
Spezifikation durch Vertrag

Mensch	<i>An early advocate of the assertional method of program proving was none other than Alan Turing himself. On 24 June 1950 at a conference in Cambridge, he gave a short talk entitled, "Checking a Large Routine" which explains the idea with great clarity. "How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."</i> Tony Hoare¹
Maschine	<i>Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs. There is a good prospect of doing this, because we can require the computer to do much more work in checking each step than a human is willing to do.</i> John McCarthy²
Theorie	<i>[...] we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.</i> Robert Floyd³
Praxis	<i>I believe that the use of Eiffel-like module contracts is the most important non-practice in software world today. By that I mean there is no other candidate practice presently being urged upon us that has greater capacity to improve the quality of software produced. [...] This sort of contract mechanism is the sine-qua-non of sensible software reuse.</i> Tom de Marco⁴

¹ C. A. R. Hoare: *The Emperor's Old Clothes*. ACM Turing Award Lecture 1980. Comm. ACM, Vol. 24, No. 2 (Feb. 1981) S. 75–83; <http://awards.acm.org/images/awards/140/articles/4622167.pdf> (Zugriff 2012-10-08).

² John McCarthy: *Towards a Mathematical Science of Computation*. In: Cicely M. Popplewell (ed): Proceedings of IFIP Congress, Munich, Germany, Aug. 27 – Sept. 1, 1962, North-Holland (1962) S. 21–28; <http://www-formal.stanford.edu/jmc/towards.pdf> (Zugriff 2012-10-08).

³ Robert W. Floyd: *Assigning Meanings to Programs*. In: Jacob T. Schwartz (ed): Proceedings of Mathematical Aspects of Computer Science, Symposia in Applied Mathematics, Vol. 19, American Mathematical Society (April 1967) S. 19–32; <http://www.cs.virginia.edu/~weimer/2007-615/reading/FloydMeaning.pdf> (Zugriff 2012-10-08).

⁴ Tom de Marco: *Letter commenting "Jean-Marc Jézéquel, Bertrand Meyer: Design by Contract: The Lessons of Ariane. IEEE Computer (Jan. 1997)". IEEE Computer (Feb. 1997)*

Schlagwörter	Spezifikationsmethode, Softwareentwicklungsprozess, Softwarequalität, Zuverlässigkeit, Vertrauenswürdigkeit, Korrektheit, Validierung, Verifikation, Test
Motivation	Erst das Was, dann das Wie klären. Spezifizieren → implementieren → validieren. Nach diesen Prinzipien sollte man Software entwickeln. Spezifikation durch Vertrag (<i>Design by Contract</i>) ist eine Softwareentwicklungsmethode für systematisches Beschreiben, Konstruieren und Prüfen von Komponenten. Sie ist nützlich, theoretisch fundiert und praktisch einsetzbar und sollte daher zur Informatikausbildung gehören.
Lernziele	Der Leser lernt Konzepte, Begriffe und Aspekte der Vertragsmethode anhand von Beispielen in <i>Eiffel</i> so gut kennen, dass er sie im <i>Informatik 3 Praktikum</i> in einem kleinen Projekt anwenden kann.

Inhalt

1	Grundlegendes	3
1.1	Was ist und wozu dient Spezifikation?	3
1.2	Was sind und wozu dienen Verträge?	3
1.3	Klasseninvarianten	6
1.4	Vorbedingungen	7
1.5	Nachbedingungen	7
1.5.1	Vorzustände	8
1.5.2	Abfragenergebnisse	9
1.6	Vertrag mit Invariante und Vor- und Nachbedingung	9
1.7	Initialisierung	10
1.8	Aufgaben	12
2	Tieferschürfendes	13
2.1	Spezifikationsmodelle	13
2.2	Eigenschaften von Verträgen	14
3	Weiterführendes	16
3.1	Elementare, abgeleitete und parametrisierte Abfragen	16
3.2	Wertebereiche und Beziehungen	18
3.3	Rekursion	18
3.4	Quantoren	18
3.5	Vollständigkeit	19
3.6	Änderungsverbote	19
3.6.1	Attributwerte	19
3.6.2	Attributobjekte	23
3.6.3	Argumentobjekte	23
3.7	Nebeneffektfreiheit	24
3.8	Vererbung	24
3.9	Laufzeitprüfungen	25
3.10	Aufgaben	26
4	Einordnendes	26
4.1	Die Vertragsmethode im Softwareentwicklungsprozess	26
4.2	Die Vertragsmethode verglichen mit anderen Ansätzen	27
4.3	Die Vertragsmethode umgesetzt in anderen Sprachen	27
A	Literatur	28

1

Grundlegendes

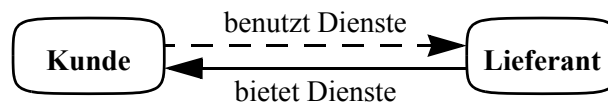
Dieser Abschnitt erläutert Grundkonzepte und -begriffe der Vertragsmethode.

1.1

Was ist und wozu dient Spezifikation?

Das Grundmodell bilden Anwendungen, bei denen (eventuell im Internet verteilte) **Komponenten** kooperieren, um gemeinsam Aufgaben auszuführen. Eine Komponente kann ein Modul, eine Klasse, eine Komponente im Sinne von Komponententechniken (wie Microsofts *COM*, *.NET* oder Suns/Oracles *EJB*), oder eine ähnliche Modellierungseinheit sein. Jede Komponente erfüllt eine Teilaufgabe. Als Kooperationsmodell dient das **Kunden-Lieferanten-Modell** (*client-supplier model*), eine Metapher aus dem Geschäftsleben. Komponenten erscheinen hier in zwei Rollen:

Bild 1
Kunden-Lieferanten-Modell



- Jede Komponente ist **Lieferant** (*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 diese Aspekte zu unterscheiden sind:

- Die **Syntax** umfasst die Namen der Dienste und die Anzahlen, Reihenfolgen und Arten der Parameter.
- Die **statische Semantik** umfasst die Typbindung von Parametern und Ergebnissen von Diensten und die Zugriffsrechte an Diensten.
- Die **dynamische Semantik** umfasst das Verhalten der Dienste, ihre Ergebnisse und ihre Wirkungen auf Zustände von Komponenten.

Qualität durch
Spezifikation

Viele Anwendungen erfordern Zuverlässigkeit und Vertrauenswürdigkeit. Keiner will Schaden nehmen, weil eine Softwarekomponente versagt. Zuverlässiges Funktionieren von Software setzt korrekte Komponenten voraus. Eine Komponente ist **korrekt**, wenn ihre Implementation ihrer Spezifikation entspricht. Daher lässt sich ohne exakte und vollständige Spezifikation nicht über Korrektheit rasonieren. Konventionelle Schnittstellendefinitions- und Implementationssprachen erlauben die Beschreibung der Syntax und statischen Semantik von Diensten durch Signatures, aber selten mehr. Als Kommentare ergänzte informale, verbale Spezifikationen der dynamischen Semantik von Diensten sind oft mehrdeutig und genügen daher nicht den Anforderungen an qualitätsvolle Anwendungen. Diese lassen sich nur mit Methoden zur formalen, exakten Spezifikation der dynamischen Semantik von Komponentenschnittstellen erfüllen.

1.2

Was sind und wozu dienen Verträge?

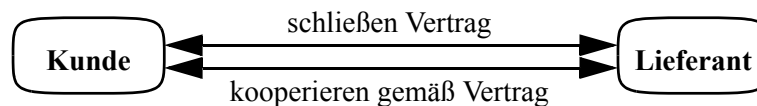
Spezifikation durch Vertrag (SdV, *Design by Contract*¹, *DbC*, *Programming by Contract*) ist eine Methode zur formalen Spezifikation der dynamischen Semantik von Softwarekomponenten mittels Verträgen aus erweiterten booleschen Ausdrücken. Sie beruht auf Theorien zu Zusicherungen und abstrakten Datentypen.

¹ „Design by Contract“ ist ein Warenzeichen von Eiffel Software.

Die Vertragsmethode begründet hat Bertrand Meyer, der mit der Programmiersprache *Eiffel* auch ein hervorragendes Mittel zu ihrer praktischen Anwendung geschaffen hat [Mey92a], [Mey92b], [Mey94], [Mey95], [Mey97], [Mey05a], [Mey05b], [Mey05c], [ECMA367]. Jean-Marc Nerson und Kim Waldén haben die Vertragsmethode auf Analyse und Entwurf übertragen und dazu die Modellierungssprache *BON* (**B**usiness **O**bject **N**otation) entwickelt [WaN95]. Jim McKim und Richard Mitchell haben wesentlich zur Ausarbeitung der Vertragsmethode beigetragen [McK96], [McK99], [MiM01], [MiM99], [Mit00]. Diese und andere Autoren haben Prinzipien und Regeln formuliert, nach denen sich die Vertragsmethode systematisch im Softwareentwicklungsprozess anwenden lässt. Diese Regeln haben zu Leitlinien in diesem Text und in [LS Leitlinien zum Softwareentwurf](#) angeregt.

Verträge ergänzen die Metapher des Kunden-Lieferanten-Modells:

Bild 2
Vertragsmodell



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

- Eine **Abfrage** (*query*) gibt Auskunft über den Zustand ihrer Komponente, ändert aber keine Zustände. Ihr Zweck ist, als Ergebnis einen Wert zu liefern. Die Abfragen einer Komponente beschreiben ihren abstrakten Zustand.
- Eine **Aktion** (*action*) oder eiffelisch ein **Kommando** (*command*) ändert Zustände von Komponenten, liefert aber kein Ergebnis. Ihr Zweck ist, einen Effekt zu erzielen. Die Aktionen einer Komponente bewirken ihre Zustandsänderungen.

Leitlinie 1
Trenne Abfragen und Aktionen

Unterscheide bei Diensten genau zwischen Abfragen und Aktionen!

Der Unterschied zwischen Abfragen und Aktionen soll an ihren Namen deutlich sichtbar sein. Dienste sind ihrem Zweck und ihrer Semantik entsprechend sinnvoll zu benennen. Namen sollen erklären und erhellen, nicht verschleiern oder irreführen.

Leitlinie 2
Benenne Abfragen nach dem, was sie liefern

Wähle als Name einer Abfrage entweder ein Substantiv, welches das gefragte Ding, oder ein Adjektiv oder Partizip, das die gefragte Eigenschaft beschreibt! Ein Substantiv kann mit einem Adjektiv oder Partizip qualifiziert sein. Eine Eigenschaft kann durch eine adjektivähnliche Wortkombination ausgedrückt sein.

Leitlinie 3
Benenne Aktionen nach dem, was sie tun

Wähle als Name einer Aktion ein Verb, das die geforderte Tätigkeit beschreibt! Das Verb kann mit einem Substantiv qualifiziert sein. Der Name beschreibt die Tätigkeit aus der Sicht des Lieferanten, nicht des Kunden.

Das Prinzip der Trennung von Abfragen und Aktionen schließt nebeneffektbehaftete Funktionen als Dienste aus. Ein **nebeneffektbehafteter** Dienst liefert ein Ergebnis *und* verändert als **Nebeneffekt** (*side-effect*) einen Zustand; er ist weder Abfrage noch Aktion, lässt sich nicht prägnant und exakt benennen und verbal nur schwer beschreiben. Oft behindern Nebeneffekte systematisches Entwickeln von Software erheblich, nur selten sind sie nützlich.¹

¹ Das Prinzip der Nebeneffektfreiheit wird auch außerhalb der Vertragsmethode anerkannt, z.B. von Martin Fowler: „**Separate Query from Modifier** You have a method that returns a value but also changes the state of an object. Create two methods, one for the query and one for the modification“ [Fow99] S. 279.

Leitlinie 4

Vermeide

Nebeneffekte

Vermeide Dienste mit Nebeneffekten! Abfragen sind nebeneffektfrei.

Ein Grund für die Leitlinien 1 und 4 ist, dass Abfragen nicht nur als Dienste für Kunden bereitstehen, sondern auch als **Spezifikatoren** (*specifier*) für andere Dienste, d.h. als Elemente von Verträgen. Ein **Vertrag** (*contract*) setzt sich aus Bedingungen zusammen, die zulässige Zustände beschreiben. Die Auswertung von Vertragsbedingungen darf keine Zustandsänderungen, also keine Nebeneffekte bewirken. Vertragsbedingungen teilen sich in drei Arten:

- **Invarianten** (*invariant*) 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; pragmatisch können sie z.B. Naturgesetze (*law of nature*) oder 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 Dienstauftrag 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 Dienstauftrag liefert.

Invarianten, Vor- und Nachbedingungen einer Komponente sind **Vertragsteile**, die zusammen den Vertrag bilden, den die Komponente ihren Kunden bietet. 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 Invariante oder Nachbedingung ein Fehler des Lieferanten.

Tabelle 1

Verantwortlichkeiten

	Kunde	Lieferant
Pflicht	1. Die Vorbedingungen einhalten.	3. Anweisungen ausführen, die die Invarianten erhalten und die Nachbedingungen herstellen.
Nutzen	4. Ergebnisse/Wirkungen nicht prüfen, da sie durch die Nachbedingungen garantiert sind.	2. 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 folgenden Abschnitte erläutern Grundlagen der Vertragsmethode anhand von Beispielen in *Eiffel*, die als modifizierte, oft vereinfachte Auszüge aus der *Eiffel*-Standardbibliotheksklasse *STRING* zusammengestellt sind. **Vertragsteile sind blau**. Jeden Abschnitt beendet ein allgemeines, *Eiffel*-unabhängiges Fazit in Form einer Leitlinie.

Zum Initialisieren von *Eiffel*-Neulingen ein Fingerhut voll *Eiffel*: Die Spezifikationseinheit, oben Komponente genannt, ist die Klasse. Dienste heißen in *Eiffel* Merkmale (*feature*). Kommandos werden durch Prozeduren, Abfragen durch Funktionen oder

Attribute implementiert. Routine ist *Eiffels* Oberbegriff für Prozedur und Funktion. Routinen und Attribute sind stets Teil von Klassen. Parameter heißen in *Eiffel* Argumente. Was mit **class interface** *NAME* beginnt, ist kein Quelltext, sondern eine daraus extrahierte Schnittstelle. Übersetzbare Quelltexte beginnen mit **class** *NAME*.

1.3

Klasseninvarianten

Jede Klasse stellt **Invarianten** an sich selbst. Die Konjunktion der Invarianten einer Klasse heißt **ihre Klasseninvariante**. „Keine Invariante“ ist äquivalent zu „die Invariante ist True“.

Programm 1

Eiffel: Invariante in
STRING

```
class interface STRING
feature
  capacity : INTEGER
    -- Number of characters allocated in Current.

  count : INTEGER
    -- Actual number of characters in Current.

invariant
  count_non_negative: 0 <= count
  count_small_enough: count <= capacity
end -- class STRING
```

Was ist an Programm 1 zu erkennen? Als Spezialfall des **Prinzips der Trennung von Schnittstelle und Implementation** (*interface-implementation separation*) verlangt das **Prinzip des gleichförmigen Zugriffs** (*uniform access*), von der Implementation argumentloser Abfragen zu abstrahieren; die Abfragen *capacity* und *count* sind als Attribute oder argumentlose Funktionen implementierbar, der Zugriff auf sie wird davon unabhängig notiert. (Wo keine Argumente sind, unterbleibt Informationsverschmutzung durch leere Klammerpaare.) *Eiffel*-Klassen können Abfragen nur schreibgeschützt an Kunden exportieren; deshalb sind die in *C** notwendigen Getter-Funktionen in *Eiffel* überflüssig.

Merkmale werden in *Eiffel* verständlich benannt und ggf. doppelt spezifiziert: informal mit Kommentaren, ggf. formal mit Verträgen. Die Kommentare erläutern die Ergebnisse oder Effekte der Merkmale mit anderen Worten, aber kurz und prägnant meist in einer Zeile; verpönt sind kryptische Namen wie



cpct : INTEGER

und aufgeblähte, triviale Kommentare wie



capacity : INTEGER -- This feature is intended to designate, denote, deliver and return
-- the capacity of the current object.

Current bezeichnet das aktuelle Objekt. Das Schlüsselwort **invariant** ist Überschrift eines Abschnitts für Invarianten am Ende des Klassentexts. Die Invarianten

0 <= count
count <= capacity

in Programm 1 schränken die Wertebereiche von *capacity* und *count* ein. Vor Vertragsbedingungen können Marken stehen wie hier

count_non_negative
count_small_enough

Sie dienen der Dokumentation, also der Verständlichkeit des Quelltexts, und dem Testen, da sie zur Laufzeit bei angeschalteter Prüfung und festgestellter Verletzung der Bedingung zusammen mit Laufzeitinformation zum Fehler ausgegeben werden.

Leitlinie 5

Dokumentiere
Invarianten

Formuliere mit den Abfragen der Komponente allgemeine, unveränderliche Aussagen über die Zustände ihrer Exemplare, die vor und nach jedem Aufruf jedes Dienstes gelten, als Invarianten!

1.4**Vorbedingungen**

Jede Routine stellt **Vorbedingungen** an ihre Aufrufer. Die Konjunktion der Vorbedingungen einer Routine heißt **ihre** Vorbedingung. „Keine Vorbedingung“ ist äquivalent zu „die Vorbedingung ist True“.

Programm 2

Eiffel: Vorbedingung in
STRING

```
class interface STRING
feature
  is_integer : BOOLEAN
    -- Does Current represent an INTEGER?

  to_integer : INTEGER
    -- 32-bit integer value.

  require
    represents_an_integer_value: is_integer

invariant
  conversion_correct: is_integer implies to_integer.out ~ Current

end -- class STRING
```

Die Abfrage `is_integer` stellt keine Vorbedingung, darf also immer aufgerufen werden:

```
("hello").is_integer  -- False.
("123").is_integer    -- True.
```

Die Abfrage `to_integer` stellt die Vorbedingung `is_integer`, offenbar ein boolescher Ausdruck, der eine Bedingung an den Zustand des aktuellen Objekts *Current* darstellt. `to_integer` darf also nur aufgerufen werden, wenn das Objekt eine Ganzzahl enthält:

```
("hello").to_integer  -- undefiniert, da Vorbedingung verletzt.
("123").to_integer    -- 123 als Ganzzahl.
```

Ein Kunde, der eine Zeichenkette in eine Zahl konvertieren will, muss die beiden Abfragen nacheinander aufrufen: erst das unbedingte `is_integer`, dann bei *True*-Antwort das bedingte `to_integer`. Deshalb sei das hier angewandte Programmierschema **Abfrage-Abfrage-Schema** genannt.

Was ist noch an Programm 2 zu erkennen? Vorbedingungen sind mit dem Schlüsselwort **require** überschrieben. Die Invariante

`is_integer implies to_integer.out ~ Current`

spezifiziert, dass beim Hin-und-Zurück-Konvertieren der Zeichenkette in eine Ganzzahl (mit `to_integer`) und zurück in eine Zeichenkette (mit `out`) das Ergebnis dem Original gleicht:

`("123").to_integer.out ~ (123).out ~ "123".`

Leitlinie 6

Dokumentiere
Vorbedingungen

Formuliere für jeden Dienst (außer parameterlose boolesche Abfragen) passende Vorbedingungen mittels Abfragen und Parametern! Parameterlose boolesche Abfragen sollen i.d.R. keine Vorbedingungen stellen.

1.5**Nachbedingungen**

Jede Routine einer Klasse garantiert ihren Aufrufern **Nachbedingungen**, sofern diese die Vorbedingungen erfüllt haben. Die Konjunktion der Nachbedingungen einer Rou-

tine heißt **ihre** Nachbedingung. „Keine Nachbedingung“ ist äquivalent zu „die Nachbedingung ist True“.

Programm 3
Eiffel: Nachbedingung
in STRING

```
class interface STRING
feature
  count : INTEGER
    -- Actual number of characters in Current.

  wipe_out
    -- Remove all characters.
  ensure
    is_empty: count = 0
end -- class STRING
```

Das Kommando *wipe_out* bewirkt einen Effekt auf die Abfrage *count*: Es löscht alle Zeichen aus dem aktuellen Objekt, sodass die neue Anzahl der Zeichen 0 ist. An Programm 3 ist zu sehen, dass Nachbedingungen dem Schlüsselwort **ensure** folgen.

Lokale Variablen dürfen in Nachbedingungen nicht vorkommen. Da sie beim Verlassen der Routine ungültig werden, wäre es sinnlos, Bedingungen an sie zu garantieren.

Leitlinie 7
Dokumentiere
Nachbedingungen von
Aktionen

Formuliere für jede Aktion Nachbedingungen, die die Effekte der Aktion auf Abfragen abhängig von Parametern der Aktion beschreiben!

1.5.1

Vorzustände

Oft will man ausdrücken, wie sich der Zustand des aktuellen Objekts durch ein Kommando ändert, also den neuen Zustand *nach* dem Kommandoaufruf mit dem alten Zustand *vor* dem Kommandoaufruf vergleichen. Dazu dient das **old**-Konstrukt, das als eine Erweiterung der booleschen Ausdrücke nur in Nachbedingungen verwendbar ist.

Programm 4
Eiffel: Nachbedingung
mit old in STRING

```
class interface STRING
feature
  insert_character (c : CHARACTER; i : INTEGER)
    -- Insert c at index i, shifting characters between ranks i and count rightwards.
  require
    valid_insertion_index: 1 <= i and i <= count + 1
  ensure
    one_more_character: count = old count + 1
    inserted: item (i) = c
    stable_before_i:
      Elks_checking implies substring (1, i - 1) ~ (old substring (1, i - 1))
    stable_after_i:
      Elks_checking implies substring (i + 1, count) ~ (old substring (i, count))
end -- class STRING
```

In Programm 4 besteht die Nachbedingung von *insert_character* aus vier Einzelbedingungen. Sie sind untereinander hingeschrieben und werden implizit konjunktiv verknüpft. Das spart dem Programmierer das explizite Verknüpfen der Bedingungen mit **and**. Die Nachbedingung

count = **old** *count* + 1

verlangt, dass nach dem Einfügen eines Zeichens die Anzahl der Zeichen um 1 höher als vorher ist. Die Nachbedingung

substring (1, *i* - 1) ~ (**old** *substring* (1, *i* - 1))

spezifiziert, dass das Anfangsstück vor dem eingefügten Zeichen gleich geblieben ist. Die Nachbedingung

substring (i + 1, count) ~ (old substring (i, count))

fordert, dass das Endstück nach dem neuen Zeichen gegenüber vorher um 1 nach rechts verschoben ist. Offenbar kann man mit **old** nicht nur ausdrücken, was sich geändert hat, sondern auch, was ungeändert blieb, wie *capacity* nach *wipe_out* in Programm 5.

1.5.2

Abfragenergebnisse

Programm 5 zeigt auch, wie Ergebnisse von Abfragen in Nachbedingungen spezifizieren werden: Da eine Abfrage einen Wert aus dem Wertebereich eines Typs liefert, ist sie typisiert, d.h. an einen Typ gebunden. Zu jeder Abfrage ist die lokale Variable *Result* vom Ergebnistyp der Abfrage implizit vereinbart. *Result* enthält den Ergebniswert des Abfragenaufrufs und kann in den Nachbedingungen der Abfrage verwendet werden.

Programm 5

Eiffel: Nachbedingung
mit Result in STRING

```
class interface STRING
feature
  capacity : INTEGER
    -- Number of characters allocated in Current.
  ensure
    capacity_non_negative: Result >= 0
  count : INTEGER
    -- Actual number of characters in Current.
  ensure
    count_non_negative: Result >= 0
    count_small_enough: Result <= capacity
  wipe_out
    -- Remove all characters.
  ensure
    is_empty: count = 0
    same_capacity: capacity = old capacity
end -- class STRING
```

Die Nachbedingungen der parameterlosen Abfragen *capacity* und *count* in Programm 5 sind einfach: Sie schränken nur den Wertebereich der Abfragen ein. Solche Nachbedingungen lassen sich auch als Klasseninvarianten formulieren. Die Invarianten

capacity >= 0
count >= 0
count <= *capacity*

sind semantisch äquivalent mit den entsprechenden Nachbedingungen in Programm 5. Die erste Invariante ist redundant, da sie aus den andern beiden Invarianten folgt. Deshalb ist sie in Programm 1 weggelassen. Pragmatisch unterscheiden sich die beiden Formulierungen bei angeschalteter Prüfung der Verträge zur Laufzeit: Eine Nachbedingung wird nur geprüft, wenn ihre Routine in einem Implementationsteil aufgerufen wird. Eine Klasseninvariante wird vor und nach jedem Routinenaufruf in einem Implementationsteil geprüft, also viel öfter. Die Formulierung als Klasseninvariante wird also stärker getestet und beansprucht entsprechend mehr Laufzeit.

1.6

Vertrag mit Invariante und Vor- und Nachbedingung

Als kleines Beispiel mit allen drei Arten von Vertragsteilen modelliert Programm 6 natürliche Zahlen aus einem ganzzahligen Grundtyp.

Programm 6
Eiffel: Vertrag in
NATURAL

```

class interface NATURAL
feature
  value : INTEGER
  set (new_value : INTEGER)
    require
      new_value_positive: new_value > 0
    ensure
      new_value_accepted: value = new_value
invariant
  value_positive: value > 0
end -- class NATURAL

```

Der Vertrag in Programm 6 wird in der Reihenfolge Invariante → Nachbedingung → Vorbedingung konstruiert. Im **invariant**-Abschnitt schränkt die Invariante *value > 0* den Wertebereich der Abfrage *value* ein. Das Kommando *set*, eine einfache Setter-Prozedur, soll bewirken, dass *value* den als Argument *new_value* übergebenen Wert liefert. Der **ensure**-Abschnitt mit der Nachbedingung *value = new_value* spezifiziert dies. Da *set* die Invariante nicht verletzen darf, schränkt es im **require**-Abschnitt mit der Vorbedingung *new_value > 0* den akzeptablen Wertebereich von *new_value* ein. *set* lässt sich leicht mit einer Zuweisung implementieren:

```

set (new_value : INTEGER)
  require
    new_value_positive: new_value > 0
  do
    value := new_value
  ensure
    new_value_accepted: value = new_value
end -- set

```

Das Beispiel dient in Tabelle 2 dazu, Unterschiede zwischen Spezifikation und Implementation zusammenzustellen [Mey05a] Folie 31:

Tabelle 2
Spezifikation und
Implementation

Spezifikation	Implementation
Was?	Wie?
ensure <i>value = new_value</i>	do <i>value := new_value</i>
einen Zustand beschreibend	eine Zustandsänderung vorschreibend
deskriptiv	preskriptiv
applikativ	imperativ
denotational	operational
Ausdruck, wird ausgewertet	Anweisung, wird ausgeführt
kann Abfragen aufrufen	kann Aktionen aufrufen

1.7

Initialisierung

Mit der Regel von S. 5, dass die Invarianten einer Komponente vor und nach jedem Aufruf jedes Dienstes gelten, stellt Programm 6 exemplarisch die Frage der Initialisierung der Komponente – eines Moduls oder hier eines Objekts *n* der Klasse *NATURAL*: *n.value* muss schon vor dem ersten Aufruf von *n.set* größer 0 sein, d.h. das an *n* gebun-

dene Objekt muss so initialisiert werden, dass seine Abfragen nach der Initialisierung die Klasseninvariante erfüllen.

Programmiersprachen bieten zum Initialisieren von Daten (Variablen, Objekten) meist die Optionen

- (1) implizite, automatische Initialisierung mit Standardwerten (*default value*),
- (2) explizite, programmierte Initialisierung mit programmiererdefinierten Werten, oft durch parametrisierte Initialisierungsprozeduren, die normale Prozeduren oder spezielle Sprachkonstrukte sein können (Erzeugungsprozeduren in *Eiffel*, Konstruktoren in *C**).

Option (1) genügt, wenn die mit Standardwerten initialisierten Daten der Komponente ihre Invarianten erfüllen. Das trifft erfreulicherweise oft zu, aber freilich nicht immer, wie Programm 6 zeigt. Die Standardwerte sind in *Eiffel* wie in vielen Sprachen *False*, 0, 0.0, "", *Void* usw.. Demnach wird *n.value* implizit mit 0 initialisiert, sodass die Invariante von *n* verletzt ist. Die Lehre aus dem Beispiel sind diese allgemeinen Regeln:

Der Zweck der Initialisierung einer Komponente ist, ihre Invarianten herzustellen. Stellt die Standardinitialisierung nach Option (1) die Invarianten nicht implizit her, so muss es mindestens eine Initialisierungsprozedur nach Option (2) geben, die die Invarianten explizit herstellt. Eine der Initialisierungsprozeduren muss aufgerufen werden, wenn die Komponente zu existieren beginnt. Vor dem Aufruf einer Initialisierungsprozedur ist keine Aussage über die Invarianten möglich, d.h. sie müssen noch nicht gelten. Die Regel von S. 5 wird so modifiziert:

Leitlinie 8
Initialisiere
invariantengemäß

Die Invarianten einer Komponente gelten nach jedem Aufruf jeder Initialisierungsprozedur und vor und nach jedem Aufruf jedes nicht-initialisierenden Dienstes.

In *Eiffel* heißen Initialisierungsprozeduren **Erzeugungsprozeduren** (*creation procedure*) und haben diese Eigenschaften:

- (3) Erzeugungsprozeduren sind syntaktisch und semantisch normale Prozeduren, denen nur per Deklaration im Klassenkopf die Rolle als Erzeugungsprozeduren zugeordnet ist.
- (4) Erzeugungsprozeduren stellen keine Vorbedingungen an Abfragen.
- (5) Falls eine Klasse Erzeugungsprozeduren hat, so muss eine davon bei einer Objekterzeugung aufgerufen werden. Diese Regel löst das Problem der fehlenden Initialisierung, eines häufigen Programmierfehlers.
- (6) Erzeugungsprozeduren können auch außerhalb ihrer Spezialrolle wie normale Prozeduren aufgerufen werden, sofern die Rechte dies zulassen.

In den Eigenschaften (4) und (5) gleichen sich *Eiffels* Erzeugungsprozeduren und *C**s Konstruktoren, in (3) und (6) unterscheiden sie sich.

Um das Initialisierungsproblem in Programm 6 zu lösen, genügt es, *set* zu einer Erzeugungsprozedur zu erklären. Das geschieht in Programm 7 mit einem **create**-Abschnitt. Erlaubt ist es, weil *set* keine Vorbedingung an den Zustand des Datenelements *value* stellt, sondern nur eine an das Argument *new_value*.

Programm 7

Eiffel: NATURAL mit
Erzeugungsprozedur

```

class NATURAL
  create
    set
  feature
    value : INTEGER
    set (new_value : INTEGER)
      require
        new_value_positive: new_value > 0
      do
        value := new_value
      ensure
        new_value_accepted: value = new_value
      end -- set
  invariant
    value_positive: value > 0
end -- class NATURAL

```

Kundenseitig ist zum Benutzen der Klasse *NATURAL* mit

n : *NATURAL*

in einem Vereinbarungsteil eine Referenzgröße *n* des statischen Typs *NATURAL* zu vereinbaren und mit

create *n.set* (123)

in einem Anweisungsteil ein Objekt des Typs *NATURAL* dynamisch zu erzeugen, mit *set* zu initialisieren und an *n* zu binden.

1.8**Aufgaben**

Das Folgende ist Wiederverwertetes aus [Hug01], [HugI12].

Aufgabe 1

Kreis spezifizieren

Spezifizieren Sie einen Kreis nach der Vertragsmethode! (Man kann den Radius, den Umfang und die Fläche des Kreises abfragen und einstellen.) ☐

Aufgabe 2

Widerstandsschaltung
spezifizieren

Spezifizieren Sie eine Schaltung mit einem ohmschen Widerstand nach der Vertragsmethode! (Man kann Widerstand, Spannung und Stromstärke abfragen und einstellen.) ☐

Aufgabe 3

Rechteck spezifizieren

Spezifizieren Sie ein Rechteck nach der Vertragsmethode! (Man kann die Breite, die Höhe, die Diagonale, den Umfang und die Fläche des Rechtecks abfragen und die Breite und die Höhe einstellen.)

Hinweis: Transformieren Sie arithmetische Ausdrücke mit Wurzeln in äquivalente Ausdrücke ohne Wurzeln! ☐

Aufgabe 4

Dreieck spezifizieren

Spezifizieren Sie ein Dreieck nach der Vertragsmethode! (Man kann die drei Seiten, den Umfang, die Fläche und die Eigenschaften ungleichseitig (*scalene*), gleichschenkelig (*isosceles*), gleichseitig (*equilateral*), rechtwinklig (*right-angled*), stumpfwinklig (*obtuse*) und spitzwinklig (*acute*) des Dreiecks abfragen und die drei Seiten zusammen (nicht einzeln!) einstellen.)

Hinweis: Sind *a*, *b*, *c* die drei Seiten und *s* die Hälfte des Umfangs, so ist die Fläche nach der Formel von Heron gleich der Wurzel $\sqrt{s(s-a)(s-b)(s-c)}$. ☐

Aufgabe 5

Uhr spezifizieren

Spezifizieren Sie eine Uhr nach der Vertragsmethode! (Man kann die Stunde und die Minute abfragen und einstellen. Die Uhr kann ticken.) Eine Lösung steht in [Skript Teil 1 Kapitel 2, Abschnitt 2.3](#). ☐

- Aufgabe 6**
Datum spezifizieren
Spezifizieren Sie eine Datumsanzeige nach der Vertragsmethode! (Man kann das Jahr, den Monat und den Tag abfragen und einstellen. Täglich kann man das Tageskalenderblatt abreißen.) □
- Aufgabe 7**
Kaffeeautomat spezifizieren
Spezifizieren Sie einen Kaffeeautomaten, wie er z.B. in der Mensa steht, nach der Vertragsmethode! (Das Betriebspersonal kann den Preis einer Tasse Kaffee einstellen und den Münzbehälter leeren. Man kann Münzen eingeben, eine leere Tasse hinstellen und gefüllt wegnehmen und Rückgeld entnehmen.) Eine Lösung steht in [Hug01]. □
- Aufgabe 8**
Geldautomaten spezifizieren
Spezifizieren Sie nach der Vertragsmethode Geldautomaten, an denen Sie
(1) mit Ihrer Scheckkarte Geld abheben;
(2) Ihre Geldkarte mit Geld auffüllen! □
- Aufgabe 9**
CD-Gerät spezifizieren
Spezifizieren Sie ein CD-Gerät nach der Vertragsmethode! (Man kann eine CD reinschieben und rausholen, das Gerät ein- und ausschalten, starten und anhalten, den nächsten und den vorigen Titel wählen, sowie pausieren. Abfragen kann man, ob das Gerät aus oder an ist, falls es an ist ob es spielt oder nicht, und falls es spielt die Nummer des gerade gespielten Titels.) □
- Aufgabe 10**
Getriebebeschaltung spezifizieren
Spezifizieren Sie ein Getriebe mit vier Gängen und einem Rückwärtsgang nach der Vertragsmethode! (Man kann den eingelegten Gang, die Übersetzungsverhältnisse der Gänge und die Umdrehungszahlen an der Eingangs- und der Ausgangswelle abfragen, in einen anderen Gang umschalten und die Umdrehungszahl der Eingangswelle erhöhen oder erniedrigen.) □

2

Tieferschürfendes

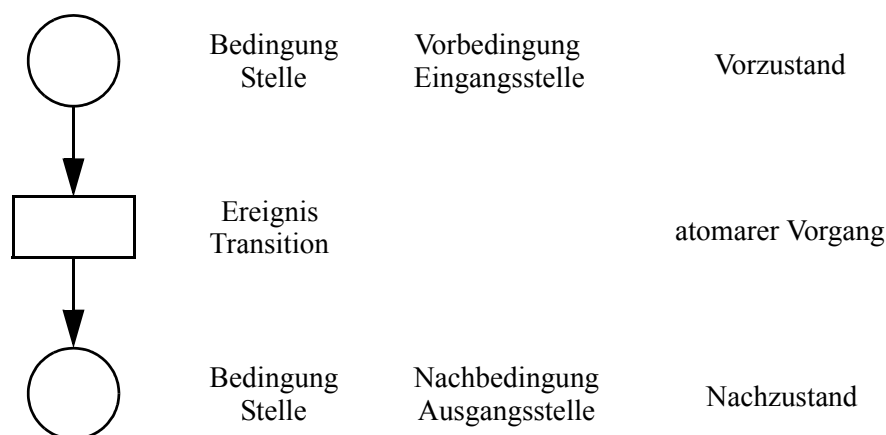
Dieser Abschnitt informiert kurz über theoretische Grundlagen der Vertragsmethode und diskutiert Eigenschaften von Verträgen.

2.1

Spezifikationsmodelle

Praktische Anwendung in der Entwicklung qualitätvoller Software ist das Ziel der Vertragsmethode. Zu ihren Quellen gehören theoretische Arbeiten zur Spezifikation und Verifikation von Programmen, die mit Ideen von Alan Turing, John McCarthy, Peter Naur, Robert Floyd, Tony Hoare, Edsger Dijkstra und anderen beginnen.

Bild 3
Imperatives Modell als
Petri-Netz



Das Grundmodell der imperativen Programmierung ist in Bild 3 grafisch als *Petri-Netz* dargestellt [Pet62]. Das Ereignis oder die Transition ist als Befehl, Instruktion, Anweisung, Aktion, Kommando, Prozeduraufruf usw. zu interpretieren, die Bedingungen oder Stellen als Vor- und Nachzustände, Voraussetzungen und Wirkungen usw.. Ein textuelles Pendant zu Bild 3 ist das **hoaresche Tripel** $\{P\} S \{Q\}$ aus Vorbedingung P , Anweisung S und Nachbedingung Q in Bild 4 [Hoa69].

Bild 4

Hoaresches Tripel

$\{P\}$	Vorbedingung <i>precondition</i>	Aussage, beschreibt Vorzustand
S	Anweisung <i>statement</i>	Befehl, ändert Zustand
$\{Q\}$	Nachbedingung <i>postcondition</i>	Aussage, beschreibt Nachzustand

Im hoareschen Tripel steht S für eine Anweisung eines imperativen Programms, P und Q stehen für Aussagen über Zustände von Variablen des Programms, und es bedeutet: Ist der Programmablauf in einem Zustand, in dem die Vorbedingung P erfüllt ist (Anfangszustand), und wird dann die Anweisung S ausgeführt, so ist der Programmablauf danach in einem Zustand, in dem die Nachbedingung Q erfüllt ist (Endzustand), sofern er terminiert. Das hoaresche Tripel ist der Kern der *axiomatischen Semantik*, die für zwei Zwecke einsetzbar ist:

- (1) **Spezifikation der Semantik von Anweisungen:** Bietet eine Programmiersprache die Anweisungsarten $S_1, \dots, S_n$, so gib zu jeder Anweisungsart S_i ein Bedingungs paar P_i, Q_i so an, dass das Tripel $\{P_i\} S_i \{Q_i\}$ als Muster für jeden konkreten Fall dient. Die Tripel sind Beweisregeln, „Axiome“ für Verifikationen.
- (2) **Verifikation der Korrektheit von Programmen:**
 - (a) **Korrektheit:** Gegeben sei eine (Programm-)Spezifikation $\{P, Q\}$ und eine (Programm-)Implementation S . S heißt **partiell korrekt** (bzgl. $\{P, Q\}$), wenn $\{P\} S \{Q\}$ gilt. S heißt **korrekt**, wenn es partiell korrekt ist und terminiert.
 - (b) **Verifikation:** Ist zur Spezifikation $\{P, Q\}$ eine Implementation S gegeben, die aus der Anweisungsfolge $S_1; \dots; S_n$ besteht, so beweise die Korrektheit von $\{P\} S \{Q\}$, indem du (α) Tripel $\{P_i\} S_i \{Q_i\}$ mit $P = P_1, Q_i \Rightarrow P_{i+1}, Q_n = Q$ bildest und (β) die Terminierung von jedem S_i beweist.

Bertrand Meyer hat diese Ideen zur Verifikation von Algorithmen mit der von Barbara Liskov und anderen entwickelten Theorie der abstrakten Datentypen verbunden. Mit obigem Formalismus und den Bezeichnungen

C	Name einer Komponente (Modul, ADT, Klasse)
Inv	Invariante von C
Pre_s	Vorbedingung des Dienstes s von C
$Post_s$	Nachbedingung des Dienstes s von C
$Impl_s$	Implementation des Dienstes s von C

lässt sich die Korrektheit von C definieren [Mey05a] Folie 34. Die Komponente C heißt **korrekt**, wenn gilt:

- für jede Initialisierungsprozedur ip von C : $\{Pre_{ip}\} Impl_{ip} \{Inv \wedge Post_{ip}\}$, und
- für jeden nicht-initialisierenden Dienst s von C : $\{Inv \wedge Pre_s\} Impl_s \{Inv \wedge Post_s\}$.

2.2

Eigenschaften von Verträgen

Was ist beim vertraglichen Spezifizieren zu beachten? Was charakterisiert gute, was schlechte Verträge?

- ⊗ Ist die Invariante einer Klasse stets falsch, so darf es in einem korrekten System kein Objekt dieser Klasse geben, weil keine Initialisierungsprozedur dieser Klasse terminieren darf.

- ⊗ Ist die Vorbedingung einer Routine stets falsch, so darf die Routine nie aufgerufen werden.
- ⊗ Ist die Nachbedingung einer Routine stets falsch, so darf die Routine nicht terminieren.

Alle drei Randfälle sind praktisch nutzlos. Deshalb sind sie beim Spezifizieren von Klassen zu vermeiden. Die Überlegung führt zu folgenden Definitionen. Eine vertragliche Spezifikation einer Klasse heißt

- **konsistent** (*consistent*), wenn sich keine Vertragsteile widersprechen, insbesondere keines ihrer Vertragsteile stets falsch, d.h. zu stark ist, d.h. zu wenige Zustände zulässt; sonst heißt sie **inkonsistent**;
- **zyklenfrei** (*cycle-free*), wenn in den Vorbedingungen ihrer Abfragen keine zirkulare Abhängigkeit vorkommt; sonst heißt sie **zyklisch**;
- **vollständig** (*complete*), wenn ihre Vertragsteile nicht zu schwach sind, d.h. nicht zu viele Zustände zulassen; sonst heißt sie **unvollständig**;
- **unabhängig** (*independent*), wenn keine einzelne Vertragsbedingung aus anderen Bedingungen folgt; sonst heißt sie **redundant**.

Programm 8
Eiffel: Zyklische
Spezifikation



```
class interface A
feature
  b : BOOLEAN
    require
      c
  c : BOOLEAN
    require
      b
  do_it
    require
      b
end
```

In Programm 8 führt die Vorbedingung von *do_it* in den Zyklus $b \leftrightarrow c$. Der Zyklus verschwindet durch Streichen der Vorbedingung von *b* oder von *c*.

Programm 9
Eiffel: Inkonsistente
Spezifikation



```
class interface A
feature
  b : BOOLEAN
    require
      b and c
  c : BOOLEAN
    ensure
      Result /= b
invariant
  b = c
  b /= c
end
```

In Programm 9 gilt: Die Vorbedingung von *b* führt in einen Zyklus. Ist die Nachbedingung von *c* wahr, so ist die Vorbedingung von *b* falsch. Die beiden Bedingungen widersprechen sich, sind inkonsistent. Die beiden Invarianten können nicht beide wahr sein, auch sie widersprechen sich, sind inkonsistent. Wir beseitigen die Inkonsistenzen, indem wir zwei Bedingungen streichen.

Programm 10Eiffel: Redundante
Spezifikation

```

class interface A
feature
  b : BOOLEAN
  c : BOOLEAN
  ensure
    Result /= b
invariant
  b /= c
end

```

In Programm 10 gilt: Die Nachbedingung von c und die Invariante sind gleich, also nicht unabhängig voneinander. Die Klasse ist überspezifiziert. Eine der beiden Bedingungen ist redundant, verzichtbar. Welche soll wegfallen, welche bleiben? Die Invariante betont eher das Allgemeine der Beziehung zwischen b und c , die Nachbedingung eher das Besondere. Ein praktischer Kosten-/Nutzenaspekt sind die Laufzeitprüfungen der Verträge.

Programm 11Eiffel: Kostengünstige
Spezifikation

```

class interface A
feature
  b : BOOLEAN
  c : BOOLEAN
  ensure
    Result /= b
end

```

In Programm 11 wird bei angeschalteten Laufzeitprüfungen der Verträge die Nachbedingung von c nur nach einem Aufruf von c geprüft.

Programm 12Eiffel:
Prüfungsgünstige
Spezifikation

```

class interface A
feature
  b,
  c : BOOLEAN
invariant
  b /= c
end

```

In Programm 12 wird bei angeschalteten Laufzeitprüfungen der Verträge die Invariante vor und nach jedem Aufruf jeder Routine geprüft.

Leitlinie 9Spezifiziere konsistent,
zyklenfrei, vollständig,
unabhängig

Eine vertragliche Spezifikation muss konsistent und zyklenfrei sein, sie sollte möglichst vollständig sein, was aber oft schwer zu erreichen ist, und sie sollte unabhängig sein.

3**Weiterführendes**

Nachdem die Grundkonzepte und -begriffe der Vertragsmethode bekannt sind, stellt dieser Abschnitt weitere Aspekte vor.

3.1**Elementare, abgeleitete und parametrisierte Abfragen**

Abfragen dienen dazu, Abfragen und Aktionen zu spezifizieren. Um Zyklen 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 höchstens verbal beschrieben: „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). Elementare Abfragen dienen als **Spezifikatoren** (*specifier*), d.h. zur Spezifikation von **abgeleiteten Abfragen** (*derived query*) und Aktionen. Spezifikatoren erscheinen also als Operanden von Verträgen. Sind die Spezifikatoren verständlich, so sollten auch die damit formulierten Verträge verständlich sein.

Programm 13

Eiffel: Elementare und
abgeleitete Abfragen in
STRING

```
class interface STRING
  feature
    capacity : INTEGER
      -- Number of characters allocated in Current.

    count : INTEGER
      -- Actual number of characters in Current.

    is_empty : BOOLEAN
      -- Is structure empty?

    full : BOOLEAN
      -- Is structure full?

    valid_index (i : INTEGER) : BOOLEAN
      -- Is i within the bounds of Current?

  ensure
    definition: Result = (1 <= i and i <= count)

  invariant
    count_non_negative: 0 <= count
    count_small_enough: count <= capacity
    empty_definition: is_empty = (count = 0)
    full_definition: full = (count = capacity)

end -- class STRING
```

In Programm 13 sind *capacity* und *count* elementare Abfragen, *is_empty* und *full* sind abgeleitete Abfragen. In den letzten beiden Invarianten erscheinen *capacity* und *count* als Spezifikatoren, mit denen *is_empty* und *full* vollständig spezifiziert sind.

Exkurs. In der Eiffel-Standardbibliothek erbt *STRING* *is_empty* und *full* von abstrakten Vorgängerklassen; vermutlich deshalb sind deren Kommentare in Programm 13 redundant. Besser wären sie mit anderen Worten beschrieben:

```
is_empty : BOOLEAN
  -- Does Current contain no character?

full : BOOLEAN
  -- Is Current filled to capacity?
```

Exkurs. Warum sind die Namen *is_empty* und *full* uneinheitlich gebildet? Warum nicht *empty*? Weil „empty“ nicht nur ein Adjektiv, sondern auch ein Verb und damit ohne Kontext missverständlich ist: Es könnte statt der Eigenschaft „leer“ auch das Kommando „leeren“ bedeuten, das zur Vermeidung eines Missverständnisses eindeutig *wipe_out* heißt. Und warum dann nicht auch *is_full*? Weil bei „full“ und „to fill“ kein Missverständnis zu befürchten ist und das Präfix *is_* nur dazu dient, Zweideutigkeiten aufzulösen; sonst ist es redundanter Ballast.

Leitlinie 10

Bestimme einfache
Spezifikatoren

Trenne elementare und abgeleitete Abfragen! Spezifiziere elementare Abfragen nicht vertraglich, sondern nutze sie als Spezifikatoren! Spezifiziere abgeleitete Abfragen in Termen elementarer Abfragen!

In Programm 13 ist *valid_index* eine parametrisierte Abfrage, die mit der elementaren Abfrage *count* und ihrem Argument *i* vollständig spezifiziert ist.

Leitlinie 11

Dokumentiere
Nachbedingungen von
Abfragen

Formuliere für jede abgeleitete oder parametrisierte Abfrage eine Nachbedingung, die das Ergebnis der Abfrage mittels elementarer Abfragen und Parameter beschreibt!

Nun lässt sich Leitlinie 7 präzisieren:

Leitlinie 12

Dokumentiere
Nachbedingungen von
Aktionen

Formuliere für jede Aktion eine Nachbedingung, die den Effekt der Aktion auf jede elementare Abfrage abhängig von Parametern der Aktion beschreibt!

3.2**Wertebereiche und Beziehungen**

Die bisherigen Beispiele haben gezeigt, dass Einschränkungen von Wertebereichen und einfache Beziehungen zwischen Abfragen und Parametern oft vorkommen.

Leitlinie 13

Beachte
Wertebereiche

Untersuche bei Abfragen und Parametern Einschränkungen ihrer Wertebereiche (Intervalle bei Zahlen, nichtleere Referenzen) und formuliere diese als Invarianten, Vor- und Nachbedingungen!

Leitlinie 14

Beachte Beziehungen

Untersuche die Abfragen paarweise auf permanente Beziehungen und formuliere diese als Invarianten!

3.3**Rekursion**

Rekursive Funktionen lassen sich rekursiv vertraglich spezifizieren, wie Programm 14 am Beispiel der Fakultätsfunktion *factorial* zeigt, die zu einer Klasse mit mathematischen Funktionen gehören könnte. Die Nachbedingung besteht aus zwei Bedingungen, nämlich Implikationen, die der rekursiven Definition der Fakultät entsprechen.

Programm 14

Eiffel: Rekursive
Spezifikation

```
factorial (n : INTEGER) : INTEGER
  require
    n_non_negative:  0 <= n
    n_small_enough:  n <= max_n
  ensure
    simple_case:      n <= 1  implies Result = 1
    recursive_case:  n > 1   implies Result = n * factorial (n - 1)
```

Bei angeschalteter Vertragsprüfung zur Laufzeit würden rekursive Aufrufe in Vertragsteilen erheblichen Laufzeitaufwand kosten. Deshalb prüft *Eiffel* Verträge nur bei Aufrufen in Implementationsteilen, nicht bei weiteren Aufrufen in Vertragsteilen.

3.4**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.

Leitlinie 15

Benutze Quantoren

Nutze die Ausdruckstärke der Prädikatenlogik, wo es sinnvoll ist! Bietet die Sprache keine Quantoren, formuliere die Prädikate als Kommentare!

Eiffel hat **Agenten**, die u.a. Quantoren realisieren können [Mey00]. Ausdrücke mit einem Quantor sind gut lesbar; Programm 15, bei dem zwei Inline-Agenten zwei Allquantoren implementieren, liest sich wegen seiner Länge weniger leicht:

Programm 15

Eiffel: Spezifikation mit Quantoren

```
sorted : BOOLEAN
  -- 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)))
```

Für manche Situationen ist auch die Prädikatenlogik erster Stufe nicht ausdrucksstark genug. Dann bieten programmiererdefinierte parametrisierte boolesche Funktionen Auswege.

3.5**Vollständigkeit**

Oft muss informale Spezifikation durch Kommentare die formalen Verträge ergänzen, denn diese sind oft

- ⊗ schon zu komplex, um mit einem Blick erfassbar zu sein;
- ⊗ trotzdem unvollständig.

Informale und formale Spezifikation schließen sich nicht aus, sondern ergänzen sich. Ziel ist, Verträge möglichst vollständig zu formulieren. Vollständige Spezifikation gelingt nur, wenn Abfragen alle Komponentenzustände erfassen, d.h. alle durch Aktionen bewirkten Zustandsänderungen durch Abfragen beschreibbar sind. Die Vertragsmethode verlangt daher, Abfragen und Aktionen sorgfältig aufeinander abzustimmen.

Leitlinie 16

Ergänze informale Beschreibungen

Vervollständige eine partielle formale Spezifikation durch Vertrag mittels informaler, verbaler Beschreibungen der Dienste! Nutze dazu ggf. andere formale Beschreibungen wie prädikatenlogische Ausdrücke!

3.6**Änderungsverbote**

Zustandsänderungen können nur durch Routinen bewirkt werden, und zwar auf folgende drei Arten. Dabei bedeutet salopp formuliert ein Attribut- oder Argumentobjekt ein von einem Attribut bzw. Argument referenziertes Objekt.

- (1) *Attributwert*: Eine Routine bewirkt direkt oder indirekt Änderungen von Werten von Attributen des aktuellen Objekts.
- (2) *Attributobjekt*: Eine Routine ruft eine Routine eines Attributobjekts des aktuellen Objekts auf, die Änderungen im Attributobjekt bewirkt.
- (3) *Argumentobjekt*: Eine Routine ruft eine Routine eines ihrer Argumentobjekte auf, die Änderungen im Argumentobjekt bewirkt.

Spezifizieren heißt genau beschreiben, was sich ändert *und* was sich *nicht* ändert. Zu beschreiben, was sich nicht ändert, kann schwieriger sein als zu beschreiben, was sich ändert. Das Problem ist als *frame problem* bekannt [BMR95]. Eine pragmatische Vorgehensweise ist, nur Änderungen zu beschreiben und pauschal zu postulieren, dass „sich sonst nichts ändert“ [BMR93].

Leitlinie 17

Nimm's leicht

Verzichte in Nachbedingungen darauf, das zu spezifizieren, was sich nicht ändert!

Doch wie wird das pauschale Postulat im Detail prüfbar? Dies ist noch Gegenstand der Forschung. Sie hat Lösungsansätze hervorgebracht, aber noch keinen Konsens erreicht. Folgendes untersucht Verbotsmöglichkeiten für die obigen Änderungsarten.

3.6.1**Attributwerte**

Programm 5 hat schon gezeigt, wie man mit **old** Änderungen an Ausdrücken verbieten kann. Die Situation ist in Programm 16 abstrahiert.

Programm 16

Eiffel: Änderungsverbot
an einzelner Abfrage

```
class interface A
feature
  b : B
  do_it
    ensure
      b = old b
end
```

Diese Möglichkeit funktioniert gut, wenn B ein Werttyp (eiffelisch: expandierter Typ) ist. Ist B aber ein Referenztyp, also b eine Referenz, so bedeutet

$b = \text{old } b$

nur, dass sich b vor und nach dem Aufruf von do_it auf dasselbe Objekt bezieht; es erlaubt durchaus, dass das an b gebundene Objekt seinen Zustand geändert hat, und zwar durch einen Routinenaufruf der Art $b.do_it$. Das Änderungsverbot in Programm 16 deckt also nur die Änderungsart (1) ab.

Klassen haben im Allgemeinen viele Attribute. Viele Routinen bewirken nur an wenigen Attributen Änderungen. Man kann dann in einer Nachbedingung viele Bedingungen obiger Art hinschreiben:

Programm 17

Eiffel: Änderungsverbot
an vielen Abfragen



```
class interface A
feature
  b : B
  c : C
  d : D
  ...
  z : Z
  change_only_b_and_c
    ensure
      d = old d
      ...
      z = old z
end
```

Die in Programm 17 gezeigte Möglichkeit des Aufzählens, welche Attribute sich nicht ändern dürfen, ist unhandlich und änderungsunfreundlich, wenn in A oder einer Erweiterungsklasse weitere Attribute hinzukommen. *Eiffel* bietet deshalb mit **strip**-Ausdrücken eine Möglichkeit des Aufzählens, welche Attribute sich ändern dürfen:

Programm 18

Eiffel: Änderungsverbot
an vielen Abfragen
nach [Mey92a] S. 396–398

```
class interface A
feature
  b : B
  c : C
  d : D
  ...
  z : Z
  change_only_b_and_c
    ensure
      equal(strip(b, c), old strip(b, c))
end
```

Programm 18 ist äquivalent zu Programm 17, aber handlicher. Der ECMA-Standard von *Eiffel* kennt keine **strip**-Ausdrücke mehr, statt dessen bietet er das einfachere **only**-

Konstrukt, das nicht in jeder *Eiffel*-Implementation realisiert ist. Programm 19 ist äquivalent zu Programm 18.

Programm 19

Eiffel: Änderungsverbot
an vielen Abfragen
nach [ECMA367] S.
60–61

```
class interface A
feature
  b : B
  c : C
  d : D
  ...
  z : Z
  change_only_b_and_c
  ensure
    only b, c
end
```

Bertrand Meyer schlägt in seinem Technikblog vor, auch auf **only** zu verzichten und die oben erwähnte pragmatische Regel statisch prüfbar zu machen:

Leitlinie 18

Bertrand Meyer's
„Implicit Framing Rule“
[Mey11]

*„A routine may change the value of a query only if the query is specified in the routine's postcondition [...] a feature is "specified" in a postcondition if it appears there outside of the scope of an **old** expression“ [Mey11].*

Mit dieser Regel genügt es nicht, in der Nachbedingung mit

only b, c

nur zu spezifizieren, dass sich nur *b* und *c* ändern dürfen; statt dessen ist anzugeben, *wie* sich *b* und *c* ändern, etwa wie in Programm 20.

Programm 20

Eiffel: Änderungsverbot
an vielen Abfragen
nach [Mey11]

```
class interface A
feature
  b : B
  c : C
  d : D
  ...
  z : Z
  change_only_b_and_c
  ensure
    some_property(b)
    other_property(c)
end
```

Der Übersetzer kann aus dem Vorkommen von *b* und *c* in der Nachbedingung schließen, dass nur sie sich ändern dürfen und sonst nichts.

Als größeres Beispiel folgt eine Intervall-Klasse. Die meisten der obigen Programme enthalten keinen *Eiffel*-Quelltext, sondern aus Quelltext extrahierte Schnittstellen, erkennbar am Schlüsselwort **interface** nach **class**. Dagegen enthält Programm 21 den Quelltext der abstrakten (**deferred**) Klasse *RANGE*, um bisher vorgestellte Vertragselemente in einem Zusammenhang zu zeigen.

Programm 21

Eiffel: Intervall,
spezifiziert mit
Änderungsverbot nach
[ECMA367] S. 60–61

```

note
  description : "Integer range with a cursor"
deferred class RANGE
feature
  lower,
  upper,
  size,
  cursor : INTEGER

  make (new_lower, new_upper : INTEGER)
    require
      new_lower <= new_upper
    deferred
    ensure
      lower = new_lower
      upper = new_upper
      cursor = lower
    end

  to_lower
    deferred
    ensure
      cursor = lower
      only cursor
    end

  to_upper
    deferred
    ensure
      cursor = upper
      only cursor
    end

  back
    require
      lower < cursor
    deferred
    ensure
      cursor = old cursor - 1
      only cursor
    end

  forth
    require
      cursor < upper
    deferred
    ensure
      cursor = old cursor + 1
      only cursor
    end

invariant
  size = upper - lower + 1
  lower <= cursor
  cursor <= upper

end -- class RANGE

```

3.6.2

Attributobjekte

Für Verbote der Änderungsart (2) bietet *Eiffel* kein Spezifikationsmittel. Sie lassen sich aufwändig mit Mitteln der Implementation so bewerkstelligen:

Programm 22

Eiffel: Änderungsverbot
an einzelmem
Abfrageobjekt



```

class A
feature
  b : B_REF
  do_it
    local
      old_b : like B_REF
    do
      if b /= Void then
        old_b := b.twin
      end
      ...
      check
        postcondition: equal (b, old_b)
      end
      ensure
        b = old_b
      end
    end
end
end

```

In Programm 22 vereinbart die Routine *do_it* eine lokale Variable *old_b* gleichen Referenztyps wie *b* und initialisiert sie mit einer Kopie des an *b* gebundenen Objekts. Lokale Variablen (ausgenommen *Result*) dürfen nicht in Nachbedingungen vorkommen. Deshalb muss vor dem **ensure**-Abschnitt eine normale Zusicherung (**check**) garantieren, dass die an *b* und *old_b* gebundenen Objekte gleich sind. Offenbar taugt diese Änderungsverbotsmöglichkeit nicht für die tägliche Programmierpraxis.

3.6.3

Argumentobjekte

Eine Routine kann nicht nur auf Attribute ihrer Klasse zugreifen, sondern auch auf ihre formalen Argumente. Die einzige Argumentübergabeart ist die Wertübergabe. Argumente sind stets schreibgeschützt, d.h. die Routine kann die Werte ihrer Argumente nicht ändern. Deshalb sind beide Nachbedingungen in Programm 23 redundant.

Programm 23

Eiffel: Stets erfüllte
Nachbedingung



```

class interface A
feature
  do_it (b : B)
    ensure
      b = old b
      equal (b, old b)
    end
end

```

Ist das Argument *b* von einem Referenztyp und bezieht es sich auf ein Objekt, so kann die Routine durch einen Routinenaufruf der Art *b.do_it* den Zustand des an *b* gebundenen Objekts ändern. Die Nachbedingungen in Programm 23 verhindern dies nicht. Auch für Verbote der Änderungsart (3) bietet *Eiffel* kein Spezifikationsmittel. Sie lassen sich aufwändig mit Mitteln der Implementation analog zu Programm 22 so bewerkstelligen:

Programm 24

Eiffel: Änderungsverbot
an einzelner
Argumentobjekt



```

class A
feature
  do_it (b : B_REF)
    local
      old_b : like B_REF
    do
      if b /= Void then
        old_b := b.twin
      end
      ...
      check
        postcondition: equal (b, old_b)
      end
    end
end
end

```

In Programm 24 vereinbart die Routine *do_it* eine lokale Variable *old_b* gleichen Referenztyps wie das Argument *b* und initialisiert sie mit einer Kopie des an *b* gebundenen Objekts. Eine normale Zusicherung garantiert, dass die an *b* und *old_b* gebundenen Objekte gleich sind. Offenbar taugt auch diese Änderungsverbotsmöglichkeit nicht für die tägliche Programmierpraxis.

3.7**Nebeneffektfreiheit**

Nach dem Prinzip der Trennung von Abfragen und Kommandos sind Abfragen neben-effektfrei (*pure*). Lässt sich die Nebeneffektfreiheit spezifizieren? Nach dem über Änderungsverbote Bekannten nur eingeschränkt.

Programm 25

Eiffel: Eingeschränktes
Änderungsverbot für
Funktion

```

class interface A
feature
  b : B
  ...
  z : Z
  function (ab : B; ...; az : Z) : RESULT
    ensure
      equal (strip (), old strip ()) -- [Mey92a] S. 396–398
    only                             -- [ECMA367] S. 60–61
end

```

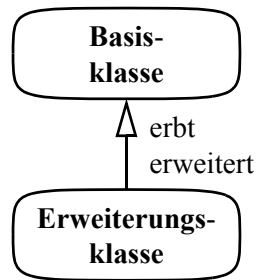
Programm 25 zeigt, wie man in verschiedenen *Eiffel*-Versionen spezifiziert, dass eine Funktion keine Werte von Attributen ihrer Klasse ändert. Schwer auszudrücken ist ein Änderungsverbot für Objekte, auf die sich Attribute oder Argumente beziehen. Es ist kurios, dass *Eiffel*, das auf Nebeneffektfreiheit großen Wert legt, diese Eigenschaft bei Funktionen nicht spezifizieren kann, während C++, das auf dem Konzept neben-effekt-behafteter Operatoren und Ausdrücke beruht, mit dem *const*-Spezifikator eine Möglichkeit bietet, die Nebeneffektfreiheit von Funktionen zu garantieren.

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

3.8**Vererbung**

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

Bild 5
Erweiterungs-
beziehung



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

Abstrakte Basisklassen werden durch **abstrakte Verträge** spezifiziert. Erweiterungsklassen dürfen u.a. **geerbte Verträge** anpassen:

- geerbte Invarianten verstärken,
- Vorbedingungen geerbter Dienste abschwächen (*Eiffel*: **require else**),
- Nachbedingungen geerbter Dienste verstärken (*Eiffel*: **ensure then**),

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 [LiW94].

Leitlinie 19
Erweiterungsklassen
müssen geerbte
Verträge erfüllen

Eine Erweiterungsklasse erbt die Verträge ihrer Basisklassen und darf konsistent mit dem liskovschen Substitutionsprinzip

- Invarianten verstärken,
- Vorbedingungen von Diensten abschwächen,
- Nachbedingungen von Diensten verstärken,
- nicht mehr Ausnahmen auslösen.

3.9

Laufzeitprüfungen

Eiffel-Implementationen bieten folgende Stufen zur Prüfung von Zusicherungen zur Laufzeit; für jede Klasse ist die Stufe zur Übersetzungszeit einstellbar.

- (1) Keinerlei Prüfung.
- (2) Nur Vorbedingungen.
- (3) Vor- und Nachbedingungen.
- (4) Verträge (Vor- und Nachbedingungen und Invarianten).
- (5) Alle Zusicherungen (Verträge und normale Zusicherungen).

Leitlinie 20
Lass prüfen

Schalte zum Testen stets alle Laufzeitprüfungen von Zusicherungen an!

Da Studentenprogramme das Teststadium nicht verlassen, gilt für sie Leitlinie 20 immer. Bei professionellen Programmen im Betrieb kann es Gründe geben, Laufzeitprüfungen abzuschalten. Aber übt man Trockenschwimmen mit einem Schwimmring und schmeißt den Ring vor dem ersten Gang ins Wasser weg? Legt man den Sicherheitsgurt nur beim Probieren von Lenkrad und Schaltknüppel in der Garage an? Baut man die Airbags aus, bevor man auf der Autobahn davon rast?

Selbst wenn der Laufzeitaufwand für Zusicherungsprüfungen eine Rolle spielt, sollten wenigstens die Laufzeitprüfungen der Vorbedingungen angeschaltet bleiben.

Leitlinie 21
Mach Vorbedingungen
billig prüfbar

Sorge dafür, dass Abfragen in Vorbedingungen schnell zu berechnen sind!

3.10

Aufgaben

Aufgabe 11
Intervallklasse
implementieren

Implementieren Sie die vertraglich spezifizierte Klasse *RANGE* in Programm 21! □

Aufgabe 12
Zeichenkettenklasse
spezifizieren

Spezifizieren Sie eine Klasse *STRING* nach der Vertragsmethode mit den Eigenschaften, die Sie von Zeichenketten erwarten! □

4

Einordnendes

4.1

Die Vertragsmethode im Softwareentwicklungsprozess

Wie ordnet sich die Vertragsmethode in den Softwareentwicklungsprozess ein? Wie wirkt sie sich auf die Softwareentwicklung und ihre Ebenen sowie auf das Softwareprodukt und seine Qualitätsmerkmale aus?

- (1) *Analyse*: SdV ist allgemeinverständlich, kundenfreundlich, formalisiert auf hohem Abstraktionsniveau. Die Trennung von Abfragen und Aktionen ersetzt die von datenzentrierten Ansätzen empfohlene Trennung von Daten und Operationen.
- (2) *Entwurf*: SdV garantiert die Integrität von Klassenhierarchien durch vererbte Verträge. SdV unterstützt und präzisiert Entwurfsmuster durch abstrakte Verträge in abstrakten Musterklassen [JTM00].
- (3) *Implementierung*: Verträge sind zur Laufzeit prüfbare Spezifikationen, eingebaute Selbsttests zur Prüfung der Korrektheit von Implementationen.
- (4) *Test*: SdV beseitigt Mehrdeutigkeiten, reduziert den Testaufwand und ermöglicht automatisierte effektive Testverfahren.
- (5) *Dokumentation*: SdV integriert Code und Dokumentation und vermeidet so Inkonsistenzen. Verträge sind wesentlicher Teil der Dokumentation.
- (6) *Betrieb*: Laufzeitprüfungen erhöhen die Vertrauenswürdigkeit von Implementationen und Spezifikationen. Einstellbarer Prüfgrad liefert nach Bedarf Sicherheit durch demonstrierbare Korrektheit oder maximale Effizienz.
- (7) *Wartung und Pflege*: SdV unterstützt Wartbarkeit und Erweiterbarkeit durch integrierte konsistente Dokumentation.
- (8) *Iterative, nahtlose, reversible Entwicklung*: SdV ist nahtlos anwendbar, vermeidet Brüche zwischen Spezifikation und Implementation.

SdV unterstützt die Softwarekonstruktion als ganzheitlichen, stetigen, iterativen Prozess [WaN95]. 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 [JeM97], [MMS98];
- **struktureller Qualitätsmerkmale** wie Wartbarkeit, Änderbarkeit und Wiederverwendbarkeit [Mey94], [Mik99].

4.2

Die Vertragsmethode verglichen mit anderen Ansätzen

Wie verhält sich die Vertragsmethode zu anderen Ansätzen?

- (1) *Ausnahmebehandlung (exception handling)*:
 - ◆ Kein Mittel der Spezifikation, sondern der Implementation; daher nicht mit SdV vergleichbar.
 - ◆ Kann guten Zwecken dienen, wenn bei unerwünschten Situationen das Erkennen und das Beheben im Programmablauf weit auseinander liegen.
 - ⊗ Es gibt kaum Methoden zum sinnvollen Einsatz von A.
 - ⊗ A. widerspricht den Regeln strukturierten Programmierens. Undisziplinierte Verwendung von A. kann Probleme verschärfen, zu deren Lösung A. beitragen sollte.
 - ⊗ A. wird leider oft in Situationen missbraucht, die besser mit normaler Ablaufsteuerung zu behandeln sind.
- (2) *Zusicherungen (assertion)*:
 - ☺ Besser als gar nichts!
 - ⊗ Keine Differenzierung nach Art.
 - ⊗ Nicht Teil der Dokumentation.
 - ⊗ Nicht für Kunden sichtbar.
 - ⊗ Nicht mit Vererbung integriert.
- (3) *Defensives Programmieren*: D.P. legt keine Verantwortlichkeiten zwischen Komponenten fest; Komponenten misstrauen sich gegenseitig. Das führt zu unnötiger Redundanz und Komplexität. SdV vermeidet Redundanz und reduziert die Komplexität.
- (4) *Formale Korrektheitsbeweise*: SdV ist softwarepraktisch orientiert, nicht auf separate Beweise. Ein Ziel ist jedoch, K. unterstützt durch SdV zu führen.
- (5) *Technische Reviews* sind ohne Spezifikationen sinnlos, werden durch SdV effektiver.
- (6) *Statische Analysewerkzeuge* ermöglichen statische Analyse von Vertragsspezifikationen.
- (7) *Tests* sind ohne Spezifikationen sinnlos, liefern mit SdV höhere Fehlererkennungsraten.
- (8) *Testgetriebene Softwareentwicklung*: T.S. steckt die Spezifikation in die Testtreiber, getrennt von der Implementation. SdV folgt dem Prinzip des einzigen Dokuments: Spezifikation und Implementation zusammen an einer Stelle.

4.3

Die Vertragsmethode umgesetzt in anderen Sprachen

Wie unterstützen verbreitete Spezifikations- und Implementationssprachen, Notationen und Werkzeuge die Vertragsmethode?

- (1) *Eiffel*: SdV pur.

- (2) Andere Sprachen mit SdV: *Sather, D, Spec#*.
- (3) *Component Pascal*: Zusicherungen + Laufzeitinfo durch Post-Mortem-Debugger.
- (4) C++: Zusicherungsmakro *assert (assert.h)*, Ausnahmebehandlung; Erweiterungen: *Assertions.h, GNU Nana, A++ (Annotated C++)*, *Larch*,...
- (5) *Java*: Zusicherungen, Ausnahmebehandlung; Erweiterungen: *iContract, jContractor, Handshake, jassert, JASS, JMScript*,...
- (6) *C#*: Zusicherungen, Ausnahmebehandlung; Spracherweiterung: *Spec#*.
- (7) *IDL* (CORBA, Microsoft COM): keine Unterstützung für SdV.
- (8) *BON* (*Business Object Notation*): SdV + Prädikatenlogik.
- (9) *OCL* (*Object Constraint Language* von *UML*): SdV.

A

Literatur

- [BMR93] Alex Borgida, John Mylopoulos, Raymond Reiter: “...And Nothing Else Changes”: *The Frame Problem in Procedure Specifications*. ICSE '93 Proceedings of the 15th international conference on Software Engineering, Baltimore (May 1993) S. 303–314; <http://dl.acm.org/citation.cfm?id=257636> (Zugriff 2012-10-08)
- [BMR95] Alex Borgida, John Mylopoulos, Raymond Reiter: *On the Frame Problem in Procedure Specifications*. IEEE Transactions on Software Engineering, Vol. 21, No. 10 (October 1995) S. 785–798; http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=469460 (Zugriff 2012-10-08)
- [ECMA367] Standard ECMA-367: *Eiffel Analysis, Design and Programming Language*. (June 2006) 2nd Edition, 174 S.; <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf> (Zugriff 2012-10-08)
- [Fow99] Martin Fowler (with contributions by Kent Beck, John Brant, William Opdyke, Dan Roberts): *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, Reading (1999) 431 S.
- [Hoa69] C. A. R. Hoare: *An Axiomatic Basis for Computer Programming*. Communications of the ACM, Vol. 12, No. 10 (October 1969) S. 576–583; <http://dl.acm.org/citation.cfm?id=363259> (Zugriff 2012-10-08)
- [Hug01] Karlheinz Hug: *Module, Klassen, Verträge. Ein Lehrbuch zur komponentenorientierten Softwarekonstruktion*. Vieweg, Braunschweig, Wiesbaden (2001) 2. Aufl., 446 S.
- [HugI12] Karlheinz Hug: *Lehrmaterial zu Informatik 1 + 2 für mki-B*. Hochschule Reutlingen (2008) http://informatik.karlheinz-hug.de/Lehrmaterial/Informatik_1+2/, ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrmaterial/Informatik_1+2/
- [JeM97] Jean-Marc Jézéquel, Bertrand Meyer: *Design by Contract: The Lessons of Ariane*. IEEE Computer, Vol. 30, No. 1 (January 1997) S. 129–130; <http://eiffel.com/doc/manuals/technology/contract/ariane/page.html> (Zugriff 2012-10-08)
- [JTM00] Jean-Marc Jézéquel, Michel Train, Christine Mingins: *Design Patterns and Contracts*. Addison-Wesley, Reading (2000) 348 S.
- [LiW94] Barbara H. Liskov, Jeannette M. Wing: *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, Vol. 16, No. 6 (November 1994) S. 1811–1841; <http://dl.acm.org/citation.cfm?id=197383> (Zugriff 2012-10-08)

- [McK96] James C. McKim, Jr.: *Programming by contract: Designing for Correctness*. Journal of Object-Oriented Programming (May 1996) S. 70–74
- [McK99] James C. McKim, Jr.: *Advanced Programming By Contract: Designing for Correctness*. TOOLS Pacific, Melbourne (1999) tutorial, 47 S.; http://faculty.winthrop.edu/mckimj/tools_pacific_99.pdf (Zugriff 2012-10-08)
- [Mey92a] Bertrand Meyer: *Eiffel: The Language*. Prentice Hall International, Hertfordshire (1992) 594 S.
- [Mey92b] Bertrand Meyer: *Applying „Design by Contract“*. IEEE Computer, Vol. 25, No. 10 (October 1992) S. 40–51
- [Mey94] Bertrand Meyer: *Reusable Software. The Base Object-Oriented Component Libraries*. Prentice Hall International, Hertfordshire (1994) 514 S.
- [Mey95] Bertrand Meyer: *Eiffel: The Reference*. ISE Technical Report TR-EI-41/ER (1995) Version 3.3.4, 100 S.
- [Mey97] Bertrand Meyer: *Object-oriented Software Construction*. Prentice Hall PTR, Upper Saddle River (1997) 2nd edition, 1260 S.
- [Mey00] Bertrand Meyer: *Toward More Expressive Contracts*. Journal of Object-Oriented Programming (July/August 2000) S. 39–43
- [Mey05a] Bertrand Meyer: *Object-Oriented Software Construction. Lecture 11: Design By ContractTM*. ETH Zürich (Summer Semester 2005) 44 Folien; http://se.inf.ethz.ch/old/teaching/ss2005/0250/lectures/oosc_11_dbc_1up.pdf (Zugriff 2012-10-08)
- [Mey05b] Bertrand Meyer: *Object-Oriented Software Construction. Lecture 12: Design By ContractTM*. ETH Zürich (Summer Semester 2005) 30 Folien; http://se.inf.ethz.ch/old/teaching/ss2005/0250/lectures/oosc_12_dbc_1up.pdf (Zugriff 2012-10-08)
- [Mey05c] Bertrand Meyer: *Object-Oriented Software Construction. Lecture 13: Design By ContractTM*. ETH Zürich (Summer Semester 2005) 33 Folien; http://se.inf.ethz.ch/old/teaching/ss2005/0250/lectures/oosc_13_dbc_1up.pdf (Zugriff 2012-10-08)
- [Mey11] Bertrand Meyer's technology blog: *If I'm not pure, at least my functions are*. (4 July 2011) <http://bertrandmeyer.com/tag/framing/> (Zugriff 2012-10-08)
- [Mik99] 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) S. 76–85; <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=779001> (Zugriff 2012-10-08)
- [MiM99] Richard Mitchell, James McKim: *Extending a method of devising software contracts*. In: C. Mingins, B. Meyer (ed): Proc. TOOLS 32 Melbourne, IEEE (November 1999) S. 234–251; <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=809429> (Zugriff 2012-10-08)
- [MiM01] Richard Mitchell, Jim McKim: *Design by Contract, by Example*. Addison-Wesley, Reading (2001) 256 S.
- [Mit00] Richard Mitchell: *Fulfilling a contract. Satisfying a precondition*. Journal of Object-Oriented Programming (May 2000) S. 34–37

- [MMS98] Bertrand Meyer, Christine Mingins, Heinz Schmidt: *Trusted Components for the software industry*. IEEE Computer (May 1998) S. 104–105; <http://se.ethz.ch/~meyer/publications/computer/trusted.pdf>; long variant: <http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer/trusted/page.html> (Zugriffe 2012-10-08)
- [Pet62] Carl Adam Petri: *Kommunikation mit Automaten*. Dissertation, TH Darmstadt, Uni Bonn (1962)
- [WaN95] Kim Waldén, Jean-Marc Nerson: *Seamless Object-Oriented Software Architecture. Analysis and Design of Reliable Systems*. Prentice Hall, New York (1995) 438 S.