
Vorbedingung, Fehlanzeige oder Ausnahme?

*Ist es machbar, so mach's!
Und wenn's nicht machbar ist, was dann?*

Schlagwörter	Spezifikation durch Vertrag, Vorbedingung, Fehler, Misserfolg, Anzeige, globale Variable, Ausgabeparameter, Ausnahmebehandlung
Motivation	Die Konversion einer Zeichenkette in eine Zahl lösen fünf Programmiersprachen mit fünf Programmierschemas, dargestellt in <i>Informatik 3 Skript Teil 1 Konzepte und Entwürfe objektorientierter Programmiersprachen, Kapitel 2 Einmal quer durch die Sprachen</i> . Das „Problem“ daran ist der „schlechte Fall“, dass die Zeichenkette keine Zahl enthält. Das Öffnen einer Datei gegebenen Namens lösen Bibliotheksroutinen mit verschiedenen Programmierschemas. Das „Problem“ daran ist der „schlechte Fall“, dass die Datei nicht existiert. Lösungen müssen etwas für „schlechte Fälle“ vorsehen.
Lernziele	Der Leser lernt Programmierschemas für bedingtes Ausführen einer Aufgabe kennen, um sie bewerten und in Programmierpraxis und I3-Prüfung anwenden zu können.

Inhalt

1	Einleitung	1
2	Programmierschema mit Vorbedingung	2
3	Programmierschema mit globaler Variable	4
4	Programmierschema mit Ausgabeparameter	4
5	Programmierschema mit Ausnahme	5
6	Programmierschema mit Abfrage und Ausnahme	6
7	Ausleitung	8

1

Einleitung

Eine häufige Situation beim Programmieren ist: Eine Routine (Prozedur oder Funktion) kann ihre Aufgabe nur unter einer bestimmten Bedingung ausführen. Was soll geschehen, wenn diese Bedingung nicht erfüllt ist? Zwei prinzipiell verschiedene Ansätze für das Protokoll zwischen der Routine und ihrem Aufrufer führen zu verschiedenen Programmierschemas:

- Beim **restriktiven Ansatz** ist der Aufruf der Routine nur erlaubt, wenn die Bedingung für das Ausführen der Aufgabe erfüllt ist. Die Bedingung wird zur Vorbedingung der Routine erklärt. Jeder erlaubte Routinenaufruf ist erfolgreich.
- Beim **liberalen Ansatz** ist der Aufruf der Routine stets erlaubt. Die Routine informiert ihren Aufrufer darüber, ob die Bedingung für das Ausführen erfüllt war und die Routine die Aufgabe ausgeführt hat (Erfolg) oder nicht (Misserfolg). Für die Kommunikation von der Routine zu ihrem Aufrufer gibt es mindestens drei Programmierschemas:

- (a) Eine **globale Variable** wird von der Routine gesetzt und muss vom Aufrufer abgefragt werden.
- (b) Ein **Ausgabeparameter** der Routine muss vom Aufrufer ausgewertet werden.
- (c) Eine **Ausnahme** (*exception*) wird von der Routine ausgelöst (geworfen) und muss vom Aufrufer behandelt (aufgefangen) werden.

Bei (a) und (b) sollte die Routine eine Prozedur sein, denn wenn sie eine Funktion ist, bewirkt die Kommunikation über eine globale Variable oder einen Ausgabeparameter einen Nebeneffekt, den es zu vermeiden gilt. Nebeneffektbehaftete Funktionen setzen wir nur selten aus guten Gründen bei schwachen Alternativen ein.

Spezifikation

Welche Prinzipien unterscheiden die beiden Ansätze? Der restriktive Ansatz löst das Problem auf der Ebene der Spezifikation mit der Methode Spezifikation durch Vertrag. Vorbedingungen sind ein Mittel der Spezifikation, ein Mittel der Vertragsmethode. Die Vorbedingung einer Routine drückt aus, unter welcher Bedingung sie von einer anderen Routine aufgerufen werden darf, also wann ein Aufruf korrekt ist. Damit ist auch gesagt, was Vorbedingungen *nicht* sind:

- Vorbedingungen sind *kein* Mittel der Implementation! Alle Vorbedingungen, ja alle Zusicherungen in einem korrekten Programm zu streichen oder ihre Laufzeitprüfung abzuschalten darf das Verhalten keines Programmablaufs ändern.
- Deshalb sind Vorbedingungen auch *kein* Mittel, um Eingaben von Benutzern zu prüfen! Ein Benutzer darf alles eingeben, was ihm möglich ist. Keine Vorbedingung kann ihm eine Eingabe verbieten. Die Implementation muss prüfen und entscheiden, welche Benutzereingaben zu welchen Reaktionen und Meldungen führen.

Implementation

Der liberale Ansatz löst das Problem auf der Ebene der Implementation mit Mitteln der Kommunikation und der Ablaufsteuerung. Globale Variablen, Ausgabeparameter, Ausnahmen, `if-else`- und `try-catch`-Anweisungen sind Mittel der Implementation. Damit ist auch gesagt, was sie *nicht* sind: Sie sind *keine* Mittel der Spezifikation! Variablen, Parameter, Ausnahmen, `else`- oder `catch`-Zweige aus einem Programm zu streichen ändert i.A. das Verhalten von Programmabläufen.

Das Folgende beschreibt fünf Programmierschemas an einem abstrakten Beispiel mit zwei Klassen: einer Lieferantenklasse `Something` mit einer Prozedur `doIt` für die bedingt auszuführende Aufgabe und einer Kundenklasse `ClientOfSomething`, die die Aufgabe ausgeführt haben will und dazu `doIt` aufruft. Vorausgesetzt ist stets, dass die Bedingung für das Ausführen der Aufgabe korrekt spezifiziert und implementiert ist. Als Programmiersprache dient javaischer Pseudocode. Spezifikationsteile erscheinen **blau**, Misserfolgsbehandlungen **rot**, informale Teile *kursiv*.

2

Programmierschema mit Vorbedingung

Programm 1

Pseudocode: Lieferant stellt Vorbedingung

```
class Something1 {
    // Can work be done?
    public boolean doable () {...}

    // Effect: work is done.
    public void doIt () {
        assert doable() : "precondition";
        do the work
    }
}
```

Der Lieferant muss die Vorbedingung für das Ausführen der Aufgabe für seine Kunden prüfbar machen; Programm 1 sieht dafür die Abfrage `doable` vor. Da `doIt` die Bedingung zugesichert ist, kann es ohne Fallunterscheidung die Aufgabe ausführen.

Programm 2

Pseudocode: Kunde
garantiert
Vorbedingung

```
class ClientOfSomething1 {
    private Something1 something = new Something1();
    public void doSomeCall () {
        ...
        if (something.doable()) {
            something.doIt();
        } else {
            handle undoable case
        }
        ...
    }
    public void doOtherCall () {
        ...
        assert something.doable() : "invariant";
        something.doIt();
        ...
    }
}
```

Beim Kunden sind zwei Situationen möglich; Programm 2 zeigt sie in zwei Prozeduren: `doSomeCall` prüft die Vorbedingung des Lieferanten und verzweigt, falls sie nicht erfüllt ist, zur Misserfolgsbehandlung. Dieses Programmierschema nennen wir **Abfrage-Aktion-Schema**. In `doOtherCall` garantiert der Kontext, dass die Bedingung erfüllt ist, deshalb können das Prüfen der Abfrage und die Alternative wegfallen. Fazit: Eine Fallunterscheidung beim Kunden genügt.

Auf S. 2 wird vorausgesetzt, dass korrekt spezifiziert *und* implementiert wird. Aber weil die Vertragsmethode nicht nur dem Spezifizieren, sondern auch dem Testen dient, zeigt Tabelle 1 die vier Fälle, die bei korrekter Spezifikation auftreten können.

Tabelle 1

Spezifikation korrekt

		Laufzeitprüfung der Zusicherungen	
		aus	an
Vorbedingung korrekt formuliert	wenigstens einmal nicht erfüllt = Implementation fehlerhaft	Verhalten nicht spezifiziert, seltsame Fehlersymptome möglich	Programmablauf stoppt
	stets erfüllt = Implementation korrekt	kein Effekt auf Verhalten	

Gegen den restriktiven Ansatz mit der Vertragsmethode könnten Kritiker einwenden, dass Programmierern Spezifikationsfehler unterlaufen können und dagegen der Verzicht auf Spezifikation helfe: Wer nicht spezifiziert, kann keine Spezifikationsfehler machen. Richtig, aber nicht stichhaltig! Denn wer nicht spezifiziert, muss doch implementieren. Wer die Bedingung zum Ausführen der Aufgabe beim Spezifizieren nicht korrekt formulieren kann, der kann sie auch beim Implementieren nicht korrekt erfassen. Die Fehler, die er durch Nichtspezifizieren vermeidet, begeht er beim Implementieren. Schwache Programmierer schreiben bei jedem Programmierschema fehlerhafte, sich seltsam verhaltende Programme. Alle denkbaren Programmierfehler bei den Programmierschemas zu diskutieren würde dieses Dokument aufblähen.

Vertragskritiker sollten wissen: Vor- und Nachbedingungen und Invarianten existieren unabhängig davon, ob sie dokumentiert und in Zusicherungen ausgedrückt sind oder nicht. Ein Programm, bei dem eine Bedingung in einem Ablauf falsch ist, obwohl sie wahr sein sollte, ist fehlerhaft – selbst dann, wenn der Programmierer nirgends einen

Kommentar zu dieser Bedingung hinterlassen hat. Ein 300 Millionen Euro teures Beispiel war der Absturz der Ariane-5-Rakete wegen einer nicht dokumentierten Bedingung (siehe [Krimskrams – Softwarequalität](#), S. 2).

3

Programmierschema mit globaler Variable

Programm 3

Pseudocode: Lieferant
setzt Anzeige

```
class Something2 {
    private boolean done;
    // Did tryDoIt succeed to do the work?
    public boolean done () {return done;}
    // Effect: If work is accomplished, then done, else not done.
    public void tryDoIt () {
        if (doable) {
            do the work
            done = true;
        } else {
            done = false;
        }
    }
}
```

In Programm 3 muss tryDoIt die private Bedingung *doable* prüfen: Ist sie erfüllt, kann es die Aufgabe ausführen und die globale Variable done, **Anzeige (flag)** genannt, auf true setzen, sonst muss es done auf false setzen. Eine Getter-Funktion erlaubt Kunden Lesezugriffe auf die Anzeige.

Programm 4

Pseudocode: Kunde
liest Anzeige

```
class ClientOfSomething2 {
    private Something2 something = new Something2();
    public void doSomeCall () {
        ...
        something.tryDoIt();
        if (!something.done()) {
            handle undoable case
        }
        ...
    }
}
```

Im Kundenprogramm 4 ist eine Situation möglich: doSomeCall ruft tryDoIt auf, prüft dann done und verzweigt ggf. zur Misserfolgsbehandlung. Dieses Programmierschema nennen wir **Aktion-Anzeige-** oder **Aktion-Abfrage-Schema**. Eine Schwäche dabei ist: Der Programmierer kann die Prüfung nach dem Aufruf von tryDoIt vergessen; dieser Fehler wird u.U. beim Testen schwer entdeckt. Fazit: Lieferanten- und kundenseitig je eine Auswahlanweisung genügt.

4

Programmierschema mit Ausgabeparameter

Das **Aktion-Ausgabeparameter-Schema** genannte Muster ähnelt dem Aktion-Anzeige-Schema, nur die Anzeige ist durch einen Ausgabeparameter ersetzt.

Programm 5

Pseudocode: Lieferant
setzt
Ausgabeparameter

```
class Something3 {
    // Effect: If work is accomplished, then done, else not done.
    public void tryDoIt (out boolean done) {
        if (doable) {
            do the work
            done = true;
        } else {
            done = false;
        }
    }
}
```

In Programm 5 muss tryDoIt die private Bedingung *doable* prüfen: Ist sie erfüllt, kann es die Aufgabe ausführen und den Ausgabeparameter done auf true setzen, sonst muss es done auf false setzen.

Programm 6

Pseudocode: Kunde
liest
Ausgabeparameter

```
class ClientOfSomething3 {
    private Something3 something = new Something3();
    public void doSomeCall () {
        boolean done;
        ...
        something.tryDoIt(out done);
        if (!done) {
            handle undoable case
        }
        ...
    }
}
```

Im Kundenprogramm 6 ist eine Situation möglich: doSomeCall ruft tryDoIt auf, prüft dann den aktuellen Ausgabeparameter done und verzweigt ggf. zur Misserfolgsbehandlung. Die Schwäche von Programm 4 ist weniger fehleranfällig geworden: Der Programmierer kann zwar das Prüfen des Ausgabeparameters vergessen, er muss aber zumindest eine lokale Variable dafür vereinbaren. Die Lieferantenschnittstelle ist „aus einem Guss“. Fazit: Lieferanten- und kundenseitig je eine Auswahlanweisung genügt.

5**Programmierschema mit Ausnahme****Programm 7**

Pseudocode: Lieferant
löst Ausnahme aus

```
class Something4 {
    // Effect: work is done or an UndoableException is thrown.
    public void tryDoIt () throws UndoableException {
        if (doable) {
            do the work
        } else {
            throw new UndoableException("undoable");
        }
    }
}
```

In Programm 7 muss tryDoIt die private Bedingung *doable* prüfen: Ist sie erfüllt, kann es die Aufgabe ausführen, sonst muss es eine Ausnahme auslösen. Der Lieferant bietet dem Kunden keine Information über die Bedingung, unter der der Lieferant die Ausnahme wirft.

Programm 8

Pseudocode: Kunde
behandelt Ausnahme

```
class ClientOfSomething4 {
    private Something4 something = new Something4();
    public void doSomeCall () {
        ...
        try {
            something.tryDoIt();
        } catch (UndoableException e) {
            handle undoable case
        }
        ...
    }
    public void doOtherCall () throws UndoableException {
        ...
        something.tryDoIt();
        ...
    }
    public void doIndirectCall () {
        ...
        try {
            doOtherCall();
        } catch (UndoableException e) {
            handle undoable case
        }
        ...
    }
}
```

Im Kundenprogramm 8 sind zwei Situationen möglich: `doSomeCall` ruft `tryDoIt` direkt auf und behandelt selbst die mögliche Ausnahme. `doOtherCall` ruft `tryDoIt` auch direkt auf, behandelt aber die mögliche Ausnahme nicht, sondern gibt sie an seinen Aufrufer weiter. Am Anfang der Aufrufkette steht `doIndirectCall`, das `tryDoIt` indirekt aufruft und dessen Ausnahme behandelt. Dieses Programmierschema nennen wir **Aktion-Ausnahme-Schema**. Fazit: Beim Lieferanten eine zweiseitige Auswahl, beim Kunden ein try-catch-Block genügen.

Ein Vorteil des Ausnahmekonzepts liegt darin, dass Ausnahmen weit von ihrer Auslösung entfernt behandelt werden können. Dieser Vorteil verschwindet, wenn eine Ausnahme schon vom Aufrufer ihres Auslösers behandelt wird, weil für diese Situation auslöserseitig eine if-else-Fallunterscheidung, behandlerseitig eine try-catch-Fallunterscheidung zu programmieren ist.

6**Programmierschema mit Abfrage und Ausnahme**

Ein Nachteil der Schnittstelle des Lieferantenprogramms 7 ist, dass der Kunde nichts über die Bedingung erfährt, unter der `tryDoIt` eine `undoableException` auslöst, und deshalb das Auslösen der Ausnahme nicht vermeiden kann. Um dem abzuhelpen, kann der Lieferant eine Abfrage `doable` wie in Programm 1 bereitstellen. Diesen Ansatz verfolgt z.B. die Java-Standardbibliotheksklasse `Stack`:¹

Java-Stack

```
public class Stack<E>
    public boolean empty()
    public E peek()
    Throws: EmptyStackException - if this stack is empty.
```

Mit dem **Abfrage-Aktion-Ausnahme-Schema** erhält das abstrakte Beispiel diese Form:

¹ <http://download.oracle.com/javase/6/docs/api/java/util/Stack.html> (Zugriff 2010-11-30)

Programm 9

Pseudocode: Lieferant
bietet Abfrage und löst
Ausnahme aus

```
class Something5 {
    // Can work be done without UndoableException?
    public boolean doable () {...}

    // Effect: If doable, work is done,
    //          else an UndoableException is thrown.
    public void tryDoIt () throws UndoableException {
        if (doable()) {
            do the work
        } else {
            throw new UndoableException("undoable");
        }
    }
}
```

In Programm 9 muss tryDoIt trotzdem die Bedingung doable prüfen und ggf. eine Ausnahme auslösen.

Programm 10

Pseudocode: Kunde
prüft Bedingung und
behandelt Ausnahme
mit Zusicherung

```
class ClientOfSomething5 {
    private Something5 something = new Something5();

    public void doSomeCall () {
        ...
        if (something.doable()) {
            try {
                something.tryDoIt();
            } catch (UndoableException e) {
                assert false : "unreachable branch";
            }
        } else {
            handle undoable case
        }
        ...
    }

    public void doOtherCall () throws UndoableException {
        ...
        if (something.doable()) {
            something.tryDoIt();
        } else {
            handle undoable case
        }
        ...
    }

    public void doIndirectCall () {
        ...
        try {
            doOtherCall();
        } catch (UndoableException e) {
            assert false : "unreachable branch";
        }
        ...
    }
}
```

In Programm 10 muss doSomeCall trotz der von Programm 1 übernommenen if-else-Fallunterscheidung die try-catch-Fallunterscheidung von Programm 8 vorsehen, kann aber im catch-Zweig zusichern, dass er nie erreicht wird. doOtherCall übernimmt die if-else-Fallunterscheidung von Programm 1 und die throws-Klausel von Programm 8, obwohl schon feststeht, dass es die UndoableException nicht wirft. doIndirectCall muss die try-catch-Fallunterscheidung von Programm 8 vorsehen und kann im catch-Zweig zusichern, dass er nie erreicht wird. Fazit: Das Doppelmop-

pel-Abfrage-Ausnahme-Schema kompliziert das Protokoll, statt es zu vereinfachen – eine eklektische Stilmischung ohne klare Vorzüge.

7

Ausleitung

Die vorgestellten Programmierschemas eignen sich zusammen mit den aus dem *Informatik 3 Skript* bekannten Schemas hervorragend dafür, dass der Student sie in Programmierübungen ausprobiert und variiert und der Dozent sie zum Gegenstand von Prüfungsaufgaben macht. Welche Varianten sind denkbar? Beispielsweise diese:

- Lieferant und Kunde könnten Module statt Klassen, d.h. der Programmierstil könnte modular statt objektorientiert sein.
- Die Prozeduren `doIt` und `tryDoIt` könnten Funktionen, d.h. der Programmierstil könnte mehr funktional als prozedural sein.
- Die Prozeduren oder Funktionen könnten Eingabe-, die Prozeduren auch Ausgabe-parameter haben. Den Schemas fehlen Parameter der Übersichtlichkeit halber.
- An jede konkrete Anwendungssituation sind alle Namen softwaretechnisch qualitativvoll anzupassen, z.B. nach *LS Leitlinien zum Softwareentwurf*, Leitlinien 16, 33, 34.