Marker-Methoden fehlerfrei kompilieren. Im nächsten Schritt müssen wir also `crewrite` mit der kompilierten Assembly aufrufen, das Programm sollte dann den Bytecode der Assembly nach Aufrufen von Marker-Methoden der `Contract`-Klasse durchsuchen und diese durch den eigentlichen Contract-Checking Code ersetzen. Im Unterschied zu einer kompilierten .NET-Assembly wird bei einem kompilierten Eiffel-Projekt nicht nur eine `.exe`-Datei erzeugt sondern einige weitere `.dll`-Dateien. Nach dem wir die erzeugte Assembly mit einem Disassembler geladen haben, konnten wir feststellen, dass die Stack-Klasse, welche die Aufrufe der Marker-Methoden enthält, in einer dieser `.dll`-Dateien zu finden ist. Daher muss `crewrite` nicht auf die `.exe`-Datei, sondern auf die entsprechende DLL ausgeführt werden. Abbildung 6.2 zeigt den Aufruf und die Ausgabe von `crewrite.exe`.



```

C:\Windows\system32\cmd.exe
C:\>crewrite.exe Eiffel\module_158.dll
crewrite(0,0) : error : The given assembly could not be loaded.
elapsed time: 34,002ns
C:\>_

```

Abbildung 6.2: Aufruf von `crewrite.exe`

### 6.3 Ergebnis

Wie aus Abbildung 6.2 entnommen werden kann, schlägt der Aufruf von `crewrite` fehl, mit der Fehlermeldung, dass die angegebene Assembly nicht geladen werden kann. Da aufgrund dieser Fehlerbeschreibung keine aussagekräftige Begründung für diesen Fehler ersichtlich ist und auch im Internet noch keine Erfahrungsberichte für diesen Test vorliegen, können wir nur Vermutungen über die Gründe für das Auftreten des Fehlers anstellen.

Der naheliegendste Grund für das Problem ist wohl die Tatsache, dass der *Eiffel-for-.NET*-Compiler den Code nicht in eine einzige `.exe` Assembly kompiliert, sondern mehrere `.dll`-Dateien erzeugt, welche den übersetzten Source-Code enthalten. Diese `.dll`-Dateien dürften nicht einer von `crewrite` erwarteten Assembly entsprechen, was zu oben beschriebenem Fehler führt.

# Kapitel 7

## Performancevergleich

Um einen aussagekräftigen Vergleich zwischen der in .NET 4.0 eingeführten *Code Contracts* in C# und der erstmaligen Implementierung des Konzepts von *Design by Contract* in Eiffel anstellen zu können, muss neben der Bewertung des Funktionsumfangs auch eine Performancemessung durchgeführt werden. Dadurch wird ein direkter Vergleich in Form von Zahlenwerten erst möglich.

### 7.1 Ausgangslage

Um den Test für beide Programmiersprachen unter den selben Bedingungen durchzuführen, verwenden wir als Testprogramm unser Beispiel mit dem Bankkonto. Das Testprogramm wurde in beiden Programmiersprachen implementiert und besitzt denselben Funktionsumfang und die gleichen Zusicherungen. Sämtliche Zusicherungen führen nur Operationen mit konstantem Aufwand aus. Zudem wurde Wert auf vergleichbar effiziente Datenstrukturen gelegt.

#### 7.1.1 Testszenario

Das Testszenario sieht wie folgt aus:

- Ein leeres Bankkonto wird erstellt.
- Es werden 10000 Einzahlungen mit einem Betrag von 250 Euro getätigt.
- Es werden 10000 Abhebungen von 250 Euro durchgeführt.

Dieses Testszenario wird 20 mal iteriert, wobei für jede Iteration ihre Ausführungszeit in Millisekunden gemessen wird. Anschließend wird die durchschnittliche Ausführungszeit einer Iteration berechnet und als Messwert zum Vergleich herangezogen.

#### Stopwatch

Für das exakte Messen der Ausführungszeit wird die im .NET Framework enthaltene Klasse `Stopwatch` verwendet. Da wir bei unserer Arbeit den Eiffel/.NET Compiler einsetzen, können wir die Assembly, in der die `Stopwatch`-Klasse enthalten ist, auch einfach in unser Eiffel-Programm einbinden und die Klasse problemlos verwenden. Listing 7.1 und Listing 7.2 zeigen die Main-Methoden des Bankkonto-Beispiels, in denen die Performancemessung durchgeführt wird.

Listing 7.1: Stopwatch in C#

```
static void Main(string[] args)
{
    List<long> executionTimes = new List<long>();
    //Stop Program Execution Time
    Stopwatch watch = new Stopwatch();
    //Start Watch
    for (int j = 1; j <= 20; j++)
    {
```

```

    watch.Restart();
    BankKonto konto = new BankKonto(true);

    //10001 * 250 Euro einzahlen
    for (int i = 0; i <= 10000; i++)
        konto.Einzahlen(250);

    //10001 * 250 Euro abheben
    for (int i = 0; i <= 10000; i++)
        konto.Abheben(250);

    //Stop Watch
    watch.Stop();

    executionTimes.Add(watch.ElapsedMilliseconds);
}

//Print Result
System.Console.WriteLine("Execution_Time:_" + executionTimes.Average());
System.Console.ReadLine();
}

```

Listing 7.2: Stopwatch in Eiffel

```

local
    execution_times : INTEGER_64
    watch : STOPWATCH
    j, i : INTEGER
    bew : BEWEGUNG

    konto : BANKKONTO

do
    create watch.make
    execution_times := 0

    from j := 1 until j > 20 loop
        watch.reset
        watch.start

        create konto.make (true)

        from i := 0 until i > 10000 loop
            bew := konto.einzahlen (250)
            i:=i+1
        end

        from i := 0 until i > 10000 loop
            bew := konto.abheben (250)
            i:=i+1
        end

        watch.stop

        execution_times := execution_times + watch.elapsed_milliseconds
        print(j)
        j := j+1
    end

    print ("%N")
    print (execution_times / 20)
    io.read_line

end

```

## Testfälle

Das TestszENARIO wird auf mehrere Varianten des Contract-Checking angewendet, um nicht nur einen Vergleich zwischen den beiden Programmiersprachen anstellen zu können, sondern auch

um Performanceunterschiede zwischen aktiviertem und deaktiviertem Contract-Checking bei einer Programmiersprache festzustellen.

**Full** Alle Varianten von Zusicherungen werden evaluiert (Pre - und Postconditions, Invarianten, Assertions)

**Pre & Post** Nur die Vor - und Nachbedingungen werden überprüft

**Off** Es wird kein Contract-Checking durchgeführt

## 7.2 Ergebnisse

Testfall	Durchschnittliche Ausführungszeit pro Iteration	
	<i>C#</i>	<i>Eiffel</i>
<b>Full</b>	50,85ms	726,15ms
<b>Pre &amp; Post</b>	55,75ms	512,0ms
<b>Off</b>	31,05ms	315,2ms

Tabelle 7.1: Ergebnisstabelle

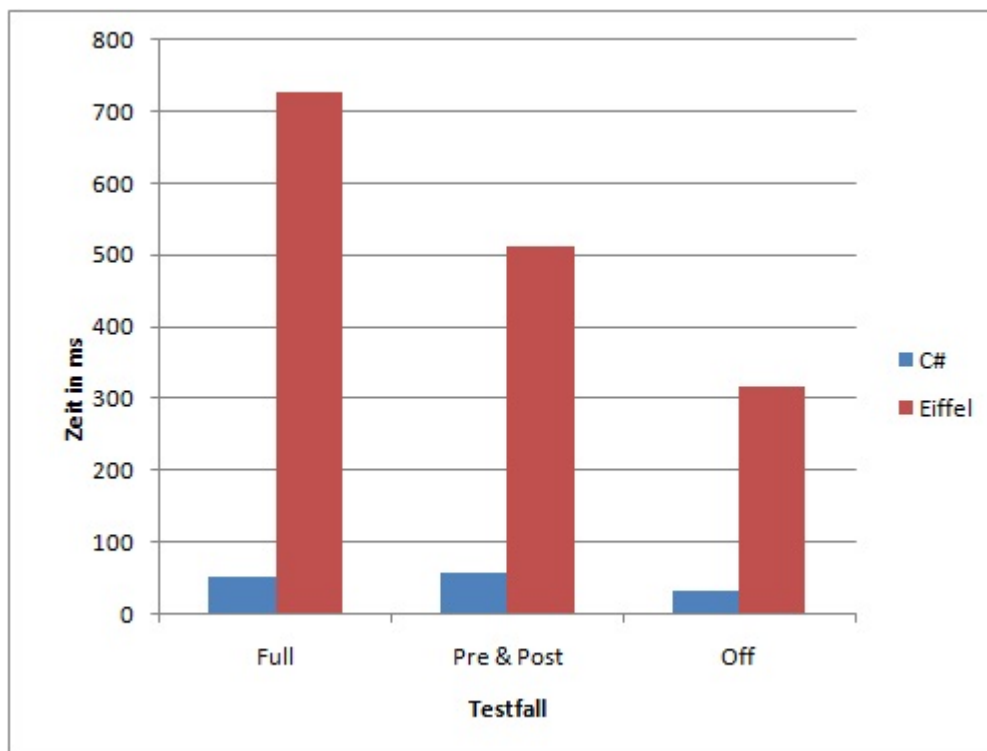


Abbildung 7.1: Performancevergleich der beiden Sprachen

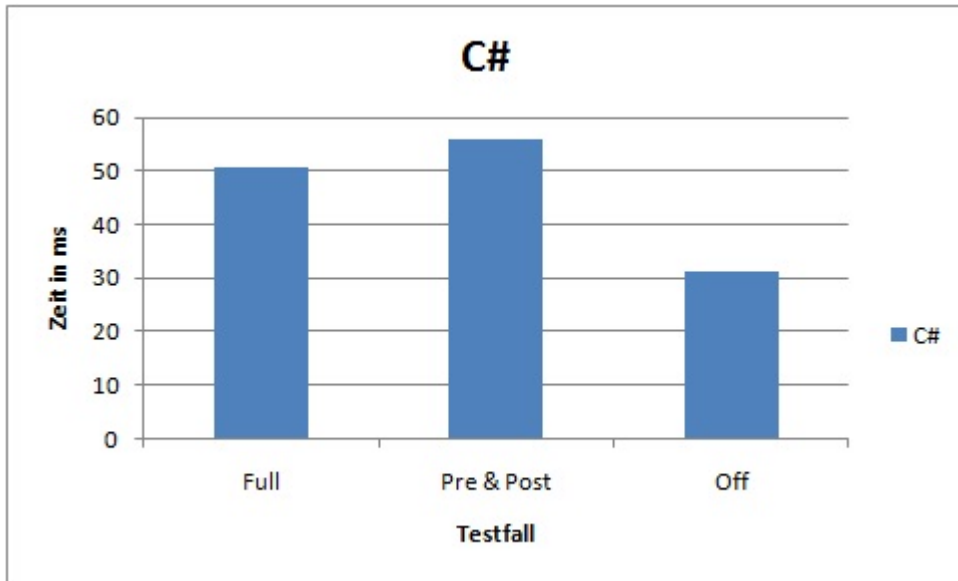


Abbildung 7.2: Vergleich der Testfälle in C#

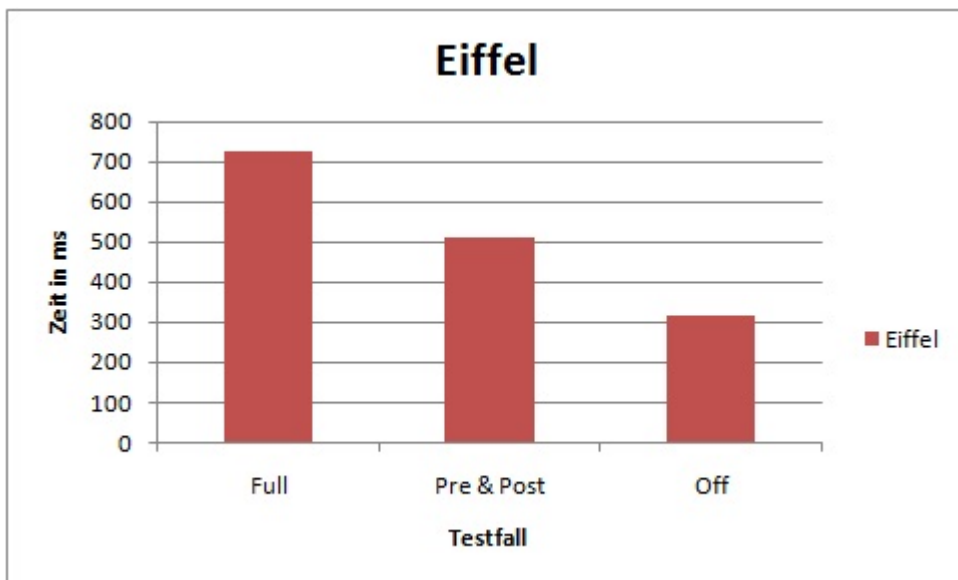


Abbildung 7.3: Vergleich der Testfälle in Eiffel

## 7.3 Analyse der Ergebnisse

### 7.3.1 Inkonsistente Performance in C#

Abbildung 7.2 zeigt einen unerwarteten Geschwindigkeitsunterschied zwischen den beiden Contract-Checking Varianten *Full* und *Pre & Post*. Dieser Unterschied ist logisch nicht nachvollziehbar, da bei der *Full*-Variante zusätzlich zu den Vor- und Nachbedingungen auch Invarianten und Assertions überprüft werden. Trotzdem ist die durchschnittliche Ausführungszeit bei der *Pre & Post*-Variante um etwa 10% länger als bei voll aktiviertem Contract-Checking.

Durch Disassemblierung des von den beiden unterschiedlichen Contract-Checking Varianten generierten Bytecodes konnten wir den Grund für diese Anomalie feststellen:

Listing 7.3: Getter- und Setter-Methode der Saldo-Property

```
[CompilerGenerated]
public decimal get_Saldo()
{
    decimal num2 = this.<Saldo>k__BackingField;
    if ((__ContractsRuntime.insideContractEvaluation <= 1) && !this.
        $evaluatingInvariant$)
    {
        try
        {
            __ContractsRuntime.insideContractEvaluation++;
            __ContractsRuntime.Ensures(this.ueberziehbar || (num2 >= 0M), null,
                "ueberziehbar_||_Saldo_>=_0");
        }
        finally
        {
            __ContractsRuntime.insideContractEvaluation--;
        }
    }
    return num2;
}

[CompilerGenerated]
protected void set_Saldo(decimal value)
{
    if ((__ContractsRuntime.insideContractEvaluation <= 4) && !this.
        $evaluatingInvariant$)
    {
        try
        {
            __ContractsRuntime.insideContractEvaluation++;
            __ContractsRuntime.Requires(this.ueberziehbar || (value >= 0M),
                null, "ueberziehbar_||_Saldo_>=_0");
        }
        finally
        {
            __ContractsRuntime.insideContractEvaluation--;
        }
    }
    this.<Saldo>k__BackingField = value;
}
```

Listing 7.3 zeigt die vom Compiler generierten `get`- und `set`-Methoden der `Saldo`-Property. `crewrite.exe` kopiert die Invariantenzusicherungen lokal in den Rumpf dieser beiden Methoden, anstatt an dieser Stelle (wie in anderen Methoden) `$InvariantMethod$()` aufzurufen. Diese Invariantenzusicherungen werden jedoch auch bei abgeschaltetem Invariant-Checking (nur *Pre & Post*) in die Getter- und Setter-Methoden kopiert, weshalb wir auf ein fehlerhaftes Verhalten von `crewrite` schließen.

Das oben beschriebene Fehlverhalten erklärt aber noch nicht den unerwarteten Performanceunterschied zwischen den beiden Contract-Checking Varianten. Der Grund dafür ist an einer anderen Stelle des Bytecodes zu finden:

Listing 7.4: BankKonto.Einzahlen(decimal betrag) Full

```

public virtual Bewegung Einzahlen(decimal betrag)
{
    ...
    bool flag = this.$evaluatingInvariant$;
    this.$evaluatingInvariant$ = true;
    try
    {
        saldo = this.Saldo;
    }
    ...
    this.Saldo += betrag;
    ...
    if (Interlocked.Increment(ref this.$evaluatingInvariant$) <= 4)
    {
        try
        {
            Interlocked.CompareExchange(ref this.Saldo, (saldo + num2), saldo);
            ...
        }
        finally
        {
            Interlocked.Decrement(ref this.$evaluatingInvariant$);
        }
    }
    ...
    this.$evaluatingInvariant$ = flag;
    this.$InvariantMethod$();
    return bewegung3;
}

```

Listing 7.5: BankKonto.Einzahlen(decimal betrag) Pre &amp; Post

```

public virtual Bewegung Einzahlen(decimal betrag)
{
    ...
    try
    {
        saldo = this.Saldo;
    }
    ...
    this.Saldo += betrag;
    ...
    if (Interlocked.Increment(ref this.$evaluatingInvariant$) <= 4)
    {
        try
        {
            Interlocked.CompareExchange(ref this.Saldo, (saldo + num2), saldo);
            ...
        }
        finally
        {
            Interlocked.Decrement(ref this.$evaluatingInvariant$);
        }
    }
    ...
    return bewegung3;
}

```

In Listing 7.4 ist zu sehen, wie vor der Ausführung des Methodencodes das Boolean-Flag `this.$evaluatingInvariant$` auf `true` gesetzt wird, um anzuzeigen, dass innerhalb dieser Methode am Ende eine Invariantenprüfung durchgeführt wird. Das Flag soll eine unendliche Invariantenrekursion verhindern, indem die Invariante nur einmal am Ende der Methode `Einzahlen(decimal betrag)` überprüft wird, und bei möglichen Methodenaufrufen innerhalb von `Einzahlen` keine Invariantenprüfung durchgeführt wird. Deshalb wird bei jedem Zugriff auf die Property `Saldo` (lesend oder schreibend) der dort enthaltene Zusicherungsblock (siehe 7.3) nicht evaluiert.

Bei der *Pre & Post*-Variante ist den Entwicklern offenbar ein entscheidender Denkfehler passiert. Korrekterweise fehlt bei dieser Variante das Setzen des Flags `this.$evaluatingInvariant$` und der Aufruf von `this.$InvariantMethod$()` am Ende der Methode `Einzahlen`, jedoch haben die Entwickler dabei übersehen, dass der in den `get`- und `set`-Methoden enthaltene Zusicherungsblock bei jedem Zugriff auf `this.Saldo` ausgeführt wird.

Da bei jeder Iteration unseres Performancetests ein mehrmaliger Zugriff auf diese Property stattfindet, ist die nicht beabsichtigte Evaluierung dieses Zusicherungsblocks im Getter und Setter der Grund für die unerwarteten Laufzeitunterschiede zwischen den beiden Contract-Checking Varianten.

### 7.3.2 Schlechte Performance von *Eiffel for .NET*

Zudem fällt auf, dass die *Eiffel for .NET*-Implementierung jeweils um etwa eine Größenordnung langsamer ist als ihr C#-Pendant. Das ist insofern überraschend, als das Sprachdesign von Eiffel zahlreiche Optimierungen zulässt. Speziell der GNU Eiffel-Compiler *SmartEiffel* (vormals *SmallEiffel*) ist für diese Optimierungen bekannt. [29, 6] Da *SmartEiffel* mittlerweile einen anderen Dialekt der Sprache unterstützt als der Compiler von Eiffel Software, ist ein direkter Compiler-Vergleich nicht möglich.

Unsere bisherigen Analysen des vom Eiffel-Compiler erzeugten CIL-Codes legen nahe, dass diese Übersetzung sehr ineffizient erfolgt. Auch die Anzahl von Instruktionen im Kompilat beider Sprachen stützen diese Vermutung: Im *Full*-Modus generiert Eiffel 12274 Zeilen CIL-Code, die C#-Variante kommt mit 2041 aus. Auch die Anzahl der Aufrufe in Eiffel ist mit 529 `call`- und `callvirt`-Instruktionen enorm, C# kommt mit 147 aus. Zwar müssen einige der Eiffel-Features wie Mehrfachvererbung in CIL emuliert werden, dies ist aber unserer Meinung nach unterproportional zum vorgefundenen Overhead.

Nach der Jahrtausendwende schien Eiffel Software zunehmend unter Druck zu geraten, Kompatibilität zu einer der aufstrebenden Plattformen (Java, .NET) herzustellen: Um vorerst am Markt zu bestehen, musste einerseits bestehenden Entwicklern die Arbeit mit Eiffel auf neuen Plattformen ermöglicht werden, andererseits musste bestehende Eiffel-Software auf neuen Plattformen weiterlaufen, um nicht durch andere Systeme ersetzt oder komplett neu entwickelt zu werden.

Unter diesen Umständen dürfte *Eiffel for .NET* unter Zeitdruck entwickelt worden sein, um anderen .NET-Sprachen zuvorzukommen. EiffelStudio 5.1, die erste Version mit .NET-Unterstützung erschien am 2. Dezember 2001, etwa drei Monate später verließ das .NET Framework den Beta-Status. [14, 8] Dabei wurde anscheinend wenig Rücksicht auf performante Übersetzung genommen.

## 7.4 Fazit

Der Performancetest bestätigt die Vermutung, dass in C# entwickelte Applikationen in allen Testfällen deutlich schneller sind, als Anwendungen, die in der doch etwas in die Jahre gekommenen Programmiersprache Eiffel geschrieben wurden. Das Ergebnis des Performancevergleichs lässt sich jedoch nicht ausschließlich auf den Altersunterschied der beiden Sprachen und den damit verbundenen technologischen Fortschritt zurückführen, sondern hängt potentiell mit der nicht optimalen Übersetzung des *Eiffel for .NET*-Compilers in die *Common Intermediate Language* zusammen.

Außerdem kann das Deaktivieren des Contract-Checking einen deutlichen Performance-Gewinn bedeuten. Ob das (zum Beispiel im Produktivbetrieb) sinnvoll ist, kann nur für das jeweilige Projekt abgeschätzt und entschieden werden.

## Kapitel 8

# Schlussbemerkungen

Abschließend hinterlässt das `crewrite`-Tool den Eindruck eines „jungen“ dynamischen Contract-Prüfers, dem noch einige Bugs anhaften (keine Warnung bei privaten Hilfsprozeduren im Obertyp, Probleme bei der Invarianten-Prüfung). Zudem ist er nur für offiziell unterstützte Sprachen anwendbar; sogar im Paper von Fähndrich et al. [11] wird angedeutet, dass zwischen C# und Visual Basic sprachabhängige Unterscheidungen durchgeführt werden müssen.

Dies erscheint aber insgesamt und im Vergleich mit Eiffel als kleine Unzulänglichkeit. Weitreichender sind die beschriebenen Gefahren im Zusammenhang mit objektorientierten Konzepten, wie Überschreiben oder Seiteneffekte. Andererseits begründen viele dieser Aspekte aber die Mächtigkeit des objektorientierten Paradigmas. Eine Diskussion scheint nur bei statischem Binden angebracht, das in vielen Sprachen ohnehin als nicht essentiell für die Objektorientierung behandelt wird.

Aus unseren Erfahrungen können wir festhalten, dass die Definition von Contracts in der Sprache selbst leichter fällt, als mit Marker-Methoden zu hantieren.

Grundsätzlich stellt sich aber die Frage, wie weit die dynamische Prüfung von Zusicherungen überhaupt notwendig ist. DbC gliedert sich in zwei Aspekte:

- die Definition der Objektschnittstelle mit Zusicherungen (als Dokumentation für andere Entwickler, die Qualitätssicherung usw.) sowie
- die tatsächliche Überprüfung der Zusicherungen.

Dabei erscheint die Idee der Contract-Definition weitaus wichtiger als eine konkrete Überprüfung derselben. Speziell bei einer dynamischen Prüfung treten die angesprochenen Probleme und Performanceeinbußen auf. Die zentralen Ziele von *Design by Contract* könnten (bei entsprechender Motivation des Entwicklers) auch einfach mit Contract-Definitionen in Form von Kommentaren erreicht werden.

## Kapitel 9

# Verwandte Themen & Ausblick

Ein großes, nicht behandeltes Gebiet ist das Zusammenspiel von *Design by Contract* und Nebenläufigkeit. Die dazu notwendige grundlegende Betrachtung von nebenläufigen Konzepten in beiden Sprachen (Multithreading, SCOOP [22], Parallel Extensions [10]) hätte den Umfang der Arbeit bei weitem gesprengt.

Die *Code Contracts*-Umgebung stellt zudem ein Tool (`cccheck`) zur statischen Prüfung basierend auf *Abstract Interpretation* [9] zur Verfügung. Das Thema findet wegen mangelnder Vergleichbarkeit mit Eiffel keine Betrachtung in diesem Dokument.

Abschließend sei auf das Projekt *Spec#* von *Microsoft Research* verwiesen. *Spec#* stellt eine Übermenge von *C#* dar, welche das Typsystem erweitert und Sprachkonstrukte für Vor- und Nachbedingungen sowie Objektivarianten bietet. Zusätzlich zur dynamischen Prüfung zur Laufzeit steht eine statische Überprüfung mittels Theorembeweiser zur Verfügung. [2]

# Literatur

- [1] Parker Abercrombie und Murat Karaorman. “jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java”. In: *Electr. Notes Theor. Comput. Sci.* 70.4 (2002).
- [2] Mike Barnett u. a. “The Spec# programming system: An overview.” In: *CASSIS 2004, LN-CS*. Bd. 3362. Springer, 2004.
- [3] *C# Language Specification*. Microsoft Corporation. 2010. URL: <http://msdn.microsoft.com/en-us/library/aa645596.aspx> (besucht am 22. 10. 2010).
- [4] Yu Chin Cheng, Chien-Tsun Chen und Chin-Yun Hsieh. “ezContract: Using Marker Library and Bytecode Instrumentation to Support Design by Contract in Java”. In: *APSEC*. 2007, S. 502–509.
- [5] *Code Contracts User Manual*. Microsoft Corporation. Mai 2010. URL: <http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf> (besucht am 18. 04. 2010).
- [6] Dominique Colnet und Olivier Zendra. “Optimizations of Eiffel Programs: Smalleiffel, the GNU Eiffel Compiler”. In: *TOOLS (29)*. 1999, S. 341–350.
- [7] Microsoft Corporation. *LINQ - Language Integrated Queries*. URL: <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx> (besucht am 03. 10. 2010).
- [8] Microsoft Corporation. *.NET Framework Redistributable*. URL: <http://www.microsoft.com/windows/netframeredist/download.mspx> (besucht am 26. 10. 2010).
- [9] Patrick Cousot und Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*. 1977, S. 238–252.
- [10] Joe Duffy. *Concurrent Programming on Windows*. Addison-Wesley, 2008, S. 887–929.
- [11] Manuel Fähndrich, Michael Barnett und Francesco Logozzo. “Embedded contract languages”. In: *SAC*. 2010, S. 2103–2110.
- [12] Robert W. Floyd. “Assigning meanings to programs”. In: *Mathematical Aspects of Computer Science*. Bd. 19. 1967, S. 19–32.
- [13] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), S. 576–580.
- [14] Interactive Software Engineering, Inc. *ISE to Release EiffelStudio 5.1*. URL: <http://www.eiffel.com/general/news/pdf/12-02-2001.pdf> (besucht am 26. 10. 2010).
- [15] J.-M. Jazequel und B. Meyer. “Design by Contract: The Lessons of Ariane”. In: *Computer* 30.1 (Jan. 1997), S. 129–130.
- [16] Martin Lackner, Andreas Krall und Franz Puntigam. “Supporting Design by Contract in Java”. In: *Journal of Object Technology* 1.3 (2002), S. 57–76.
- [17] Barbara H. Liskov und Jeanette M. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), S. 1811–1841.
- [18] Barbara H. Liskov und Jeanette M. Wing. *Behavioral Subtyping Using Invariants and Constraints*. Techn. Ber. 1999.
- [19] Bertrand Meyer. “Applying ‘Design by Contract’”. In: *Computer* 25.10 (Okt. 1992), S. 40–51.

- [20] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1991.
- [21] Bertrand Meyer. *Object-Oriented Software Construction, second edition*. Prentice Hall, 1997.
- [22] Bertrand Meyer. “Systematic Concurrent Object-Oriented Programming”. In: *Communications of the ACM* 36.9 (1993), S. 56–80.
- [23] Bertrand Meyer, Raphael Simon und Emmanuel Stapf. “Full Eiffel on the .NET Framework”. In: *MSDN* (Juli 2002). URL: <http://msdn.microsoft.com/en-us/library/ms973898.aspx> (besucht am 18.04.2010).
- [24] Jan Newmarch. “Adding contracts to Java”. In: *TOOLS (27)*. Sep. 1998, S. 2–7.
- [25] Oracle Corporation. *Top 25 RFE's (Request for Enhancements)*. URL: [http://bugs.sun.com/bugdatabase/top25\\_rfes.do](http://bugs.sun.com/bugdatabase/top25_rfes.do) (besucht am 14.04.2010).
- [26] Reinhold Plösch. “Design by Contract for Python”. In: *APSEC*. 1997, S. 213–219.
- [27] Reinhold Plösch. “Tool Support for Design by Contract”. In: *TOOLS (26)*. 1998, S. 282–294.
- [28] *Standard ECMA-335 Common Language Infrastructure (CLI)*. 4. Aufl. ECMA. Juni 2006.
- [29] Olivier Zendra, Dominique Colnet und Suzanne Collin. “Efficient Dynamic Dispatch without Virtual Function Tables: The SmallEiffel Compiler”. In: *OOPSLA*. 1997, S. 125–141.