# Spezifikation durch Vertrag - eine Basistechnologie für eBusiness

Vortragsunterlagen für das
Forum Wirtschaftsinformatik 2001 Reutlingen - eBusiness Technologies
22. Mai 2001

Aktualisiert: 9. Mai 2001

**Karlheinz Hug**
Fachhochschule Reutlingen, Hochschule für Technik und Wirtschaft
Fachbereich Elektrotechnik und Maschinenbau, Studiengang Elektronik
Federnseestr. 4, 72764 Reutlingen
E-Mail: karlheinz.hug@fh-reutlingen.de
WWW: http://www-el.fh-reutlingen.de (enthält aktuelle Version dieses Artikels)

**Zusammenfassung**

eBusiness-Anwendungen müssen zuverlässig und vertrauenswürdig sein. Spezifikation durch Vertrag zielt als Softwareentwicklungsmethode auf systematisches Konstruieren von Komponenten hoher Qualität und eignet sich so gut zum Entwickeln von eBusiness-Anwendungen. Dieser Artikel stellt Aspekte der Vertragsmethode exemplarisch vor, diskutiert ihren Einfluss auf den Softwareentwicklungsprozess und gibt einen Überblick über den Stand der Technik in Sprachen, Werkzeugen und Projekten. Insbesondere zeigt er verschiedene Ansätze, Verträge in Java mit Werkzeugen zu unterstützen.

**Schlüsselwörter**

Softwareentwicklungsprozess, Softwaremethode, Spezifikationsmethode, Spezifikation durch Vertrag, Design by Contract, Objekttechnologie, Komponententechnologie, Softwarequalität, Zuverlässigkeit, Korrektheit, Vertrauenswürdigkeit, Softwarewerkzeuge

**Inhalt**

1 Einleitung ... 2
2 Was ist Spezifikation durch Vertrag? ... 3
3 Aspekte der Vertragsmethode ... 5
 3.1 Invarianten, Vor- und Nachbedingungen ... 5
 3.2 Ergebnisse in Nachbedingungen ... 6
 3.3 Vorzustände in Nachbedingungen ... 6
 3.4 Vollständige Spezifikation ... 7
 3.5 Änderungsverbote ... 8
 3.6 Elementare und abgeleitete Abfragen ... 8
 3.7 Verträge und Vererbung ... 10
 3.8 Quantoren ... 10

3.9 Von der Spezifikation zur Implementation ... 11
4 Die Vertragsmethode im Softwareentwicklungsprozess ... 13
4.1 Analyse ... 14
4.2 Entwurf ... 14
4.3 Spezifikation und Dokumentation ... 14
4.4 Implementierung ... 15
4.5 Qualitätssicherung und Test ... 15
4.6 Betrieb, Wartung und Pflege ... 16
5 Sprachen, Werkzeuge und Projekte ... 16
5.1 Eiffel ... 16
5.2 Business Object Notation ... 18
5.3 Object Constraint Language ... 18
5.4 Interface Definition Language ... 20
5.5 Component Pascal ... 21
5.6 C und C++ ... 22
5.7 Java ... 23
5.8 .NET, C# und Eiffel# ... 28
5.9 Weitere Ansätze ... 28
6 Fazit ... 29
Referenzen ... 29

# 1 Einleitung

Die technologische Basis von eBusiness bilden im Internet verteilte Anwendungen. Dabei kooperieren verteilte **Komponenten**, um gemeinsam Aufgaben auszuführen. Jede Komponente erfüllt eine Teilaufgabe. Ein verbreitetes Kooperationsmodell ist das **Kunden-Lieferanten-Modell** (*client-server/supplier model*), eine Metapher aus dem Geschäftsleben. Komponenten erscheinen hier in zwei Rollen:

- Eine Komponente ist **Lieferant** (*server*, *supplier*), indem sie potenziellen Kunden **Dienste** (*service*) bietet.
- Eine Komponente kann **Kunde** (*client*) eines Lieferanten sein, indem sie dessen Dienste benutzt.

Die **Schnittstelle** (*interface*) einer Komponente besteht aus ihren Diensten; sie legt fest, **was** der Lieferant bereitstellen muss *und* **was** ein Kunde erhalten kann. Um benutzbar zu sein, muss eine Schnittstelle explizit und genau beschrieben - **spezifiziert** - sein. Eine **Spezifikation** enthält Informationen, die Entwickler von Kunden zum Benutzen eines Lieferanten benötigen. Eine Spezifikation einer Schnittstelle ist also eine exakte Beschreibung der Dienste, wobei folgende Aspekte zu unterscheiden sind:

- Die **Syntax** umfasst die Namen der Dienste und die Anzahlen, Reihenfolgen und Arten der Parameter.
- Die **statische Semantik** umfasst einerseits Zugriffsrechte, andererseits die Typbindung von Ergebnissen und Parametern von Diensten.
- Die **dynamische Semantik** umfasst das Verhalten der Dienste, ihre Ergebnisse und ihre Wirkungen auf den Zustand der Komponente.

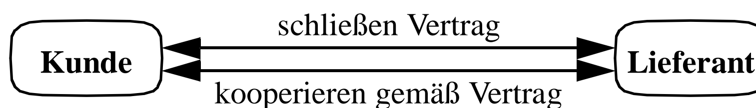
Konventionelle Schnittstellendefinitions- und Implementationssprachen erlauben die Beschreibung der Syntax und statischen Semantik von Diensten durch Signaturen, aber selten mehr. Als Kommentare hinzugefügte informale, verbale Spezifikationen der dynamischen Semantik von Diensten sind oft mehrdeutig und genügen daher nicht. Es fehlen formale, exakte semantische Spezifikationen von Komponenten.

Da mit eBusiness-Anwendungen Geschäfte abgewickelt werden, bei denen viel Geld im Spiel sein kann, ist ein hohes Maß an Zuverlässigkeit und Vertrauenswürdigkeit gefordert. Kein Anwender will Geld verlieren, weil eine Softwarekomponente nicht richtig funktioniert. Zuverlässiges Funktionieren von eBusiness-Anwendungen ist wesentlich. Es setzt korrekte Komponenten voraus. Eine Komponente ist **korrekt**, wenn ihre Implementation ihrer Spezifikation entspricht. Daher ist ohne exakte Spezifikation nicht über Korrektheit zu räsonieren. Dies zeigt, wie wichtig Methoden zur formalen Spezifikation der dynamischen Semantik von Komponentenschnittstellen sind bzw. mit der Verbreitung von eBusiness-Anwendungen werden.

Im Folgenden stellen wir eine solche Methode vor: Spezifikation durch Vertrag. Wir diskutieren ihren Nutzen und ihre Auswirkungen auf den Softwareentwicklungsprozess und geben einen Überblick über den Stand der Technik.

# 2 Was ist Spezifikation durch Vertrag?

**Spezifikation durch Vertrag** (**SdV**, *Design by Contract*[1], *Programming by Contract*) ist eine Methode zur Spezifikation der dynamischen Semantik von Softwarekomponenten mittels Verträgen aus erweiterten booleschen Ausdrücken. SdV basiert auf der Theorie der abstrakten Datentypen und formalen Spezifikationsmethoden. Spezifizierte Komponenten können Module, Klassen oder Komponenten im Sinne von Komponententechnologien (wie Microsoft's COM, .NET oder Sun's EJB) sein [60]. Verträge ergänzen die Metapher des Kunden-Lieferanten-Modells:




Grundlegend für die Vertragsmethode ist das **Prinzip der Trennung von Diensten in Abfragen und Aktionen** (*command-query separation*):

- **Abfragen** geben Auskunft über den Zustand einer Komponente, verändern ihn aber nicht. Sie liefern als Ergebnis einen Wert. Die Abfragen einer Komponente beschreiben ihren abstrakten Zustand.

[1] „Design by Contract“ ist ein Warenzeichen von Interactive Software Engineering.

- **Aktionen** verändern den Zustand einer Komponente, liefern aber kein Ergebnis. Die Aktionen einer Komponente bewirken ihre Zustandsveränderungen.

Diesem Prinzip folgend sind seiteneffektbehaftete Funktionen als Dienste zu vermeiden.[2] Ein Grund dafür ist, dass Abfragen als **Spezifikatoren** dienen, d.h. als Elemente von Verträgen. **Verträge** setzen sich aus Bedingungen folgender Art zusammen:

- **Invarianten** einer Komponente sind allgemeine, unveränderliche Konsistenzbedingungen an den Zustand der Komponente, die vor und nach jedem Aufruf eines Dienstes gelten. Formal sind Invarianten boolesche Ausdrücke über den Abfragen der Komponente; inhaltlich können sie z.B. Geschäftsregeln (*business rule*) ausdrücken.
- **Vorbedingungen** (*precondition*) eines Dienstes sind Bedingungen, die vor einem Aufruf des Dienstes erfüllt sein müssen, damit er ausführbar ist. Vorbedingungen sind boolesche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes.
- **Nachbedingungen** (*postcondition*) eines Dienstes sind Bedingungen, die nach einer Ausführung des Dienstes erfüllt sind; sie beschreiben, welches Ergebnis ein Dienstaufruf liefert oder welchen Effekt er erzielt. Nachbedingungen sind boolesche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes, erweitert um ein Gedächtniskonstrukt, das die Werte von Ausdrücken vor dem Dienstaufruf liefert.

Verträge legen Pflichten und Nutzen für Kunden und Lieferanten fest. Die Verantwortlichkeiten sind klar verteilt: Der Lieferant garantiert die Nachbedingung jedes Dienstes, den der Kunde aufruft, falls der Kunde die Vorbedingung erfüllt. Eine verletzte Vorbedingung ist ein Fehler des Kunden, eine verletzte Nachbedingung oder Invariante ein Fehler des Lieferanten.

| | **Kunde** | **Lieferant** |
|---|---|---|
| **Pflicht** | Die Vorbedingungen einhalten. | Anweisungen ausführen, die die Nachbedingungen herstellen und die Invarianten erhalten. |
| **Nutzen** | Ergebnisse/Wirkungen nicht prüfen, da sie durch die Nachbedingungen garantiert sind. | Aufrufe, die die Vorbedingungen verletzen, ignorieren. (Die Vorbedingungen nicht prüfen.) |

Schwache Vorbedingungen erleichtern den Kunden die Arbeit, starke Vorbedingungen dem Lieferanten. Je schwächer Nachbedingungen sind, umso freier ist der Lieferant und ungewisser über das Ergebnis/den Effekt sind die Kunden. Je stärker Nachbedingungen sind, umso mehr muss der Lieferant leisten und erhalten die Kunden.

Die Vertragsmethode begründet hat Bertrand Meyer, der mit der Programmiersprache Eiffel auch ein hervorragendes Mittel zu ihrer praktischen Anwendung geschaffen hat (s. 5.1) [28], [29], [30], [31], [32], [33], [37]. Jean-Marc Nerson und Kim Waldén haben die

[2] In bestimmten Fällen, z.B. bei Fabrikfunktionen, können Seiteneffekte sinnvoll sein. Solche Funktionen sind nicht als Spezifikatoren verwendbar und sollten entsprechend gekennzeichnet sein.

Vertragsmethode auf Analyse und Entwurf übertragen und dazu die Modellierungssprache BON (**B**usiness **O**bject **N**otation) entwickelt (s. 5.2) [63]. Jim McKim und Richard Mitchell haben wesentlich zur Ausarbeitung von SdV beigetragen [25], [26], [27], [41], [43], [44]. Weitere Quellen für Folgendes sind [14], [51], [69], [70], [71], [73].

# 3 Aspekte der Vertragsmethode

Um SdV an Beispielen zu erläutern, benutzen wir eine Notation, die sich semantisch an Eiffel und BON, syntaktisch an Oberon orientiert [55]. Diese Vertragssprache **Cleo** (**C**ontract Specification **L**anguage based on **E**iffel and **O**beron) dient in [13] dazu, die Vertragsmethode für Component Pascal zu erschließen, ist aber unabhängig von einer Implementationssprache. Cleo ist charakterisiert als

- formale Sprache, aber erweiterbar um informale Elemente;
- Modellierungssprache zur kompakten vertraglichen Spezifikation von Schnittstellen, die Implementationsdetails verbergen (keine Implementationssprache);
- deklarative Sprache mit seiteneffektfreien Ausdrücken (keine imperative Sprache mit zustandsverändernden Anweisungen);
- typisierte Sprache mit Grundtypen und der Fähigkeit, neue Typen zu definieren.

## 3.1 Invarianten, Vor- und Nachbedingungen

Das erste Beispiel, das die drei Grundelemente von Verträgen vorstellt, konstruiert natürliche Zahlen aus einem ganzzahligen Grundtyp:

```
INTERFACE Natural

  QUERIES
    N : INTEGER

  INVARIANTS
    N > 0

  ACTIONS
    Set (IN newN : INTEGER)
      PRE
        newN > 0
      POST
        N = newN

END Natural
```

Cleo trennt Abfragen und Aktionen durch die QUERIES- und ACTIONS-Abschnitte. Die Invariante N > 0 im INVARIANTS-Abschnitt schränkt den Wertebereich der Abfrage N ein. Das **Prinzip des gleichförmigen Zugriffs** (*uniform access*) verlangt, von der Implementation parameterloser Abfragen zu abstrahieren; N ist als Attribut oder parameterlose Funktion implementierbar, der Zugriff auf N wird davon unabhängig notiert.

Die Aktion Set soll bewirken, dass N nach einem Aufruf von Set den als Parameter newN übergebenen Wert liefert. Die PRE- und POST-Abschnitte für die Vor- und Nachbedingungen spezifizieren diese Semantik. Da Set die Invariante nicht verletzen darf, schränkt es durch die Vorbedingung den akzeptablen Wertebereich von newN ein.

## 3.2 Ergebnisse in Nachbedingungen

Als zweites Beispiel dient eine parametrisierte Abfrage. Die Fakultätsfunktion Factorial könnte Teil einer Komponente mit mathematischen Funktionen sein:

```
QUERIES
   Factorial (IN n : INTEGER) : INTEGER
      PRE
         n >= 0
      POST
         (n <= 1) IMPLIES (result = 1)
         (n > 1) IMPLIES (result = n * Factorial (n - 1))
```

Zum Formulieren der Nachbedingung brauchen wir einen Namen für das Ergebnis des Abfragenaufrufs. Cleo reserviert dafür den Standardnamen **result**. Die Nachbedingung besteht aus zwei Bedingungen, die konjunktiv zu verknüpfen sind; hier die beiden Implikationen, die der rekursiven Definition der Fakultät entsprechen. Wir spendieren für jede einzelne Bedingung eine Zeile und lassen Cleo die Bedingungen konjunktiv verknüpfen, ohne ein AND hinschreiben zu müssen.

## 3.3 Vorzustände in Nachbedingungen

Das dritte Beispiel modelliert eine mathematische Menge als generische Komponente Set; Element ist der generische Parameter, der als Elementtyp fungiert:

```
INTERFACE Set [Element]

   QUERIES
      Count : INTEGER
         -- Number of elements in the set.

      Has (IN x : Element) : BOOLEAN
         -- Does the set contain x?
         POST
            result IMPLIES (Count > 0)

      IsEmpty : BOOLEAN
         -- Does the set contain no element?

   INVARIANTS
      Count >= 0
      IsEmpty = (Count = 0)

   ACTIONS
      Put (IN x : Element)
         -- Include x into the set.
         POST
            Has (x)
            OLD (Has (x)) IMPLIES (Count = OLD (Count))
            NOT OLD (Has (x)) IMPLIES (Count = OLD (Count) + 1)

      Remove (IN x : Element)
         -- Exclude x from the set.
         POST
            NOT Has (x)
            OLD (Has (x)) IMPLIES (Count = OLD (Count) - 1)
            NOT OLD (Has (x)) IMPLIES (Count = OLD (Count))
```

```
        WipeOut
            -- Exclude all elements from the set.
            POST
                Count = 0
    END Set
```

Um das Verhalten der Aktionen Put und Remove, des Hinzufügens eines Elements zur Menge und des Entfernens eines Elements aus der Menge zu spezifizieren, ist der Zustand der Menge *vor* einem Aktionsaufruf mit ihrem Zustand *nach* dem Aktionsaufruf zu vergleichen. Das **Vorzustandskonstrukt** ermöglicht dies: Mit OLD (...) geklammerte Ausdrücke liefern den Wert des geklammerten Ausdrucks vor einem Dienstaufruf. OLD-Ausdrücke dürfen nur in Nachbedingungen auftreten. Der Ausdruck

```
Count = OLD (Count) + 1
```

bedeutet, dass sich der Wert von Count durch die Ausführung des Dienstes um 1 erhöht. Dabei muss es sich um eine Aktion handeln, denn da eine Abfrage q den Zustand ihrer Komponente unverändert lässt, gelten für sie Nachbedingungen der Art

```
other = OLD (other)
```

mit beliebigen von q verschiedenen Abfragen other. Nachbedingungen, die ausdrücken, was sich *nicht* ändert, lässt man meist weg.

## 3.4 Vollständige Spezifikation

Wir haben wesentliche Elemente von Verträgen kennen gelernt und wenden uns methodischen Aspekten zu. Im Beispiel der Menge von 3.3 ergänzt die informale Spezifikation durch Kommentare die formalen Verträge, denn diese sind

☹ schon zu komplex, um mit einem Blick erfassbar zu sein;

☹ trotzdem unvollständig.

Wir stellen fest, dass informale und formale Spezifikation sich nicht ausschließen, sondern ergänzen. Die Unvollständigkeit der Spezifikation von Put zeigt sich z.B. darin, dass eine Implementation ein Element y durch ein Element z ≠ y ersetzen könnte, ohne den Vertrag zu verletzen, der das Hinzufügen von x fordert. Unser Ziel ist jedoch, Verträge möglichst vollständig zu formulieren. Im Beispiel gelingt dies, indem wir Mengen - die Exemplare einer Mengenklasse - durch die Teilmengenrelation vergleichbar machen: Wir modellieren Set als Erweiterung des abstrakten Typs PartComparable, der die Relationen =, #, <, >, <=, >= bietet:

```
INTERFACE Set [Element] EXTENDS PartComparable

    QUERIES
        Count : INTEGER

        Has (IN x : Element) : BOOLEAN
            POST
                result IMPLIES (Count > 0)

        IsEmpty : BOOLEAN

        ... -- inherited services not shown here
```

```
  INVARIANTS
    Count >= 0
    IsEmpty = (Count = 0)

  ACTIONS
    Put (IN x : Element)
      POST
        Has (x)
        OLD (Has (x)) IMPLIES (this = OLD (this))
        NOT OLD (Has (x)) IMPLIES ((this > OLD (this)) AND (Count = OLD (Count) + 1))

    Remove (IN x : Element)
      POST
        NOT Has (x)
        OLD (Has (x)) IMPLIES ((this < OLD (this)) AND (Count = OLD (Count) - 1))
        NOT OLD (Has (x)) IMPLIES (this = OLD (this))

    WipeOut
      POST
        Count = 0

END Set
```

**this** ist der für das aktuelle Objekt verwendete Standardname. Der Ausdruck

```
this < OLD (this)
```

vergleicht den Wert der aktuellen Menge *nach* dem Dienstaufruf mit ihrem Wert *vor* dem Dienstaufruf, < bedeutet hier „ist echte Teilmenge von". So sind alle Dienste von Set außer Count und Has vollständig vertraglich spezifiziert.

Vollständige Spezifikation gelingt nur, wenn Abfragen alle Zustände der Komponente erfassen, d.h. alle durch Aktionen bewirkten Zustandsänderungen durch Abfragen beschreibbar sind. SdV setzt daher voraus, Abfragen und Aktionen sorgfältig aufeinander abzustimmen.

## 3.5 Änderungsverbote

Oft ist das, was sich *nicht* ändert, schwerer als Änderungen zu beschreiben. Ist vollständige Spezifikation gefordert, so kann ein nur in Nachbedingungen erlaubtes **Änderungsverbotskonstrukt** nützen:

```
SomeAction
  POST
    CHANGE_ONLY {a, b, c}
```

Hier darf SomeAction nur Änderungen der Abfragen a, b, c bewirken. Für eine Abfrage SomeQuery gilt, da sie seiteneffektfrei ist:

```
SomeQuery
  POST
    CHANGE_ONLY {}
```

## 3.6 Elementare und abgeleitete Abfragen

Abfragen dienen dazu, Abfragen und Aktionen zu spezifizieren. Um endlose Rekursionen in Verträgen zu vermeiden, empfiehlt es sich, zwischen elementaren und abgeleiteten Abfragen zu unterscheiden. **Elementare Abfragen** werden nicht vertraglich

spezifiziert, sondern als selbsterklärend vorausgesetzt und als **Spezifikatoren** (*specifier*), d.h. zur Spezifikation von **abgeleiteten Abfragen** und Aktionen eingesetzt.[3] Sind die Spezifikatoren verständlich, so sollten auch die damit formulierten Verträge verständlich sein. In den Beispielen der Menge von 3.3 und 3.4 sind Count und Has elementare Abfragen, wobei sich Has partiell mittels Count spezifizieren lässt. IsEmpty ist eine abgeleitete Abfrage, die vollständig durch Count spezifiziert ist. Put, Remove und WipeOut lassen sich mittels Count und Has spezifizieren.

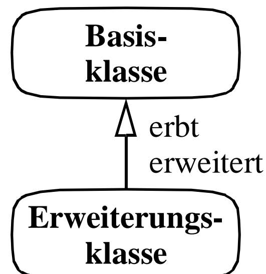
Das Trennen elementarer und abgeleiteter Abfragen ist eine von vielen Leitlinien, die helfen, SdV systematisch im Softwareentwicklungsprozess anzuwenden. Von McKim und Mitchell formulierte Prinzipien und Regeln haben zu folgendem Kasten angeregt [25], [26], [27], [41], [43], [44].

| **Leitlinien für Spezifikation durch Vertrag** | |
|---|---|
| Wende folgende Leitlinien iterativ an. | |
| (1) | Trenne Abfragen und Aktionen. |
| (2) | Trenne elementare und abgeleitete Abfragen. Spezifiziere elementare Abfragen nicht vertraglich, sondern nutze sie als Spezifikatoren. Spezifiziere abgeleitete Abfragen in Termen elementarer Abfragen. |
| (3) | Formuliere Invarianten als allgemeine, unveränderliche Konsistenzbedingungen zwischen Abfragen. |
| (4) | Formuliere für jeden Dienst (außer parameterlosen booleschen Abfragen) eine passende Vorbedingung mittels Abfragen und Parameter. |
| (5) | Formuliere für jede abgeleitete oder parametrisierte Abfrage eine Nachbedingung, die das Ergebnis der Abfrage mittels elementarer Abfragen und Parameter beschreibt. |
| (6) | Formuliere für jede Aktion eine Nachbedingung, die den Effekt der Aktion auf jede elementare Abfrage abhängig von Parametern beschreibt. Verzichte in Nachbedingungen von Aktionen darauf, das zu spezifizieren, was sich *nicht* ändert. |
| (7) | Untersuche bei Abfragen und Parametern Einschränkungen ihrer Wertebereiche (Intervalle bei Zahlen, nichtleere Referenzen) und formuliere diese als Invarianten, Vor- und Nachbedingungen. |
| (8) | Untersuche die Abfragen paarweise auf permanente Beziehungen und formuliere diese als Invarianten. |
| (9) | Sorge dafür, dass Abfragen in Vorbedingungen günstig zu berechnen sind. |
| (10) | Um bei Klassen Redefinition von Diensten zu ermöglichen, impliziere jede Nachbedingung aus der entsprechenden Vorbedingung. |

[3] „Keins von den Dingen definieren wollen, die in sich selbst so bekannt sind, dass man keine noch klareren Begriffe hat, um sie zu erklären“ (Blaise Pascal, 1623-1662).

## 3.7 Verträge und Vererbung

Ein Grundkonzept objektorientierter Softwarekonstruktion ist das Erben oder Erweitern von Klassen. Eine Erweiterungsklasse erbt von ihren Basisklassen Schnittstellen und Implementationen, also aus Sicht der Spezifikation




- alle Dienste,
- die Invarianten,
- die Vor- und Nachbedingungen aller Dienste.

Im Beispiel

```
INTERFACE SequenceOfPrimes EXTENDS Sequence
END SequenceOfPrimes
```

erbt SequenceOfPrimes alle Dienste und Verträge von Sequence. Sequence lässt sich als abstrakte Konzept- und Schnittstellenklasse realisieren, SequenceOfPrimes als konkrete Implementationsklasse. Abstrakte Basisklassen werden durch **abstrakte Verträge** spezifiziert. Erweiterungsklassen dürfen u.a. **geerbte Verträge** anpassen:

- geerbte Invarianten verstärken,
- Vorbedingungen geerbter Dienste abschwächen,
- Nachbedingungen geerbter Dienste verstärken,

also von Kunden weniger verlangen und ihnen mehr bieten als ihre Basisklassen. Dies entspricht der anschaulichen Einsetzungsregel (**Ersetzbarkeitsprinzip** von Liskov): Wo ein Objekt einer allgemeinen Klasse erwartet wird, ist ein Objekt einer speziellen Klasse einsetzbar, da es alle geforderten allgemeinen Dienste besitzt und Verträge einhält [22].

## 3.8 Quantoren

Die Aussagenlogik kann nicht alle Bedingungen ausdrücken. Gelegentlich ist die Prädikatenlogik erster Stufe nützlich, z.B. bei Assoziationen. Daher ist es sinnvoll, die Spezifikationssprache um Elemente der Mengenlehre, Quantoren und prädikatenlogische Ausdrücke zu erweitern.

```
INTERFACE SortableList [Element EXTENDS Comparable]

  QUERIES
    Count : INTEGER
      -- Number of elements in the list.
```

```
      Item (IN i : INTEGER) : Element
         -- Element at position i in the list.
         PRE
            (1 <= i) AND (i <= Count)
         POST
            -- Number (result) > 0
      Number (IN x : Element) : INTEGER
         -- How many times does the list contain x?
         POST
            (0 <= result) AND (result <= Count)
            (result > 0) IMPLIES (EXISTS i IN {1 .. Count} IT_HOLDS x = Item (i))
      Sorted : BOOLEAN
         -- Are the elements in the list totally ordered?
         POST
            result = FOR_ALL i, k IN {1 .. Count} IT_HOLDS
                                             (i <= k) IMPLIES (Item (i) <= Item (k))
      Permutes (IN other : LIKE this) : BOOLEAN
         -- Are the elements in the list a permutation of the elements in the other list?
         POST
            Count = other.Count
            result = FOR_ALL i IN {1 .. Count} IT_HOLDS
                                             Number (Item (i)) = other.Number (Item (i))
   INVARIANTS
      Count >= 0
      Permutes (this)
   ACTIONS
      Sort
         POST
            Sorted
            Permutes (OLD (this))
END SortableList
```

Das Beispiel nutzt den **Allquantor** FOR_ALL, den **Existenzquantor** EXISTS und die **Elementrelation** IN. Die Abfrage Number ist unvollständig spezifiziert; die mit Quantoren mögliche vollständige Spezifikation sei dem Leser als Übung überlassen.

Zudem zeigt das Beispiel **eingeschränkte Generizität**: Der generische Parameter Element muss den abstrakten Typ Comparable erweitern, damit Listen sortierbar sind. Der Parameter other von Permutes ist mit einem **verankerten Typ** vereinbart: LIKE this ist der Typ des aktuellen Objekts (der wegen Erweiterung und Objekterzeugung nicht exakt mit SortableList [Element] beschrieben ist).

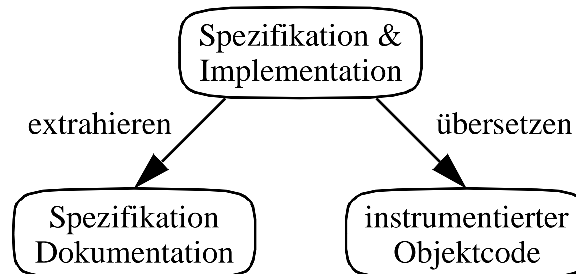
## 3.9 Von der Spezifikation zur Implementation

Spezifikationen gewinnen Wert durch passende Implementationen. Eine Implementation soll gegen ihre Spezifikation prüfbar sein. Zwei Ansätze führen von den Ebenen Analyse, Entwurf und Spezifikation zu den Ebenen Implementierung, Test und Betrieb:

(1) **Eine Sprache**: Die Programmiersprache unterstützt SdV und dient der Spezifikation *und* der Implementation. Diesen Ansatz realisiert z.B. Eiffel. Eine Spezifikation besteht aus Quelltext, der Schnittstellen mit Signaturen und Verträgen

beschreibt. Eine Implementation ergänzt diesen Quelltext, indem sie Abfragen als Attribute oder Funktionen definiert und Aktionen mit Algorithmen versieht. Aus Sicht der Implementation sind Verträge Zusicherungen, die zur Laufzeit prüfbar sind. Die Laufzeitprüfungen sind flexibel an- und abschaltbar. Die Spezifikation wird so zu einem eingebauten Selbsttest für die Implementation:

**Design by Contract = Specs'n'Checks**




Dieser Ansatz folgt den **Prinzipien**

☺ **der Selbstdokumentation**, wonach gute Software sich selbst beschreibt;

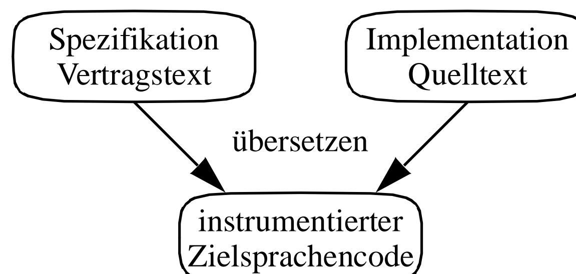
☺ **des einzigen Quelldokuments**, wonach es pro Komponente nur *ein* manuell manipuliertes Dokument gibt, das Dokumentation von Analyse- und Entwurfskonzepten, Spezifikation und Implementation integriert; und

☺ **der nahtlosen, reversiblen, werkzeuggestützten Entwicklung** (s. 4).

Voller Nutzen aus SdV ist nur mit Ansatz (1) zu erzielen. Da verbreitete Implementationssprachen dem nicht entsprechen, betrachten wir auch Ansatz (2):

(2) **Zwei Sprachen**: Eine Spezifikationssprache, z.B. hier Cleo, dient der Formulierung der Schnittstelle mit Verträgen, eine Implementationssprache dient der Formulierung eines ausführbaren Programms mit Daten und Algorithmen, das die getrennt vorliegende Spezifikation erfüllt. Gewählte Implementationssprache, Entwicklungsumgebung und Werkzeuge können SdV mehr oder weniger gut unterstützen. Drei Varianten dazu sind:

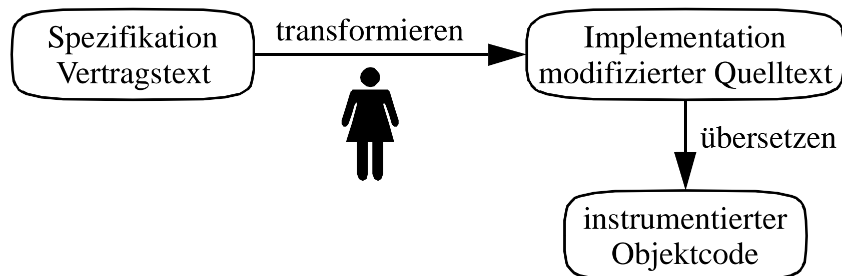
(2.1) Automatisches Übersetzen der Spezifikation und der Implementation in instrumentierten Zielsprachencode. Die Zielsprache kann u.a. mit der Implementationssprache übereinstimmen, eine Zwischensprache wie Java-Bytecode oder eine Maschinensprache sein.




☹ Zwei Quelldokumente widersprechen dem Prinzip des einzigen, selbstdokumentierenden Quelldokuments.

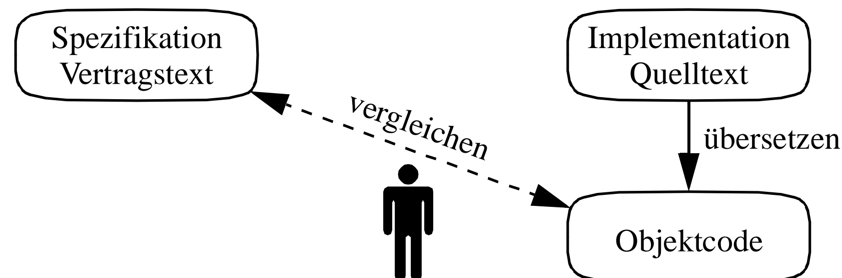
☹ Zwischen Spezifikations- und Implementationssprache zu wechseln widerspricht nahtloser Entwicklung.

(2.2) Manuelles Übertragen der Spezifikation in die Implementation, automatisches Übersetzen des modifizierten Quelltexts in ausführbaren Objektcode.




☹ Spezifikationstext in zwei Dokumenten manuell konsistent zu halten behindert reversible Entwicklung.

(2.3) Getrennthalten von Spezifikation und Implementation.




☹ Spezifikation und Implementation manuell zu vergleichen ignoriert Möglichkeiten werkzeuggestützter Entwicklung.

# 4 Die Vertragsmethode im Softwareentwicklungsprozess

Ausgehend von Ansatz 3.9 (1) diskutieren wir, wie sich SdV auf die Softwareentwicklung und ihre Ebenen sowie auf das Softwareprodukt und seine Qualitätsmerkmale auswirkt. Die Vertragsmethode unterstützt die Softwarekonstruktion als ganzheitlichen, stetigen, iterativen Prozess [63]. Dabei ist die Entwicklung

- **nahtlos** (*seamless*), da eine einheitliche Spezifikations- und Implementationssprache Brüche zwischen Spezifikation und Implementierung vermeidet und Verträge direkt in Implementationen abbilden kann;
- **reversibel**, da der Entwickler die Ebenen beliebig wechseln kann - z.B. von der Implementierung zurück zum Entwurf und zur Analyse - ohne die Konsistenz der Sichten der verschiedenen Ebenen zu gefährden.

Erreicht wird dies durch Einhalten des Prinzips des einzigen, selbstdokumentierenden Quelldokuments. Die Vertragsmethode ist darauf ausgerichtet, höherwertige Software zu liefern bezüglich

- **funktionaler Qualitätsmerkmale** wie Zuverlässigkeit, insbesondere Korrektheit und Vertrauenswürdigkeit, sowie Benutzbarkeit und Verständlichkeit [15], [38];

- **struktureller Qualitätsmerkmale** wie Wartbarkeit, Änderbarkeit und Wiederverwendbarkeit [30], [39].

## 4.1 Analyse

Wie andere Methoden hat man die Vertragsmethode von der Programmierung auf den Entwurf und die Analyse übertragen. Folgende Schritte umreißen die **Analysemethode**:

(1) Finde Komponenten des Anwendungsbereichs und Beziehungen zwischen diesen Komponenten.

(2) Definiere die Schnittstellen dieser Komponenten und beschreibe ihre dynamische Semantik durch Verträge.

Die Vertragsmethode formalisiert semantische Eigenschaften einer Anwendung frühzeitig in der Analyse auf hohem Abstraktionsniveau, ist aber allgemeinverständlich und daher kundenfreundlich. Entwickler können z.B. Geschäftsregeln von eBusiness-Anwendungen als Invarianten formulieren und diese mit Anwendern besprechen.

Das Trennen von Abfragen und Aktionen ersetzt das von datenzentrierten Ansätzen empfohlene Trennen von Daten und Operationen. Die Vertragsmethode abstrahiert von der Implementation und vermeidet den Nachteil, Implementationsentscheidungen in der Analyse vorwegzunehmen.

## 4.2 Entwurf

Als Entwurfsmethode ist SdV umrissen durch die Schritte 4.1 (1) und (2) mit „Lösungsbereich“ statt „Anwendungsbereich“. Sehr nützlich ist die Vertragsmethode bei **Entwurfsmustern**, deren Bedeutung Gamma, Helm, Johnson und Vlissides mit ihrem Katalog von Musterlösungen zu wiederkehrenden Fragen des objektorientierten Softwareentwurfs herausstellen [12]. Jézéquel, Train und Mingins präzisieren diese Entwurfsmuster durch abstrakte Verträge in abstrakten Musterklassen [16]. Entwurfskonzepte lassen sich mit Verträgen direkt in Code integrieren bzw. dort dokumentieren. Entwurfsmuster gewinnen so durch Verträge an Exaktheit, Verständlichkeit und Wiederverwendbarkeit.

Mit einer ganzheitlichen Sicht auf den Prozess und das Produkt entspricht die Vertragsmethode dem **Prinzip der konstruktiven Voraussicht**: Sie antizipiert den späteren Test im früheren Entwurf durch den „Entwurf für Testbarkeit“ (*design for testability*). Vertraglich spezifizierte Komponenten sind besser testbar, werden so zuverlässiger und vertrauenswürdiger, und damit wiederum leichter, angstfreier wiederverwendbar.

Vererbte Verträge garantieren die Konsistenz von Klassenhierarchien, d.h. Objekte von Erweiterungsklassen verhalten sich gemäß den Verträgen ihrer Basisklassen. Konsistente Klassenstrukturen bleiben leichter länger stabil. SdV hilft Entwurfsfehler vermeiden, die z.B. durch undisziplinierte Vererbung zu instabilen Klassenhierarchien führen.

## 4.3 Spezifikation und Dokumentation

Die semantische Schnittstellenspezifikation durch Verträge ist formal, exakt, eindeutig und z.T. vollständig möglich. SdV reduziert Mehrdeutigkeiten, die bei informaler Spezifikation kaum vermeidbar sind. Verträge sind sowohl wesentlicher Teil der Dokumenta-

tion als auch prüfbarer Teil der Implementation. Annahmen des Codes wie Voraussetzungen und Wirkungen von Diensten sind explizit, klar dokumentiert.

SdV integriert verschiedene Sichten der Software - Spezifikation und Implementation, Dokumentation und Code - in ein Quelldokument und vermeidet so, dass sie divergieren. Spezifikation und Dokumentation sind stets konsistent mit der Implementation, aktuell und vertrauenswürdig. Die Spezifikation veraltet nicht und geht nicht verloren, wenn die Implementation weiter entwickelt wird, sondern wird mit dieser aktualisiert. Dokumentieren ist in den Entwicklungsprozess integriert, keine lästige Nachbeschäftigung.

Es ist jederzeit möglich, Dokumentationen und Spezifikationen für Anwender aus Quelldokumenten automatisch zu extrahieren. Die Vertragsmethode reduziert die Anzahl manuell zu bearbeitender Dokumente, nutzt die Möglichkeiten von Werkzeugen, vereinfacht so das Entwickeln und reduziert die Dokumentationskosten.

## 4.4 Implementierung

Da Spezifikation und Implementation in demselben Quelldokument stehen, hat ein Entwickler die Spezifikation stets vor Augen, wenn er implementiert, und schreibt daher leichter eine korrekte Implementation. Verträge sind zur Laufzeit prüfbar, eingebaute Selbsttests zum Prüfen der Korrektheit der Implementation bzgl. der Spezifikation, sodass ein Entwickler Implementierungsfehler früher und leichter findet. Die Laufzeitprüfungen erhöhen die Vertrauenswürdigkeit der Implementation *und* der Spezifikation.

Ein Implementierer kann eine gegebene Spezifikation ergänzen: Hat er geeignete Datenstrukturen zur Implementation gewählt, so kann er zusätzliche Konsistenzbedingungen an diese Datenstrukturen als **implementationsbedingte Invarianten** und **Nachbedingungen** formulieren und damit die Korrektheit der Algorithmen bzgl. der Daten prüfen.

Alle Dienste einer Schnittstelle teilen sich in Abfragen und Aktionen, Abfragen sind als Attribute oder seiteneffektfreie Funktionen implementierbar, Aktionen als Prozeduren. Dieses Prinzip führt Entwickler dazu, ohne seiteneffektbehaftete Funktionen zu implementieren. Fehlerträchtiges Programmieren mit Seiteneffekten wird zurückgedrängt.

## 4.5 Qualitätssicherung und Test

Die Vertragsmethode ist für alle Maßnahmen zur Qualitätssicherung vorteilhaft, denn diese sind ohne Spezifikationen sinnlos. Technische Reviews werden effektiver, wenn man informale und formale Verträge mit Implementationen vergleichen kann. Da sich formale Verträge durch statische Analysewerkzeuge prüfen lassen, z.B. einen Übersetzer, sind sie informalen Spezifikationen überlegen.

Tester kennen mit den Verträgen einer Komponente ihr Sollverhalten genau und wissen, worauf sie beim Testen achten müssen. Verträge als eingebaute Prüfungen liefern Tests mit hohen Fehlererkennungsraten. Fehlverhalten ist nahe bei der Fehlerursache zu beobachten: Der Grund einer verletzten Zusicherung liegt oft nur einige Anweisungen zurück. Das verkürzt die Zeit zum Suchen und Beheben von Fehlern.

SdV ermöglicht effektive Verfahren für Black-Box-Tests, die auf Verträgen basieren. Ein schematisches Testverfahren, das aus einem Entwurfsmuster, einem wiederverwendbaren Testwerkzeug, randomisierten Einzeltests und Dauertests im Vorder- oder Hinter-

grund besteht, stellt [13] ausführlich vor. Dabei sind nur Einzeltests zu programmieren. Da dieses Verfahren keine Testausgabe produziert, vermeidet es aufwändiges manuelles Prüfen umfangreicher Testprotokolle und erlaubt einfach und schnell durchzuführende Regressionstests. Ein Ansatz für ein automatisiertes Testverfahren ist, mittels eines Werkzeugs aus den Verträgen eines Prüflings einen Testtreiber zu generieren.

Insgesamt reduziert SdV den Testaufwand erheblich. Je weniger Testzeit man braucht, umso genauer lässt sie sich abschätzen. Freilich erfordert das Spezifizieren mehr Aufwand, doch spart man bei Implementierung, Test, Wartung und Pflege weitaus mehr ein.

### 4.6 Betrieb, Wartung und Pflege

Der Grad der Laufzeitprüfungen ist nach Komponenten und Zusicherungsarten flexibel einstellbar und liefert so „best of both worlds“: Nach Bedarf kann sich ein Anwender entscheiden für

- mehr Laufzeitprüfungen und damit mehr Vertrauenswürdigkeit und Sicherheit durch demonstrierbare Korrektheit, oder
- weniger Laufzeitprüfungen und damit höhere Effizienz.[4]

Die Vertragsmethode unterstützt die Wartbarkeit und Erweiterbarkeit der Software durch die integrierte konsistente Dokumentation. Entwickler sehen die Spezifikationen beim Warten, Pflegen, Ändern und Erweitern von Komponenten und bauen daher weniger leicht neue Fehler ein. Neue Versionen und andere Varianten einer Komponente lassen sich besser entwickeln. Der Wartungsaufwand reduziert sich.

## 5 Sprachen, Werkzeuge und Projekte

Wie unterstützen verbreitete Spezifikations- und Implementationssprachen, Notationen und Werkzeuge die Vertragsmethode? Wir geben einen Überblick und zeigen Beispiele.

### 5.1 Eiffel

Eiffel bedeutet SdV pur. Eiffel ist nicht nur eine objektorientierte Programmiersprache, sondern eine Methode, unterstützt durch eine elegante Notation [28], [30], [31], [32]. Die Vertragsmethode nach Ansatz 3.9 (1) bildet den Kern des Entwurfs von Eiffel. Neben Klasseninvarianten, Vor- und Nachbedingungen, Vorzuständen, Änderungsverboten, Vererben und Anpassen von Verträgen, Schleifenvarianten und -invarianten, allgemeinen Zusicherungen, gleichförmigem Zugriff auf Attribute und Funktionen bietet Eiffel u.a.:

- disziplinierte Ausnahmebehandlung, die gescheiterte Aufrufe wiederholen kann;
- spezielle Kommentare bei Zusicherungen, die bei Fehlern als Laufzeitinformationen bereitstehen;
- flexible Kontrolle darüber, welche Klassen welche Arten von Zusicherungen zur Laufzeit prüfen sollen;

[4] Man sollte den Aufwand für Laufzeitprüfungen nicht überschätzen. Ein guter Entwurf kann mehr zu optimaler Leistung beitragen als das Abschalten von Prüfungen, das einige Prozent Geschwindigkeitszuwachs auf Kosten von Sicherheit bringt.

- automatisches Verhindern des Prüfens geschachtelter Zusicherungen beim Prüfen von Zusicherungen (um endlose Rekursionen zu vermeiden).

Ein exemplarisches Eiffel-Fragment einer generischen Menge (vgl. 3.3) zeigt wesentliche Elemente:

```
class SET [ELEMENT]

feature -- queries
    count : INTEGER
            -- Number of elements in the set

    has (x : ELEMENT) : BOOLEAN is
            -- Does the set contain x?
        require
            x_exists: x /= Void
        do
            ...
        ensure
            not_found_if_empty: Result implies count > 0
        end

feature -- commands
    put (x : ELEMENT) is
            -- Include x into the set
        require
            x_exists: x /= Void
        do
            ...
        ensure
            x_included:   has (x)
            had_already:  old has (x) implies count = old count
            had_not_yet:  not old has (x) implies count = old count + 1
        end

    ...

invariant
    count >= 0
end -- class SET
```

Neue Eiffel-Versionen sind um **Agenten** erweitert, die u.a. Quantoren realisieren können [34]. Ausdrücke mit einem Quantor sind gut lesbar; das Beispiel von 3.8, bei dem zwei Inline-Agenten zwei Allquantoren implementieren, liest sich wegen seiner Länge weniger leicht:

```
sorted : BOOLEAN is
        -- Are the elements in the list totally ordered?
    ensure
        Result = (1 |..| count).for_all (i : INTEGER |
                        (1 |..| count).for_all (k : INTEGER | i <= k implies item (i) <= item (k)))
```

Eiffel bietet alle Vorteile des Ansatzes 3.9 (1) für den Softwareentwicklungsprozess. Spezifikations- und Implementationstext sind in einem Dokument integriert; verschiedene Sichten darauf wie Spezifikation aus Signaturen, Verträgen und Kommentaren, Benutzungs- und Vererbungsstrukturen, oder Klassendiagramme sind extrahierbar. Indem Eiffel mit BON kooperiert - unterstützt durch das CASE-Werkzeug EiffelCase - ermöglicht es nahtloses, reversibles, werkzeuggestütztes Entwickeln.

## 5.2 Business Object Notation

Die Business Object Notation (BON) kombiniert eine objektorientierte Analyse- und Entwurfsmethode mit einer Modellierungssprache. Nerson und Waldén haben BON von 1989 bis 1994 entwickelt und dokumentiert [63]. BON konzentriert sich darauf, Klassenarchitekturen zu modellieren und das Verhalten der Klassen zu spezifizieren. Dabei ist die Vertragsmethode zentral, BON erweitert sie zu einer Analysemethode. In Kooperation mit Eiffel entspricht BON Ansatz 3.9 (1) und ermöglicht einen nahtlosen, reversiblen Entwicklungsprozess von der Analyse über die Implementierung bis zum Betrieb. Werkzeugunterstützung erfährt BON durch EiffelCase.

Während BON als Methode einen Prozess und einzelne Aktionen definiert, umfasst es als Modellierungssprache verschiedene Diagrammarten und eine integrierte Vertragssprache in zwei Varianten:

- Die **grafische Vertragssprache** dient dazu, in Klassendiagrammen dargestellte Klassen und Beziehungen zwischen Klassen kompakt zu spezifizieren.
- Die **textuelle Vertragssprache** dient dazu, die Lücke zwischen Analyse/Spezifikation und Implementation zu schließen.

Beide Varianten der Vertragssprache sind äquivalent, sie liefern nur verschiedene Sichten auf Klassen. Man kann zwischen der grafischen und der textuellen Darstellung einer Klasse wechseln - in beide Richtungen, wie es ein reversibler Prozess fordert. Eine Klasse, ihr Vertrag und ihr Diagramm sind nie getrennt - wie es ein nahtloser Prozess fordert.

Die textuelle Vertragssprache ähnelt der Eiffel-Vertragssprache und bietet alle von Eiffel bekannten Vertragskonstrukte. Darüber hinaus integriert BON Elemente der Mengenlehre und der Prädikatenlogik erster Stufe in einer Form, die der üblichen mathematischen Notation nahekommt und ihrer Semantik entspricht. Das Beispielfragment von 3.8 und 5.1 lässt sich in grafischer BON-Vertragssprache elegant so formulieren:

*sorted* : *BOOLEAN*
$\boxed{!}$ $Result = \forall\ i, k \in \{1\ ..\ count\} \bullet (i \leq k \rightarrow item\ (i) \leq item\ (k))$

BON kann so alle Arten von Beziehungen zwischen Klassen - Assoziation, Komposition, Aggregation - einschließlich Kardinalitäten ausdrücken, ohne dazu weitere Modellierungskonstrukte zu benötigen.

## 5.3 Object Constraint Language

Die Object Constraint Language (OCL) ist die Vertragssprache der Unified Modeling Language (UML) in der von der Object Management Group (OMG) 1997 standardisierten Version 1.1 [47], [48], [64], [65]. Jos Warmer hat OCL bei IBM entwickelt; als Beitrag von IBM und ObjecTime kam OCL zu UML hinzu, nachdem Booch, Jacobson und Rumbaugh ihre Modellierungssprachen zu UML vereinigt hatten. Darin drückt sich einerseits wachsendes Interesse an SdV aus, andererseits wurde mit UML eine komplexe, z.T. redundante Sammlung von Modellierungskonstrukten standardisiert, bevor man sich auf das Ausarbeiten von Methoden und Prozessen konzentrierte. Da OCL keine Implementationssprache ist, entspricht es Ansatz 3.9 (2). Nahtloses Entwickeln ist mit OCL nicht möglich, doch gibt es zu UML viele Werkzeuge.

OCL entspricht ungefähr der BON-Vertragssprache, es kennt u.a. Invarianten, Vor- und Nachbedingungen, Vorzustände und eine eingeschränkte Form von Quantoren. Als Beispiel spezifizieren wir eine Menge mit OCL:

```
XSet::count : Integer

XSet::has (x : Element) : Boolean
  pre:    -- none
  post:   result implies count > 0

XSet::put (x : Element)
  pre:    -- none
  post:   has (x)
  post:   count = count@pre or count = count@pre + 1

XSet
  count >= 0
```

Verglichen mit den Spezifikationen mit Cleo (s. 3.3) und Eiffel (s. 5.1) verzichten wir auf

☹ `Element` als generischen Parameter von `XSet`, da OCL zwar einige generische Behältertypen vordefiniert (darunter `Set`), aber sonst Generizität ignoriert;

☹ eine leere Referenz (Void, NIL, NULL), da OCL sie nicht erwähnt;

☹ den alten Wert von `has (x)`, da OCL nur Vorzustände von Attributen klar definiert, nicht von allgemeinen Ausdrücken.

Ein Ziel von OCL ist, eine formale Sprache zu sein, die für Menschen ohne mathematischen Hintergrund verständlich und benutzbar ist. Deshalb ähnelt die Syntax von OCL mehr der einer Programmiersprache als der mathematischen Notation. Formulieren wir das Beispiel mit den Quantoren (vgl. 3.8, 5.1 und 5.2) mit OCL:

```
SortableList::sorted () : Boolean
  post:   result = Integer.allInstances->forAll (i, k |
            1 <= i and i <= k and k <= count implies
              item (i) <= item (k))
```

Die eher schwerfällig wirkende Form lässt offen, was leichter lernbar ist: die Prädikatenlogik oder OCL? Einen detaillierten Vergleich von BON und UML/OCL liefern Paige und Ostroff; sie untersuchen, wie BON und OCL SdV unterstützen und diskutieren ihre Vor- und Nachteile [49]. Unterschiede zwischen BON und OCL sind u.a.:

- BON integriert Modellierungskonstrukte, Vertragssprache, Methode und Prozess. OCL ist ein Zusatz zu den Modellierungskonstrukten von UML, zu denen Methode und Prozess nachträglich entwickelt werden.
- BON kapselt Klasse, Vertrag und Diagramm und sorgt für ihre Konsistenz. In UML kann man den Vertrag einer Klasse alternativ
  - als einzelne Bemerkungen an das Klassendiagramm heften oder
  - als OCL-Text getrennt vom Klassendiagramm formulieren.

  Die erste Möglichkeit überfrachtet das Diagramm, die zweite stellt das Konsistenzproblem für die beiden Darstellungen Diagramm und Text.
- BON folgt dem Prinzip des gleichförmigen Zugriffs. OCL verwendet für Zugriffe „`.`“ und „`->`“, bei parameterlosen Funktionen ist „`()`“ anzufügen.

- BON vererbt Verträge auf Erweiterungsklassen, wo sie anpassbar sind. OCL empfiehlt das Vererben von Verträgen, schreibt es aber nicht vor, sodass ihre Anpassbarkeit offen bleibt.

Die folgende Tabelle gibt einen Überblick über Merkmale von BON und OCL, wobei unterstützte Merkmale durch zugehörige Schlüsselwörter angezeigt sind. Hinweise, Analyse durch Vertrag mit OCL anzuwenden, gibt [42].

| | **Vertragssprache** | | |
|---|---|---|---|
| **Merkmal** | **BON grafisch** | **BON textuell** | **OCL** |
| Invariante | **Invariant** | **invariant** | class constraint |
| Vorbedingung | ? | **require** | `pre` |
| Nachbedingung | ! | **ensure** | `post` |
| Ergebnis in Nachbedingung | *Result* | | `result` |
| Vorzustand in Nachbedingung | **old** expression | | attribute`@pre` association`@pre` |
| aktuelles Objekt | @ | *Current* | `self` |
| Verträge vererbbar? | ☺ | | ☹ empfohlen, nicht zwingend |
| Verträge anpassbar? | ☺ | | ☹ |
| verwendete Logik | zweiwertig | | dreiwertig |
| Menge, Elemente, Intervall | $\{a, b, c .. d\}$ | | ☹ |
| Elementrelation | ∈ | **member_of** | ☹ |
| Existenzquantor | ∃ | **exists** | collection `->exists` |
| Allquantor | ∀ | **for_all** | collection `->forAll` |
| sodass | \| | **such_that** | |
| gilt | ● | **it_holds** | \| |
| Klasse, Vertrag, Diagramm | ☺ integriert | | ☹ separiert |
| Entwicklung nahtlos & reversibel? | ☺ | | ☹ |
| Werkzeuge | ☺ EiffelCase | | ☺ viele |

## 5.4 Interface Definition Language

Die Interface Definition Language (IDL) in ihren Varianten OMG CORBA und Microsoft COM erlaubt, Syntax und statische Semantik von Diensten durch typisierte Signaturen zu beschreiben; sie unterstützt SdV also nicht. Zudem dient IDL eher als Ziel- denn als Quellsprache für Übersetzer. Neue Entwicklungen wie .NET verzichten auf Schnittstellendefinitionssprachen: Nicht Entwickler beschreiben Schnittstellen mit IDL, sondern Übersetzer erzeugen aus Quelltexten beliebiger Programmiersprachen Schnittstellenbeschreibungen in Form sprachübergreifender Metadaten [36].

## 5.5 Component Pascal

Die Sprache Component Pascal und das Werkzeug BlackBox Component Builder bilden eine der ersten komponentenorientierten Entwicklungsumgebungen; sie sind kommerzielle Weiterentwicklungen der Sprache und des Systems Oberon [55]. Component Pascal bietet allgemeine Zusicherungen, sodass wir es unter Ansatz 3.9 (2.2) einordnen. Wie man die Vertragsmethode mit Component Pascal anwenden kann, stellt [13] ausführlich dar. Als Beispiel dient hier eine polymorphe Menge (vgl. 3.3, 5.1 und 5.3):

```
MODULE ContainersSetsOfComparable;

  IMPORT BEC := BasisErrorConstants, BG := BasisGenerals;

  TYPE
    Element* = BG.Comparable;
    Set*     = POINTER TO SetDesc;
    SetDesc* = LIMITED RECORD (BG.PartComparable) ... END;

  (* Queries *)

  PROCEDURE (IN set : SetDesc) Count* () : INTEGER, NEW;
    (*! Number of elements in set. !*)
  BEGIN
    RETURN ...;
  END Count;

  PROCEDURE (IN set : SetDesc) Has* (x : Element) : BOOLEAN, NEW;
    (*! Does set contain x? !*)
    VAR result : BOOLEAN;
  BEGIN
    ASSERT (x # NIL, BEC.precondPar1NotNil);
    result := ...;
    ASSERT (~(result & set.Count () = 0), BEC.postcondResultOk);
    RETURN result;
  END Has;

  (* Invariants *)

  PROCEDURE (IN set : SetDesc) CheckInvariants, NEW;
  BEGIN
    (* Implementation invariants here. *)
    ASSERT (set.Count () >= 0, BEC.invariantClass);
  END CheckInvariants;

  (* Actions *)

  PROCEDURE (VAR set : SetDesc) Put* (x : Element), NEW;
    (*! Include x into set. !*)
    VAR oldCount : INTEGER; oldHas : BOOLEAN;
  BEGIN
    ASSERT (x # NIL, BEC.precondPar1NotNil);
    oldCount := set.Count (); oldHas := set.Has (x);
    ...
    set.CheckInvariants;
    ASSERT (set.Has (x), BEC.postcondSupplierOk);
    ASSERT (oldHas = (set.Count () = oldCount), BEC.postcondSupplierOk);
    ASSERT (~oldHas = (set.Count () = oldCount + 1), BEC.postcondSupplierOk);
  END Put;

END ContainersSetsOfComparable.
```

Zusicherungen sind in Component Pascal Aufrufe der Standardprozedur `ASSERT`. Sie werden immer geprüft (d.h. sind nicht zur Übersetzungszeit abschaltbar). BlackBox liefert bei verletzten Zusicherungen als Laufzeitinformation den Aufrufkeller mit den lokalen Daten sowie die globalen Daten. Vor- und Nachbedingungen sind direkt mit Zusicherungen formulierbar; Invarianten erfordern eine zusätzliche Prozedur. Ob eine Zusicherung eine Invariante, Vor- oder Nachbedingung darstellt, zeigt ihr zweiter Parameter an, eine symbolische Fehlerkonstante. Obwohl solche Zusicherungen ein einfaches Sprachkonstrukt sind, bieten sie eine effektive Testhilfe. Dieses Minimum an Unterstützung für SdV sollte keine Programmiersprache unterbieten.

BlackBox besteht aus einer Bibliothek von Grundkomponenten und dem BlackBox Component Framework, einem Gerüst wiederverwendbarer Schnittstellen und Standardkomponenten. Die Dokumentation nutzt Verträge, die Implementation prüft Verträge mit Zusicherungen. Entwickler von Kundenkomponenten finden so schnell Fehler, z.B. verletzte Vorbedingungen bei Aufrufen von Standardkomponenten, die ohne diese Technik mühsam aufzuspüren wären. Die Grenzen für SdV in Component Pascal/BlackBox zieht Ansatz 3.9 (2.2):

☹ Nahtloses, reversibles Entwickeln ist schwer. Dokumentation und Implementation stehen in unabhängigen Dokumenten, die konsistent zu halten aufwändig ist.

☹ Zusicherungen verbergen sich in Anweisungsteilen von Implementationen, werden in Schnittstellen nicht sichtbar und nicht automatisch als Verträge dokumentiert.

☹ Da Vorzustände und vererbbare, anpassbare Verträge nicht unterstützt werden, muss der Entwickler diese Merkmale mit Aufwand und Disziplin selbst programmieren.

☹ Öffentliche, schreibbare Variable und Attribute sind zugelassen, aber schwer mit Invarianten zu vereinbaren.

## 5.6 C und C++

C ermöglicht mit dem Zusicherungsmakro `assert()` der C-Standardbibliothek eine primitive Form von SdV nach Ansatz 3.9 (2.2), denn neben den in 5.5 genannten Nachteilen allgemeiner Zusicherungen liefert `assert()` keine Laufzeitinformation und zeigt die Rolle der Zusicherung (Invariante, Vor- oder Nachbedingung) nicht an. Es gibt jedoch Bibliotheken für C, die u.a. SdV unterstützen, z.B. die General Exception-Handling Facility (GEF), eine Sammlung von Makros [2], und BetterC, das aus Makros und Funktionen besteht [45].

C++ bietet für SdV nicht mehr als C, aber Stroustrup gibt Hinweise, wie sich Zusicherungen mit Templates und Ausnahmebehandlung implementieren lassen [59] S. 748ff. Der Aufgabe, C++-Programmierer an die Vertragsmethode heranzuführen, stellen sich [6], [11], [17], [40]. Darüber hinaus existieren zahlreiche Ansätze, C++ SdV-fähig zu machen; sie lassen sich so einteilen:

- **Vertragskonstrukte mit vorhandenen Sprachkonstrukten** realisieren:
  - Makros: GNU Nana ist eine freie Bibliothek von C-Makros, die Zusicherungen und SdV mit Vorzuständen, Quantoren u.a. bietet [12], [23].
  - Ausnahmebehandlung [68].

  - Vererben abstrakter Methoden für Verträge, Templates und Namenräume [24].
  - Percolation-Entwurfsmuster [3].
- **Spracherweiterung mit Präprozessor**, der um Vertragsausdrücke erweiterten Programmtext in C++-Text transformiert [56].
- **Separate Spezifikation**: Larch/C++ ist eine von C++ unabhängige Schnittstellenspezifikationssprache, die Quantoren umfasst und formale, vertragliche Spezifikationen von C++-Programmen gemäß Ansatz 3.9 (2.1) ermöglicht [4]. Ansatz [52] ist in ein Prototyping-System integriert; ein vertraglich erweitertes Python dient als Spezifikationssprache; zu jeder Python-Klasse wird eine C++-Klasse und eine Vertragsklasse erzeugt; verwendete Sprachkonstrukte sind Makros, mehrfaches Erben, Templates.
- **Spracherweiterung**: Annotated C++ (A++) ist ein zu einer Spezifikationssprache erweitertes C++, das Quantoren bietet und von einem CASE-Werkzeug unterstützt wird [5]. Extended xlC++ (exlC++) orientiert seine Vertragskonstrukte an Eiffel [53]. Eine minimale Spracherweiterung bietet DigitalMars [69].

Die Ansätze nutzen die breite Palette der Sprachkonstrukte von C++ (neben den genannten auch Zeiger und explizite Typanpassung), um SdV in C++ zu ermöglichen. Jeder Ansatz erschließt C++ gewisse Vertragselemente, hat aber auch Grenzen und Nachteile. Verträge voll unterstützt wie durch Eiffel in C++ zu realisieren ist deshalb so schwierig, weil einige Sprachkonzepte von C++ dem entgegenstehen.

Das wohl größte technische Problem für SdV in C, C++ und verwandten Sprachen wie Java und C# ist deren Orientierung auf seiteneffektbehaftete Operatoren, Funktionen und Ausdrücke. Seiteneffekte sind schwer mit den Prinzipien von SdV vereinbar; seiteneffektbehaftete Verträge sind unbrauchbar. Der Programmierstil mit Seiteneffekten behindert das Trennen von Abfragen und Aktionen und erhöht so den Aufwand, die Seiteneffektfreiheit von Verträgen zu garantieren.

## 5.7 Java

Java hat sich vor allem als Implementationssprache für Web-Anwendungen verbreitet, sodass es im Kontext von eBusiness-Technologien Beachtung verdient. Java unterstützt SdV nicht direkt, es bietet nicht einmal allgemeine Zusicherungen.[5] Dieser Mangel hat zu einer lebhaften Diskussion und zahlreichen Ansätzen geführt, Java um Vertragskonstrukte zu ergänzen. Links dazu finden sich in [8], zu SdV-Werkzeugen für Java in [69] unter „DesignByContractInJava“. Verschiedene Techniken für SdV in Java untersucht [46]. Es gibt u.a. folgende Möglichkeiten, Verträge in Java zu realisieren:

(1) **Aufrufe von Zusicherungsklassenmethoden**: Eine oder mehrere Bibliotheksklassen bieten allgemeine und/oder spezielle Zusicherungen als Klassenmethoden. Diese Zusicherungsmethoden lassen sich mit der Ausnahmebehandlung von Java implementieren, sodass sie im Fehlerfall Laufzeitinformation aufbereiten. Der Entwickler programmiert die Verträge als Aufrufe der Zusicherungsmethoden in die

[5] James Gosling erwähnt in einem Interview mit JavaWorld, dass er Zusicherungen, Vor- und Nachbedingungen in eine frühe Java-Spezifikation eingearbeitet hatte, diese aber wegen Termindruck strich, was er heute bedauert.

Rümpfe spezifizierter Methoden. Diesen Ansatz beschreibt [50], Class Eiffel [78] und jassert [66], [67] realisieren ihn. Er entspricht Ansatz 3.9 (2.2) und etwa dem, was Component Pascal für SdV bietet (s. 5.5).

☺ Der Ansatz ist leicht implementierbar, erfordert nur Bibliotheksklassen und öffnet einen schnellen Zugang zu eingeschränkter SdV in Java.

☹ Weitergehende SdV-Konzepte wie Invarianten, Vorzustände, Vererbung von Verträgen und Spezifikation von Interfaces unterstützt der Ansatz nicht. Verträge stehen in Implementationsteilen; sie in Schnittstellendokumente zu extrahieren ist aufwändig. Laufzeitprüfungen der Zusicherungen sind nur durch Auskommentieren vor dem Übersetzen abschaltbar.

(2) **Spezielle JavaDoc-Kommentare**: Der Entwickler schreibt die Verträge als ausgezeichnete Kommentare in den Quelltext zu spezifizierender Klassen. Eine Kombination aus einem Präprozessor und einem normalen Java-Compiler oder ein spezieller Compiler erzeugen aus vertraglich kommentiertem Quelltext modifizierten Code. Diesen Ansatz variieren die Werkzeuge iContract [9], [19], [20], [21], [77], Jass [1], JMSAssert [54], [74] und Jtest & Jcontract [75]. Er entspricht Ansatz 3.9 (1), ohne die Eleganz und den Komfort von Eiffel zu erreichen (s. 5.1).

☺ Der Ansatz erlaubt, viele SdV-Konzepte bis hin zu Quantoren zu realisieren. Verträge gehören zur offiziellen Schnittstellendokumentation, übliche Java-Werkzeuge können sie extrahieren. Durch Verwenden/Nichtverwenden des Präprozessors sind Laufzeitprüfungen an-/abschaltbar.

☹ Der Aufwand, zusätzliche Werkzeuge wie den Präprozessor einzusetzen, könnte ihren breiten Einsatz hemmen.

(3) **Zusätzliche Methoden**: Der Entwickler schreibt die Verträge als zusätzliche Methoden, die bestimmten Entwurfsmustern folgen, in den Quelltext zu spezifizierender Klassen. Ein Klassenlader oder eine Fabrik modifizieren den Code. Diesen Ansatz, der auch 3.9 (1) entspricht, realisiert das Werkzeug jContractor [18].

☺ Das An-/Abschalten von Laufzeitprüfungen kann spät erfolgen.

☹ Vertragsmethoden vervielfachen die Anzahl von Methoden, worunter die Überschaubarkeit leidet.

(4) **Separate Spezifikation**: Der Entwickler schreibt zu jeder zu spezifizierenden Klasse einen separaten Vertragstext in einer Vertragssprache. Ein Vertragscompiler erzeugt daraus Vertragscode, der dynamisch gebunden wird. Diesen 3.9 (2.1) entsprechenden Ansatz realisiert das Werkzeug Handshake [7].

☺ Mit diesem Ansatz lassen sich Klassen und Interfaces unabhängig vom Quelltext spezifizieren. Das An-/Abschalten von Laufzeitprüfungen erfolgt spät.

☹ Getrenntes Speichern von Spezifikations- und Implementationstext erschwert es, beide Texte konsistent zu halten.

(5) **Spracherweiterung**: Der populäre „Request For Enhancements“ mit der „bug id“ 4071460 fordert für Java Zusicherungen, doch lehnen die Java-Verantwortlichen eine Spracherweiterung um Verträge wegen technischer Probleme ab. Eine Java-ähnliche Sprache, die SdV unterstützt, ist JJ [76].

Die Ansätze (2) bis (4) lassen sich mit Techniken wie Codetransformation, Interpretation, Kompilation und Metaprogrammierung (*reflection*) weitgehend ohne Änderungen an Java-Werkzeugen und der JVM realisieren. Die Tabelle auf Seite 26f stellt Merkmale einiger SdV-Werkzeuge für Java zusammen. Manche Werkzeuge weisen interessante spezielle Merkmale auf:

**iContract** hat eine OCL-ähnliche Vertragssprache. iContract verhindert rekursive Aufrufe in Zusicherungen und optimiert geschachtelte Invariantenprüfungen. Wir formulieren als Beispiel wieder eine polymorphe Menge (vgl. 3.3, 5.1, 5.3 und 5.5):

```
/**
 * @inv count() >= 0
 */
public interface Set
{
  /**
   * Number of elements in the set.
   */
  int count();

  /**
   * Does the set contain x?
   *
   * @pre   x != null
   * @post  return implies count() > 0
   */
  boolean has(Element x);

  /**
   * Include x into the set.
   *
   * @pre   x != null
   * @post  has(x)
   * @post  has(x)@pre implies count() == count()@pre
   * @post  !has(x)@pre implies count() == count()@pre + 1
   */
  void put(Element x);
}
```

**Jass** bietet ein Änderungsverbotskonstrukt (s. 3.5) und erlaubt Kommentare in Vertragskommentaren (Java kennt keine geschachtelten Kommentare).

**jContractor** ermöglicht, Klassen oder Objekte zu instrumentieren. Zum Instrumentieren eines einzelnen Objekts muss der Kunde das Objekt mittels einer jContractor-Fabrikfunktion erzeugen. (Kundenquelltext zu ändern ist sicher nachteilig.) Die fabrikbasierte Instrumentierung nutzt Metaprogrammierung in Java (*Reflection API*); sie ist nur für nicht-finale Klassen möglich.

**JMSAssert** verwendet JMScript, eine proprietäre Java-basierte Skriptsprache, die auch anderweitig einsetzbar ist.

**Jtest & Jcontract**: Das Testwerkzeug Jtest führt statische Analysen durch, die u.a. die Vollständigkeit von Verträgen prüfen. (Ist es sinnvoll, für fehlende Vorbedingungen Warnungen auszugeben?) Jcontract erlaubt, Reaktionen zur Laufzeit auf verletzte Zusicherungen flexibel einzustellen.

# Übersicht über SdV-Werkzeuge für Java



| | Werkzeug | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Merkmal** | **Class Eiffel** | **Handshake** | **iContract** | **Jass** | **jassert** | **jContractor** | **JMSAssert** | **Jtest & Jcontract** |
| Invarianten | ☹ | ☺ | ☺ | ☺ | ☹ | ☺ | ☺ | ☺ |
| Vor- & Nachbedingungen | ☺ | ☺ | ☺ | ☺ | ☺ | ☺ | ☺ | ☺ |
| Vorzustände | ☹ | geplant | ☺ | ☺ | ☹ | ☺ | ☺ | ? |
| Verträge vererbbar? | ☹ | ☺ | ☺ | ☺ | ☹ | ☺ | ☺ | ? |
| Interfaces spezifizierbar? | ☹ | ☺ | ☺ | ? | ☹ | nur Vor- & Nachbedingungen über Parameter | ☺ | ? |
| Klassen ohne Quelltext spezifizierbar? | ☹ | ☺ | ☹ | ☹ | ☹ | ☹ | ☹ | ☹ |
| Schleifenvarianten & -invarianten | ☹ | ☹ | ☹ | ☺ | ☹ | ☹ | ☹ | ☹ |
| allgemeine Zusicherungen | ☺ | ☹ | ☹ | ☺ | ☺ | ☹ | ☹ | ☺ |
| Quantoren | ☹ | ☹ | ☺ | ☺ | ☹ | ☹ | ☺ | ☹ |
| disziplinierte Ausnahmebehandlung | ☹ | ☹ | ☹ | ☺ | ☹ | ☹ | ☹ | ☹ |
| Quelltext & Verträge | integriert | separiert | integriert | integriert | integriert | integriert | integriert | integriert |
| Verträge in Dokumente extrahierbar? | ☹ | - | ☺ | ☺ | ☹ | ? | ☺ | ☺ |
| Verträge kompatibel mit Java & JVM? | ☺ | ☺ | ☺ | ☺ | ☺ | ☺ | ☺ | ☺ |

| Merkmal | Werkzeug | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **Class Eiffel** | **Handshake** | **iContract** | **Jass** | **jassert** | **jContractor** | **JMSAssert** | **Jtest & Jcontract** |
| Implementation der Verträge als... | ... in Quelltext integrierte Aufrufe der Klassenmethoden Require, Ensure, Assert u.a. der Klasse Eiffel | ... separater Text, Handshake-Compiler Verträge → Code, Handshake-DLL modifiziert Klassencode zur Laufzeit | ... spezielle JavaDoc-Kommentare im Quelltext, Präprozessor Java & Verträge → Java | ... spezielle JavaDoc-Kommentare im Quelltext, Präprozessor Java & Verträge → Java | ... in Quelltext integrierte Aufrufe der Klassenmethode that der Klassen Require, Ensure, Check | ... zusätzliche Methoden, die Entwurfsmustern folgen, jContractor-Klassenlader modifiziert Klassencode | ... spezielle JavaDoc-Kommentare im Quelltext, Präprozessor Java & Verträge → JMScript, JMScript-DLL lenkt Methodenaufrufe durch JMScriptcode, ohne Klassencode zu modifizieren | ... spezielle JavaDoc-Kommentare im Quelltext, Jcontract-Compiler Java & Verträge → Bytecode |
| An-/Abschalten von Laufzeitprüfungen Zeitpunkt / Bereich | ☹ | JVM-Start / System | Vorübersetzung / Klasse | Vorübersetzung / Klasse | ☹ | Klassenlader-Start \| Laufzeit / System \| Klasse | JVM- & JMScript-Start / Klasse | Übersetzung / Klasse |
| unterstützte Plattformen | alle | Windows, Solaris | Windows, Unix | ? | alle | alle | Windows | Windows NT/ 2K, geplant: Solaris, Linux |
| erhältlich | frei | ? | frei | frei | frei | ? | frei, kommerziell | kommerziell, $1500 |
| Herkunft | Think Tank Ltd, Douglas, GB | University of California, Santa Barbara | Reto Kramer, Reliable Systems | Universität Oldenburg | Jim Weirich | University of California, Santa Barbara | Man Machine Systems, India | ParaSoft, Monrovia, USA |
| Referenzen | [78] | [7] | [9], [19], [20], [21], [77] | [1] | [66], [67] | [18] | [54], [74] | [75] |

## 5.8 .NET, C# und Eiffel#

.NET Framework, das im Juli 2000 vorgestellte Entwicklungsgerüst von Microsoft, unterstützt das Entwickeln von Web- und eBusiness-Anwendungen [36], [57], [58]. Während das Komponentenmodell Enterprise JavaBeans (EJB) auf *einer* Programmiersprache - Java - basiert, zielt .NET auf die Interoperabilität *vieler* Programmiersprachen. .NET-basierte Anwendungen setzen sich aus Komponenten zusammen, die in verschiedenen Sprachen implementiert sein können. Die Komponenten kooperieren nach einem Objekt- und Typmodell (Virtual Object System, VOS), realisiert durch eine Zwischensprache (Microsoft Intermediate Language, MSIL), die allen Übersetzern als Zielsprache dient, und nutzen eine gemeinsame Laufzeitumgebung (Common Language Runtime, CLR). In der Sprache A implementierte Klassen können in der Sprache B implementierte Klassen benutzen und erweitern.

C#, die neue Programmiersprache von Microsoft in der Entwicklungslinie C++/Java, ist auf das Objektmodell von .NET maßgeschneidert. Leider kennt C# keine Zusicherungen, sodass es SdV nicht direkt unterstützt. Wie bei C++ und Java kann man Ausnahmebehandlung nutzen, um Vertragskonstrukte zu realisieren.

Eiffel# ist eine fast vollständige Teilmenge von Eiffel, die als eine der ersten Sprachen neben C# auf .NET portiert wurde. (Im Wesentlichen fehlt Eiffel# zu Eiffel nur mehrfaches Erben von nicht rein abstrakten Klassen; es ist geplant, Eiffel# zu Eiffel zu vervollständigen.) Mit Eiffel# steht in .NET eine Sprache zur Verfügung, die SdV voll entspricht. Darüber hinaus kann man mit einem Werkzeug, dem Contract Wizard, Komponenten anderer Sprachen mit Verträgen ausrüsten, ohne auf deren Quellcode zugreifen zu müssen. Damit sind Voraussetzungen geschaffen, die Vertragsmethode beim Entwikkeln von Microsoft-Windows-basierten eBusiness-Anwendungen sprachübergreifend einzusetzen.

## 5.9 Weitere Ansätze

Eiffel ist bisher die einzige kommerzielle Programmiersprache, die SdV voll unterstützt. Es hat aber die Entwicklung Eiffel-ähnlicher Sprachen mit Vertragskonstrukten initiiert, darunter Sather, Blue, Bullant und Q. Einen Überblick gibt [69].

eBusiness-Anwendungen sollen sich aus vertrauenswürdigen Komponenten zusammensetzen - dahin zielt das **Trusted Components Project**, das Bertrand Meyer, Christine Mingins und Heinz Schmidt an der Monash University leiten [38], [72]. Dieses Forschungs- und Anwendungsprojekt soll der Softwareindustrie eine solide, breite Basis von Bibliotheken vertrauenswürdiger, wiederverwendbarer Komponenten liefern. Eine Kombination verschiedener Ansätze soll die Vertrauenswürdigkeit der Komponenten erreichen: SdV, mathematische Korrektheitsbeweise, Testen, Prüfen durch eine breite Öffentlichkeit wie bei freier Software, Bewerten mittels Metriken, Validieren in praktischen Projekten und strenges Verwalten von Änderungsanforderungen.

Die Vertragsmethode trägt dazu bei, die Qualität von Softwarekomponenten zu erhöhen - sie ist aber kein Allheilmittel, sondern hat Grenzen und lässt Fragen für die Forschung offen. So variiert der kontroverse Vorschlag „Defensive Development“ die Semantik von Vorbedingungen [10]. Meyer und Szyperski diskutieren, was Verträge für die Komponententechnologie bedeuten und wie eine Vertragssprache aussehen könnte, die nicht nur

semantische Eigenschaften von Komponentenschnittstellen ausdrücken kann, sondern auch Leistungs- und Qualitätsmerkmale [35], [61], [62].

# 6 Fazit

SdV ist eine leicht erlernbare Methode, die mit wenig Aufwand den Softwareentwicklungsprozess verbessern und die Qualität des Softwareprodukts erhöhen kann:

☺ Vertragliche Spezifikationen bilden exakte und verständliche Dokumentationen, die die Benutzbarkeit und Wiederverwendbarkeit von Komponenten fördern.

☺ Verträge sind zur Laufzeit prüfbare Zusicherungen, die als eingebaute Selbsttests die Zuverlässigkeit und Vertrauenswürdigkeit von Komponenten erhöhen.

Wo hohe Qualitätsanforderungen an Software gestellt werden, ist die Vertragsmethode eigentlich unverzichtbar. eBusiness-Anwendungen müssen vertrauenswürdig sein, um Geschäftspartner anzusprechen statt abzuschrecken. Daher sollte SdV übliche Praxis beim Entwickeln von eBusiness-Anwendungen sein. Doch leider ist SdV heute noch die „beste Nicht-Praxis“ in der Softwareindustrie.

Technologische Fortschritte bei der Softwarekonstruktion sind bisher eher langsam in die industrielle Softwarepraxis vorgedrungen. Jeder Schritt - von Maschinen- zu Hochsprachen, von Spaghetticode zu strukturierten Programmen, von monolithischen zu modularen Systemen, von globalen Daten zu gekapselten Objekten, von isolierten Einzellösungen zu wiederverwendbaren Komponenten - hat Jahrzehnte gedauert. Der Innovationsschritt zu SdV ließe sich schnell gehen - gerade bei den vielfältigen neuen eBusiness-Anwendungen, deren Entwicklung sich abzeichnet.

Wer aus der Vertragsmethode den besten Nutzen hinsichtlich der Effektivität des Entwicklungsprozesses und der Qualität der Anwendung ziehen will, dem seien Sprachen und Werkzeuge empfohlen, die SdV voll integriert unterstützen, z.B. Eiffel, BON und EiffelCase. Wer sich jedoch für eine andere Implementationssprache - z.B. Java - entscheidet, braucht deshalb nicht auf SdV zu verzichten, sondern kann geeignete SdV-Werkzeuge einsetzen. Im Bereich Microsoft-Windows-basierter eBusiness-Anwendungen stellen Eiffel# und der Contract Wizard eine Perspektive dar.

# Referenzen

[1] Detlef Bartetzko, Michael Plath: *The Jass Page*. http://semantik.informatik.uni-oldenburg.de/~jass/

[2] Bruce W. Bigby: *General Exception-Handling Facility for The C Programming Language featuring Design by Contract*. http://home.rochester.rr.com/bigbyofrocny/GEF/GEF.html

[3] Robert V. Binder: *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison-Wesley (1999)

[4] Yoonsik Cheon, Gary T. Leavens: *A quick overview of Larch/C++*. Journal of Object-Oriented Programming (October 1994) 39-49

[5] Maurice Cline, Doug Lea: *Using Annotated C++*. Proc. C++ at Work (September 1990)

[6] Patrick Doyle: *Eiffel for Native Speakers of C++*. C++ Report (March 1999) http://www.elj.com/eiffel/cpp/cpp-report-pd/

[7] Andrew Duncan, Urs Hölzle: *Adding Contracts to Java with Handshake*. Department of Computer Science, University of California, Santa Barbara, Technical Report TRCS98-32 (December 1998) http://www.cs.ucsb.edu/oocsb/papers/handshake98.html

[8] Geoff Eldridge: *Java and „Design by Contract“ and the lack thereof...* http://www.elj.com/eiffel/feature/dbc/java/ge/

[9] Oliver Enseling: *iContract: Design by Contract in Java*. JavaWorld (February 2001) http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools_p.html

[10] Donald G. Firesmith: *A Comparison of Defensive Development and Design by Contract*. In: D. Firesmith et. al. (ed): Proc. TOOLS 30 Santa Barbara, IEEE (June 1999) 258-267

[11] Earl Fong: *Being Assertive in C/C++*. http://www.acornsw.com/cujcd/HTML/15.06/FONG/FONG.HTM

[12] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Bonn (1996) 430 S.

[13] Karlheinz Hug: *Module, Klassen, Verträge. Ein Lehrbuch zur komponentenorientierten Softwarekonstruktion mit Component Pascal*. Vieweg, Wiesbaden (2001) 2. Auflage, 446 S.

[14] Interactive Software Engineering: *Building bug-free O-O software: An introduction to Design by Contract*. http://eiffel.com/doc/manuals/technology/contract/page.html

[15] Jean-Marc Jézéquel, Bertrand Meyer: *Design by Contract: The Lessons of Ariane*. IEEE Computer, Vol. 30, No. 1 (January 1997) 129-130; http://eiffel.com/doc/manuals/technology/contract/ariane/page.html

[16] Jean-Marc Jézéquel, Michel Train, Christine Mingins: *Design Patterns and Contracts*. Addison-Wesley, Reading (2000) 348 S.

[17] Paul Johnson: *The Eiffel Contract for C++ Programmers*. http://www.elj.com/eiffel/cpp/eiffel-contract/

[18] Murat Karaorman, Urs Hölzle, John Bruno: *jContractor: A Reflective Java Library to Support Design By Contract.* Department of Computer Science, University of California, Santa Barbara, Technical Report TRCS98-31 (December 1998) http://www.cs.ucsb.edu/oocsb/papers/TRCS98-31.html

[19] Reto Kramer: *iContract - The Java Design by Contract Tool*. TOOLS USA (1998) article, http://www.reliable-systems.com/tools/iContract/documentation/iContract-tools98usa.pdf

[20] Reto Kramer: *Design by Contract in Java. iContract, the Design by Contract Tool for Java*. Object World London (October 1998) slide presentation, http://www.reliable-systems.com/tools/iContract/documentation/web98.pdf

[21] Reto Kramer: *Examples of Design by Contract in Java using iContract, the Design by Contract Tool for Java*. Object World Berlin (May 1999) slide presentation, http://www.reliable-systems.com/tools/iContract/documentation/OW_BERLIN99_WEB.PDF

[22] Barbara H. Liskov, Jeannette M. Wing: *A behavioral notion of subtyping*. ACM Transactions on Programming Languages and Systems, Vol. 16, No. 6 (November 1994) 1811-1841

[23] Phil J. Maker: *GNU Nana: improved support for assertions and logging in C and C++*. http://www.cs.ntu.edu.au/homepages/pjm/nana-home/index.html

[24] David Maley, Ivor Spence: *Emulating Design by Conract in C++*. In: R. Mitchell et. al. (ed): Proc. TOOLS 29 Nancy, IEEE (June 1999) 66-75

[25] James C. McKim, Jr.: *Programming by contract: Designing for Correctness*. Journal of Object-Oriented Programming (May 1996) 70-74

[26] James C. McKim, Jr.: *Advanced Programming By Contract: Designing for Correctness*. TOOLS Pacific (1999) tutorial, http://www.rh.edu/~jcm/tools_pacific_99.pdf

[27] Jim McKim, Richard Mitchell: *Design by Contract by example*. Addison-Wesley (to be published later in 2001) preview of first 3 chapters: http://www.inferdata.com/~richard/extras/dbcextracts/

[28] Bertrand Meyer: *Eiffel: The Language*. Prentice Hall International, Hertfordshire (1992) 594 S.

[29] Bertrand Meyer: *Applying „Design by Contract“*. IEEE Computer, Vol. 25, No. 10 (October 1992) 40-51

[30] Bertrand Meyer: *Reusable Software. The Base Object-Oriented Component Libraries*. Prentice Hall International, Hertfordshire (1994) 514 S.

[31] Bertrand Meyer: *Eiffel: The Reference*. ISE Technical Report TR-EI-41/ER (1995) Version 3.3.4, 100 S.

[32] Bertrand Meyer: *Object-oriented Software Construction*. Prentice Hall PTR, Upper Saddle River (1997) 2nd edition, 1260 S.

[33] Bertrand Meyer: *Eiffel’s Design by Contract: Predecessors and Original Contributions*. (March 97) http://www.elj.com/eiffel/bm/dbc/

[34] Bertrand Meyer: *Toward More Expressive Contracts*. Journal of Object-Oriented Programming (July/August 2000) 39-43

[35] Bertrand Meyer: *Contracts for Components*. Software Development Magazine (July 2000) http://www.sdmagazine.com/articles/2000/0007/0007k/0007k.htm

[36] Bertrand Meyer: *The Significance of dot-NET.* Software Development Magazine (November 2000) http://www.sdmagazine.com/articles/2000/0011/0011l/0011l.htm

[37] Bertrand Meyer: *Design By Contract*. Prentice Hall (to be published October 2001) 350 S.

[38] Bertrand Meyer, Christine Mingins, Heinz Schmidt: *Trusted Components for the software industry.* IEEE Computer (May 1998) http://eiffel.com/doc/manuals/technology/bmarticles/computer/trusted/page.html; http://www.trusted-components.org/documents/tc_original_paper.html

[39] Anna Mikhajlova: *Consistent Extension of Components in the Presence of Explicit Invariants*. In: R. Mitchell et. al. (ed): Proc. TOOLS 29 Nancy, IEEE (June 1999) 76-85

[40] Glenn E. Mitchell: *C++ Object Model Design. Part III: Using Assertions to Implement Design by Contract*. http://www.cbuildermag.com/features/2000/08/cb200008gm_f/cb200008gm_f.asp

[41] Richard Mitchell: *Design by contract. Bringing together formal methods and software design*. (August 1995) http://www.it.brighton.ac.uk/staff/rjm4/stagingposts/handout.dbctutorial95.html

[42] Richard Mitchell: *Analysis by Contract or UML with Attitude.* In: D. Firesmith et. al. (ed): Proc. TOOLS 30 Santa Barbara, IEEE (June 1999) 466-476

[43] Richard Mitchell, James McKim: *Extending a method of devising software contracts*. In: C. Mingins, B. Meyer (ed): Proc. TOOLS 32, IEEE (1999)

[44] Richard Mitchell: *Fulfilling a contract. Satisfying a precondition*. Journal of Object-Oriented Programming (May 2000) 34-37; http://www.inferdata.com

[45] Pablo Moisset de Espanes: *Philosophy behind BetterC*. (March 2000) http://www-scf.usc.edu/~moissetd/betterc/philosophy.txt

[46] Jan Newmarch: *Adding Contracts to Java*. http://pandonia.canberra.edu.au/java/contracts/paper-long.html

[47] *The Object Constraint Language (OCL)*. http://www-4.ibm.com/software/ad/standards/ocl.html

[48] *Object Constraint Language Specification. Version 1.1*. (1 September 1997) http://www.software.ibm.com/ad/ocl

[49] Richard F. Paige, Jonathan S. Ostroff: *A Comparison of the Business Object Notation and the Unified Modeling Language*. Department of Computer Science, York University, Ontario, Technical Report CS-1999-03 (May 1999) http://www.cs.yorku.ca/techreports/1999/CS-1999-03.html

[50] Jeffery E. Payne, Michael A. Schatz, Matthew N. Schmid: *Implementing Assertions for Java, Finding bugs early.* Dr. Dobb’s Journal (January 1998) http://www.ddj.com/articles/1998/9801/9801d/9801d.htm

[51] Todd Plessel: *Design By Contract: A Missing Link In The Quest For Quality Software*. (11 August 1998) http://www.elj.com/eiffel/dbc/

[52] Reinhold Plösch, Josef Pichler: *Contracts: From Analysis to C++ Implementation.* In: D. Firesmith et. al. (ed): Proc. TOOLS 30 Santa Barbara, IEEE (June 1999) 248-257

[53] Sara Porat, Paul Fertig: *Class assertions in C++*. Journal of Object-Oriented Programming (May 1995) 30-37

[54] Krishnan Rangaraajan: *Does Java Support Design by Contract?* Dr. Dobb's Journal (November 1999) http://www.ddj.com/articles/1999/9911/9911m/9911m.htm

[55] Martin Reiser, Niklaus Wirth: *Programmieren in Oberon. Das neue Pascal*. Addison-Wesley, Bonn (1994) 338 S.

[56] David Rosenblum: *A Practical Approach to Programming with Assertions*. IEEE Transactions on Software Engineering, Vol. 21, No. 1 (January 1995)

[57] Raphael Simon, Emmanuel Stapf, Bertrand Meyer, Christine Mingins: *Eiffel on the Web: Integrating Eiffel systems into Microsoft .NET Framework*. (July 2000) http://eiffel.com/doc/manuals/technology/eiffelsharp/white_paper.html, http://msdn.microsoft.com/library/techart/pdc_eiffel.htm

[58] Raphael Simon, Emmanuel Stapf, Christine Mingins, Bertrand Meyer: *Eiffel for E-Commerce under .NET.* Journal of Object-Oriented Programming (October 2000) 42-58

[59] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley, Reading (1997) Third Edition, 911 S.

[60] Clemens Szyperski: *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley Longman, Harlow (1998) 411 S.

[61] Clemens Szyperski: *Components and Contracts*. Software Development Magazine (May 2000) http://www.sdmagazine.com/articles/2000/0005/0005l/0005l.htm

[62] Clemens Szyperski: *Components and Architecture*. Software Development Magazine (October 2000) http://www.sdmagazine.com/articles/2000/0010/0010k/0010k.htm

[63] Kim Waldén, Jean-Marc Nerson: *Seamless Object-Oriented Software Architecture. Analysis and Design of Reliable Systems*. Prentice Hall, New York (1995) 438 S.

[64] Jos B. Warmer, Anneke G. Kleppe: *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley (1999)

[65] Steve Webster: *Object Constraint Language*. slide presentation, http://dec.bournemouth.ac.uk/dec_ind/swebster/UML_OCL/index.htm

[66] Jim Weirich: *Control the DbC assertions in Java Modules*. http://w3.one.net/~jweirich/java/javadbc/jassert.html

[67] Jim Weirich: *Design by Contract for Java*. http://w3.one.net/~jweirich/java/javadbc/java-dbc.html

[68] David Welch, Scott Strong: *An Exception-based Assertion Mechanism for C++*. Journal of Object-Oriented Programming (July/August 1998)

[69] Eiffel Forum wiki: http://efsa.sourceforge.net/cgi-bin/view/Main/

[70] Extraordinarily Large Jumble (Eiffel Liberty): http://www.elj.com

[71] Interactive Software Engineering: http://eiffel.com; http://www.eiffel.com

[72] Trusted Components Initiative: http://www.trusted-components.org

[73] http://www.designbycontract.com

[74] http://www.mmsindia.com

[75] http://www.parasoft.com

[76] http://www.publicstaticvoidmain.com

[77] http://www.reliable-systems.com

[78] http://www.t-tank.com