Kapitel 4
Methoden

Dieses Kapitel behandelt Methoden. Wir schauen uns das Design, den Inhalt und den Kontrollfluss von Methoden an, und untersuchen, wie Sie Methoden am besten aufrufen können.

Methoden sind die wichtigsten Container für ausführbaren Code in ABAP. ABAP kennt zwar noch mehr Container für ausführbaren Code, wie z. B. Funktionsbausteine, Programme und Formroutinen, wir haben jedoch bereits in Abschnitt 2.3, »Objektorientierte vs. prozedurale Programmierung«, erläutert, warum Methoden die bessere Wahl sind.

Viele der folgenden Kapitel geben Ihnen detaillierte Empfehlungen für Best Practices innerhalb von Methoden. In diesem Kapitel widmen wir uns zunächst den Methoden als Ganzes, also dem Design, der Definition, der Verwendung, dem Kontrollfluss und dem Aufruf von Methoden.

Wir beginnen in Abschnitt 4.1, »Objektorientierte Programmierung«, mit einer detaillierteren Untersuchung von Methoden in der objektorientierten Programmierung an sich. Abschnitt 4.2, »Parameter«, widmet sich den Parametern von Methoden. In Abschnitt 4.3, »Methodeninhalt«, untersuchen wir, wie Sie den Inhalt von Methoden sauber gestalten können. Abschnitt 4.4, »Methoden aufrufen«, schließt das Kapitel ab, in dem wir Ihnen zeigen, wie Sie Methoden am besten aufrufen können.

4.1 Objektorientierte Programmierung

In diesem Abschnitt ordnen wir Methoden in das objektorientierte Design ein und schließen damit an die Beschreibung von Klassen und Interfaces in Abschnitt 3.1, »Objektorientierung«, an.

4.1.1 Statische Methoden und Instanzmethoden

Statische Methoden werden mit dem Schlüsselwort `CLASS-METHODS` definiert und hängen an der Klasse selbst. Um eine statische Methode aufzurufen, benötigen Sie nur den Namen der Klasse, ohne dass Sie die Klasse vorher instanziieren müssen. Statische Methoden sind also nicht wirklich objektorientiert – es sind ja gar keine Objekte,

also Klasseninstanzen, im Spiel. Sie sind auch kein richtiger Teil der Klassenhierarchie und stehen dadurch abseits von Aspekten wie Vererbung, Redefinition und dynamischer Bindung (siehe Abschnitt 3.2, »Geltungsbereich und Sichtbarkeit«). Listing 4.1 zeigt, wie eine statische Methode angelegt wird.

```
CLASS blog_post DEFINITION
  PUBLIC.

PUBLIC SECTION.

  CLASS-METHODS:
    publish.

ENDCLASS.
```

Listing 4.1 Definition einer statischen Methode

Diese Methode können Sie aufrufen, indem Sie den Namen der Klasse nennen, gefolgt von einem Pfeil mit Doppelstrich (=>):

```
blog_post=>publish( ).
```

Durch ihre Position abseits der Objektorientierung sind statische Methoden ziemlich unflexibel. Eine Instanzmethode hingegen hängt direkt an einer Instanz einer Klasse. Sie werden mit dem Schlüsselwort `METHODS` definiert, wie in Listing 4.2 gezeigt.

```
CLASS blog_post DEFINITION
  PUBLIC.

PUBLIC SECTION.

  METHODS:
    publish.

ENDCLASS.
```

Listing 4.2 Definition einer Instanzmethode

Um eine Instanzmethode aufrufen zu können, benötigen Sie zunächst eine Instanz der Klasse. Der Aufruf erfolgt dann über die Referenzvariable, die auf diese Instanz zeigt, gefolgt von einem Pfeil mit einfachem Strich (->):

```
DATA my_blog_post TYPE REF TO blog_post.
...
my_blog_post->publish( ).
```


Benutzen Sie vorrangig Instanzmethoden

Dadurch, dass Instanzmethoden an Instanzen hängen, statt an den Klassen selbst, sind sie um ein Vielfaches flexibler als statische Methoden. Eine Referenzvariable kann zur Laufzeit nicht nur auf die Klasse deuten, mit der sie typisiert ist, sondern auch auf jede ihrer Unterklassen. Der Aufruf einer Methode über eine Referenzvariable wird dadurch *dynamisch* oder auch *virtuell*.

Instanzmethoden können dadurch redefiniert und in Unit Tests durch Test-Doubles ersetzt werden. Ihr Gültigkeitsbereich ist die Instanz, sodass Sie die Ressourcen, mit denen die Methode arbeitet, in jeder Instanz anders ausgestalten können. Aus diesen Gründen empfehlen wir grundsätzlich, eher Instanzmethoden als statische Methoden zu verwenden. Weitere Details finden Sie im Kapitel über Klassen in Abschnitt 3.1.2, »Klassen und Objekte«.

Zwei Arten von Methoden sollten hingegen eher statisch sein: statische Erzeugungsmethoden und Utility-Methoden.

Statische Erzeugungsmethoden dienen als Alternativen für Konstruktoren. Sie haben sie in Abschnitt 3.3.3, »Statische Erzeugungsmethoden«, bereits genauer kennengelernt. Diese Methoden liefern eine Instanz ihrer Klasse oder einer ihrer Unterklassen, wie z. B. in Listing 4.3 zu sehen, wo die Methode `create_as_copy` eine Instanz erzeugt, indem sie eine andere, bereits existierende Instanz, die ihr im Eingabeparameter `source_blog_post` übergeben wird, inhaltlich dupliziert. Methoden sollten in zwei Fällen statisch sein: wenn sie statische Erzeugungsmethoden sind und wenn sie Utility-Methoden sind.

```
CLASS blog_post DEFINITION
  PUBLIC.

PUBLIC SECTION.

  CLASS-METHODS:
    create_as_copy
      IMPORTING
        source_blog_post TYPE REF TO blog_post
      RETURNING
        VALUE(result) TYPE REF TO blog_post.

  ...

ENDCLASS.
```

Listing 4.3 Beispiel für eine statische Erzeugungsmethode

Utility-Methoden sind Methoden, die nicht von anderen Ressourcen abhängen. Sie führen die immer gleiche statische Operation aus. Diese Operation wird ausschließlich durch die Eingabeparameter der Methode definiert und wird von keinen anderen Parametern beeinflusst. Diese Methoden finden Sie typischerweise in Utility-Klassen, wie wir sie in Abschnitt 3.1.2, »Klassen und Objekte« beschreiben. Ein einfaches Beispiel stellt die Methode `fahrenheit_to_celsius` in Listing 4.4 dar, die eine Temperaturangabe in eine andere Maßeinheit umrechnet.

```
CLASS temperature_conversion DEFINITION
  PUBLIC
  ABSTRACT
  FINAL.

PUBLIC SECTION.

  CLASS-METHODS fahrenheit_to_celsius
    IMPORTING
      temperature_in_fahrenheit TYPE temperature
    RETURNING
      VALUE(result) TYPE temperature.

  ...

ENDCLASS.
```

Listing 4.4 Beispiel für eine Utility-Methode

Statische Methoden und Instanzmethoden werden gleich implementiert. Die Schlüsselwörter `METHOD` und `ENDMETHOD` stehen dabei im `DEFINITION`-Bereich der Klasse und umschließen den Inhalt der Methode. Listing 4.5 zeigt die Implementierung der statischen Utility-Methode `fahrenheit_to_celsius` in der Klasse `temperature_conversion`.

```
CLASS temperature_conversion IMPLEMENTATION.

  METHOD fahrenheit_to_celsius.
     result = ( temperature_in_fahrenheit - 32 ) * 5 / 9.
  ENDMETHOD.

ENDCLASS.
```

Listing 4.5 Implementierung einer Methode

Rufen Sie statische Methoden nicht durch Referenzvariablen auf

Statische Methoden rufen Sie für gewöhnlich auf, indem Sie den Klassennamen nennen, gefolgt von einem Pfeil mit Doppelstrich, und dem Methodennamen, also z. B. `temperature_conversion=>fahrenheit_to_celsius`.

Sie können sie auch wie Instanzmethoden aufrufen, indem Sie eine Referenzvariable nennen, gefolgt von einem Pfeil mit Einfachstrich, und dem Methodennamen, also z. B. `my_converter->fahrenheit_to_celsius`.

Wir raten von letzterer Variante ab. Die statische Methode hängt an der Klasse selbst, nicht an der Instanz. Sie über die Instanz aufzurufen, wirkt auf die Leserinnen und Leser Ihres Codes verwirrend. Machen Sie den statischen Aufruf deutlich, indem Sie die erstere Variante benutzen.

4.1.2 Öffentliche Instanzmethoden

Öffentliche Instanzmethoden definieren die Schnittstelle einer Klasse, die von außenstehenden Konsumenten benutzt werden kann. Jedes Stück Code, das die Klasse selbst sehen kann, kann diese Methoden aufrufen.

Abhängigkeiten lassen sich besser verwalten und Komponenten besser voneinander isolieren, wenn Sie sich von konkreten Klassen loslösen und sich stattdessen auf die Abstraktion konzentrieren, die die Klasse repräsentiert. Auf diese Weise lassen sich Implementierungsdetails leichter durch alternative Implementierungen, wie etwa Test-Doubles, ersetzen.

Öffentliche Instanzmethoden sollten Teil eines Interface sein

Damit sich eine Klasse sauber ins größere Ganze einfügt, sollte sie Interfaces implementieren. Genauer gesagt sollten die meisten – wenn nicht sogar alle – ihrer öffentlichen Instanzmethoden Bestandteil eines Interface sein.

Die Klasse wird so von »der einen« Implementierung eines Konzepts zu »einer möglichen« Implementierung des Konzepts heruntergestuft. Konsumenten hängen nur noch vom Konzept – dem Interface – selbst ab, jedoch überhaupt nicht mehr von der Klasse selbst. Dieser wichtige Schritt ermöglicht das Dependency-Inversion-Prinzip, das »D« im Merkwort SOLID.

Nicht alle Klassen müssen Interfaces implementieren. Ausnahmen, Werteobjekte (siehe Abschnitt 3.1.2, »Klassen und Objekte«) und Enumerations (siehe Abschnitt 6.2.2, »Aufzählungsklassen«) sind beispielsweise oft alternativlose, einzig gültige Implementierungen. Sie müssen daher keine abstrakte Repräsentierung ihrer Operationen in Form von Interfaces anbieten. Typischerweise implementieren Klassen, die

ausschließlich Daten enthalten, jedoch keine Services dazu anbieten, keine Interfaces. Mehr über Interfaces und ihre Vorteile erfahren Sie in Abschnitt 3.1.1, »Interfaces«.

Schauen wir uns noch einmal das Design der Klasse `thermal_switch` aus Abschnitt 3.1.2, »Klassen und Objekte«, an, wie in Listing 3.15 gezeigt. Der Hauptzweck dieser Klasse ist es, das von ihr verschaltete Gerät `device` vor Überhitzung zu schützen. Ihre Methoden sind `trip`, die bei Überhitzung aufgerufen wird und das `device` abschaltet, und `reset`, die den Schalter in seinen neutralen Normalzustand zurücksetzt. Diese Methoden waren als öffentliche Instanzmethoden direkt in der Klasse selbst implementiert.

Das hier beschriebene Konzept können Sie als Abstraktion bereitstellen, indem Sie ein Interface `thermal_protector` extrahieren, wie in Listing 4.6 gezeigt.

```
INTERFACE thermal_protector
  PUBLIC.

  METHODS:
    trip
      FOR EVENT critical_temperature_reached OF thermal_sensor,
    reset.

ENDINTERFACE.
```

Listing 4.6 Das Interface thermal_protector

Die Klasse `thermal_switch` implementiert dann dieses Interface `thermal_protector`. Alle ihre öffentlichen Instanzmethoden sind dadurch Bestandteil eines Interface, wie in Listing 4.7 zu sehen.

```
CLASS thermal_switch DEFINITION
  FINAL
  PUBLIC.

PUBLIC SECTION.

  INTERFACES:
    switchable,
    thermal_protector.

  ALIASES:
    trip FOR thermal_protector~trip,
    reset FOR thermal_protector~reset.
```



```
  METHODS:
    constructor
      IMPORTING
        managed_device TYPE REF TO switchable
        sensor TYPE REF TO thermal_sensor.
...
```

Listing 4.7 Die PUBLIC SECTION der überarbeiteten Klasse thermal_switch

Abbildung 4.1 stellt das resultierende Design in UML dar. Sie sehen, dass ein `thermal_switch` sowohl ein `thermal_protector` als auch ein `switchable` ist.

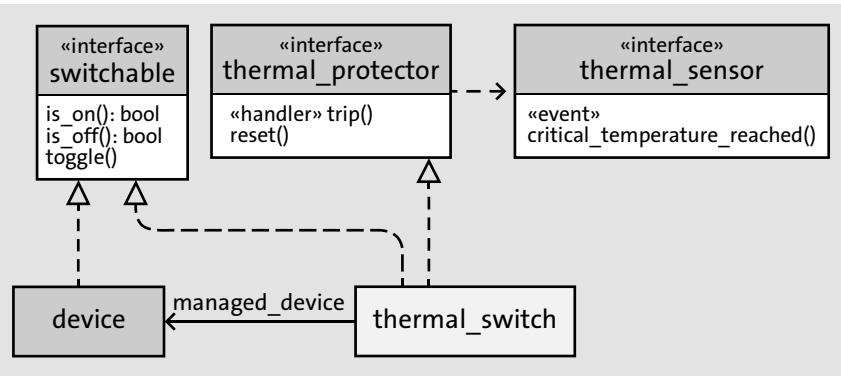

Abbildung 4.1 Die Klasse thermal_switch und das Interface thermal_protector

4.1.3 Redefinition einer Methode

Unterklassen können Instanzmethoden, die sie von ihrer Oberklasse geerbt haben, überschreiben. Voraussetzung dafür ist, dass die Methode nicht `FINAL` ist. Sie können dadurch speziellere Implementierungen anbieten, die anstelle der Methoden der Oberklasse aufgerufen werden.

Schauen Sie sich dazu noch einmal die Klasse `fan` aus Kapitel 3, »Klassen und Interfaces«, an. Angenommen, wir hätten die neue Anforderung, eine spezielle Variante eines `fan` anzubieten, der die Intensität gleich mehrere Stufen auf einmal verändert, wenn die Intensität hoch oder herunter geregelt wird. Eine Möglichkeit, dies umzusetzen, ist, eine Unterklasse `step_fan` anzulegen, die die entsprechenden Methoden redefiniert, wie in Listing 4.8 zu sehen.

```
CLASS step_fan DEFINITION
  PUBLIC
  FINAL
  INHERITING FROM fan.
```

```
PUBLIC SECTION.

  METHODS:
    constructor
      IMPORTING step TYPE int4,
    increase_intensity REDEFINITION,
    decrease_intensity REDEFINITION.

PRIVATE SECTION.

  DATA:
    step TYPE int4.

ENDCLASS.

CLASS step_fan IMPLEMENTATION.

  METHOD constructor.
    super->constructor( ).
    me->step = step.
  ENDMETHOD.

  METHOD increase_intensity.
    DO step TIMES.
      super->increase_intensity( ).
    ENDDO.
  ENDMETHOD.

  METHOD decrease_intensity.
    DO step TIMES.
      super->decrease_intensity( ).
    ENDDO.
  ENDMETHOD.

ENDCLASS.
```

Listing 4.8 Die Unterklasse step_fan

Die Klasse `step_fan` erbt von der Klasse `fan` und redefiniert die Methoden `increase_intensity` und `decrease_intensity` so, dass sie die Intensität in mehreren Schritten auf einmal verändern. Beachten Sie, wie die Methoden einfach nochmal mit dem ergänzenden Schlüsselwort `REDEFINITION` aufgeführt werden. Redefinierte Methoden können die ursprünglich geerbten Methoden sehen und aufrufen, indem sie die

Oberklasse mit der Syntax `super->` direkt ansprechen. Beachten Sie auch, dass der Konstruktor der Unterklasse als allerersten Schritt den Konstruktor der Oberklasse aufrufen muss.

Vorsicht beim Redefinieren von Methoden

Wie bereits in Abschnitt 3.1.2, »Klassen und Objekte«, erläutert, ist Vererbung ein mächtiges Werkzeug, das jedoch schwer unter Kontrolle zu halten ist.

Wenn Sie Methoden redefinieren, passen Sie das Verhalten der Oberklasse an neue Anforderungen an. Wenn ihre Veränderungen im Widerspruch zu dem stehen, was Konsumenten von der Oberklasse erwarten, verletzen Sie das Liskovsche Substitutionsprinzip, das »L« in SOLID (siehe Abschnitt 3.1.1, »Interfaces«).

Gehen Sie daher sparsam mit Redefinitionen um. Versuchen Sie, sich darauf zu beschränken, nur solche Methoden zu redefinieren, die in der Oberklasse `ABSTRACT` sind, also dort mit Absicht gar nicht implementiert. Auf diese Weise behalten Sie leichter den Überblick über das Design, denn jede Klasse hat nur eine einzige Version jeder Methode. Dieses Muster wird im Buch *Effektives Arbeiten mit Legacy Code* (mitp, 2010) auch *normalisierte Hierarchie* genannt.

4.2 Parameter

Methodenparameter lassen sich grob in drei Arten einteilen: *Eingabeparameter*, *Ausgabeparameter* und *Ein-/Ausgabeparameter*. Eingabeparameter werden mit dem Schlüsselwort `IMPORTING` deklariert, Ausgabeparameter mit `EXPORTING` oder – wenn es nur genau einen gibt – mit `RETURNING`. Ein-/Ausgabeparameter werden mit `CHANGING` deklariert.

Eine klassische Quelle der Verwirrung für ABAP-Neulinge ist, dass sich alle Schlüsselworte außer `CHANGING` beim Aufruf der Methode umkehren, wie in Tabelle 4.1 zu sehen.

Deklaration	Aufruf
IMPORTING	EXPORTING
EXPORTING	IMPORTING
RETURNING	RECEIVING
CHANGING	CHANGING

Tabelle 4.1 Schlüsselworte für Parameter bei der Deklaration und beim Aufruf

Beim Aufruf einer Methode müssen manche Schlüsselwörter explizit genannt werden, während andere weggelassen werden können. In diesem Abschnitt schauen wir uns an, wie Sie Methoden am besten deklarieren können. In Abschnitt 4.4 schauen wir uns dann an, wie Sie die Methoden am besten aufrufen können.

4.2.1 Wie viele Parameter sind zu viele?

Schauen Sie sich die Eingabeparameter der Methode in Listing 4.9 an.

```
METHODS add_item
  IMPORTING
    product_id    TYPE product_id
    product_group TYPE product_group
    amount        TYPE int4
    unit          TYPE unit_of_measure.
```

Listing 4.9 Zu viele Eingabeparameter?

Die Methode `add_item` hat Parameter, die inhaltlich zusammengehören und nur gemeinsam Sinn ergeben, während andere Parameter davon unabhängige Zustände beschreiben. Strukturen oder Klassen einzuführen, um den Zusammenhalt der Dinge hervorzuheben, macht sie für Konsumenten besser erkennbar, wie in Listing 4.10 zu sehen.

```
METHODS add_item
  IMPORTING
    product  TYPE REF TO product
    quantity TYPE quantity.
```

Listing 4.10 Weniger Eingabeparameter durch Zusammenfassen

Zu viele Eingabeparameter können ein Hinweis darauf sein, dass eine Methode zu viele verschiedene Aufgaben hat. Mehr dazu finden Sie in Abschnitt 4.3.1, »Eine Sache tun«.

[+]

Halten Sie die Anzahl der Eingabeparameter so gering wie möglich

Minimieren Sie die Anzahl der `IMPORTING`-Parameter Ihrer Methoden. Die Anzahl der Eingabeparameter beeinflusst drastisch, wie verständlich und leicht verwendbar eine Methode ist. Sie lässt auch Rückschlüsse darauf zu, wie komplex die Implementierung im Inneren ist. Mehr Eingabeparameter erschweren das Testen, denn Sie müssen mehr Kombinationen testen. Ganz allgemein gilt deshalb: Je weniger Eingabeparameter eine Methode hat, desto besser. Die meisten Ihrer Methoden sollten mit maximal zwei Eingabeparametern auskommen. Bereits ab drei Eingabeparametern nimmt die Anzahl der möglichen Eingabekombinationen extrem zu und Ihre Methode ist schwerer zu testen.

4.2.2 Optionale Eingabeparameter

ABAP unterstützt das *Überladen* von Methoden nicht. Überladen bedeutet, dass eine Klasse mehrere Methoden mit demselben Namen enthält, die sich nur in Anzahl und Art ihrer Eingabeparameter unterscheiden. In der Vergangenheit haben ABAP-Entwicklerinnen und -Entwickler diesen Umstand oft umgangen, indem sie optionale Eingabeparameter hinzugefügt haben. Diese Eingabeparameter sind mit dem Schlüsselwort `OPTIONAL` gekennzeichnet und *können* beim Aufruf mitgegeben werden, *müssen* es aber nicht. Die Methode variiert dann ihr Verhalten, je nachdem, ob der Parameter mitgegeben wird oder nicht.

Ein typisches Beispiel ist eine Methode, die sowohl ein einzelnes »Ding« als auch eine »Liste von Dingen« in Empfang nehmen kann, wie in Listing 4.11 zu sehen.

```
METHODS process
  IMPORTING
    item_to_process TYPE item OPTIONAL
    items_to_process TYPE item_table OPTIONAL.
```

Listing 4.11 Optionale Eingabeparameter, die sich gegenseitig ausschließen

Signaturen mit optionalen Eingabeparametern sind schwer zu verstehen und verwirren den Aufrufer. Sie machen auch die Implementierung schwieriger, da Sie auch Fälle berücksichtigen müssen, die nur bedingt sinnvoll sind. Diese Signaturen verschleiern, welche Eingabeparameter unbedingt notwendig sind, welche Kombinationen Sinn ergeben und welche Parameter sich gegenseitig ausschließen.

Vermutlich ist es beabsichtigt, die Methode `process` in Listing 4.11 entweder mit dem Eingabeparameter `item_to_process` oder mit dem Eingabeparameter `items_to_process` alleine aufzurufen. Doch was passiert, wenn wir beide Parameter gleichzeitig füllen? Oder keinen von beiden? Diese Aufrufe sind syntaktisch völlig in Ordnung, es ist aber unklar, welches Verhalten wir erwarten können. Auch Entwicklerinnen und Entwickler haben mehr Mühe, denn sie müssen diese eigentlich ungewollten Möglichkeiten mit bedenken, behandeln und testen.

[+]

Teilen Sie Methoden, statt optionale Eingabeparameter hinzuzufügen

Separate Verhaltensweisen sollten auch in separaten Methoden implementiert werden, siehe hierzu auch Abschnitt 4.3.1, »Eine Sache tun«. Legen Sie für jeden Fall eine eigenständige Methode mit einem sprechenden Namen und der gültigen Kombination an nichtoptionalen Eingabeparametern an. Diese sind sowohl für den Konsumenten als auch für Sie selbst leichter zu verstehen.

Das Beispiel in Listing 4.11 könnten Sie z. B. wie in Listing 4.12 gezeigt aufteilen.

```
METHODS:

  process_item
    IMPORTING
      item_to_process TYPE item,

  process_items
    IMPORTING
      items_to_process TYPE item_table.
```

Listing 4.12 Separate Methoden für separate Funktionen

Als Alternative zu optionalen Eingabeparametern bieten sich manchmal Eingabeparameter mit dem Zusatz `DEFAULT` an. Auch dieser Zusatz macht den Parameter optional, allerdings auf eine Art und Weise, die besser verständlich ist. Der Konsument kann sich diesen Parameter aus Bequemlichkeit sparen, wenn er mit dessen Vorgabewert einverstanden ist. Für die Implementierung der Methode ist der Parameter jedoch keineswegs optional, sie setzt ihn zwingend voraus.

Ein Beispiel für einen sinnvollen `DEFAULT`-Parameter ist in Listing 4.13 zu sehen. Der Eingabeparameter `separator` der Methode `to_csv` ist optional. Stellt der Aufrufer ihn nicht zur Verfügung, wird der Vorgabewert `` `,` `` verwendet. Der Code im Inneren muss auf diese Unterscheidung nicht eingehen – er tut so, als wäre der Eingabeparameter in jedem Fall verfügbar.

```
METHODS to_csv
  IMPORTING
    separator TYPE string DEFAULT `,`
  RETURNING
    ...
```

Listing 4.13 Parameter mit DEFAULT

4.2.3 Bevorzugte Eingabeparameter

Wenn eine Methode ausschließlich Eingabeparameter mit den Zusätzen `OPTIONAL` oder `DEFAULT` hat, kann einer dieser Parameter mit dem Zusatz `PREFERRED PARAMETER` gekennzeichnet werden. Listing 4.14 zeigt ein Beispiel.

```
METHODS to_csv
  IMPORTING
    separator TYPE string DEFAULT `,` PREFERRED PARAMETER
    quote type string DEFAULT `"`
  RETURNING
    ...
```

Listing 4.14 Methode mit PREFERRED PARAMETER

Dieser Zusatz erspart es dem Aufrufer, den Eingabeparameter zu benennen, wenn er exakt einen Parameter mitgibt. Der Wert wird automatisch dem bevorzugten Parameter zugeführt. Die Methode `to_csv` aus Listing 4.14 kann demzufolge wie folgt aufgerufen werden:

```
DATA(item_as_csv) = item->to_csv( `;` ).
```

Diese Aufrufart ist natürlich kürzer, kann jedoch leicht missverständlich werden – insbesondere wenn der bevorzugte Parameter nicht unbedingt die Vorlieben des Aufrufers widerspiegelt. Es kann die Verständlichkeit fördern, den Parameter trotzdem zu benennen:

```
DATA(item_as_csv) = item->to_csv( separator = `;` ).
```

Optionale Eingabeparameter, die nicht mit `PREFERRED PARAMETER` markiert sind, müssen immer benannt werden.

Setzen Sie bevorzugte Eingabeparameter spärlich ein

Nutzen Sie den Zusatz `PREFERRED PARAMETER` nur dann, wenn der Methodenname intuitiv und selbsterklärend widerspiegelt, welcher namenlose Wert da übergeben wird. Abschnitt 4.4.4, »Optionale Parameternamen«, gibt Ihnen weitere Hinweise für Methodenaufrufe.

4.2.4 Boolesche Eingabeparameter

Stellen Sie sich vor, Sie hätten eine Klasse `credit_score_service`, deren Methode `calculate_credit_score` die Kreditwürdigkeit eines Kunden berechnet und als Zahl darstellt. Ein Beispiel für eine solche Klasse sehen Sie in Listing 4.15. Die Methode `calculate_credit_score` hat darin einen Eingabeparameter mit dem Datentyp Boole (`abap_bool`), der entscheidet, wie präzise die Heuristiken des Algorithmus arbeiten sollen.

```
CLASS credit_score_service DEFINITION
  PUBLIC
  FINAL.

PUBLIC SECTION.

  METHODS:

    constructor
      IMPORTING customer TYPE REF TO customer,
```

```
    calculate_credit_score
      IMPORTING use_strong_heuristics TYPE abap_bool
      RETURNING VALUE(credit_score) TYPE int4.

PRIVATE SECTION.

  DATA:
    customer TYPE REF TO customer.

ENDCLASS.
```

Listing 4.15 Berechnung der Kreditwürdigkeit mit einer Methode mit Booleschem Parameter

Die Methode `calculate_credit_score` wird in Kurzform wie folgt aufgerufen:

```
DATA(my_credit_score) =
  my_credit_score_service->calculate_credit_score( abap_true ).
```

Dieses Stück Code lässt beim Lesen völlig offen, welche Bedeutung `abap_true` hat, und wird die meisten Leserinnen und Leser zwingen, zeitaufwendig in der Methodendeklaration nachzuschauen. Etwas besser verständlich wird es, wenn Sie den Parameter beim Aufruf benennen:

```
DATA(my_credit_score) =
  my_credit_score_service->calculate_credit_score(
    use_strong_heuristics = abap_true ).
```

Die Aufgabe, den Aufruf lesbar zu gestalten, bleibt dabei am Konsumenten hängen. Obwohl der Aufruf nun eventuell besser verständlich ist, bleibt ein grundlegendes Problem bestehen. Stellen Sie sich dazu vor, wie die Implementierung dieser Methode aussieht. Vermutlich werden Sie etwas wie in Listing 4.16 vorfinden.

```
METHOD calculate_credit_score.

  IF use_strong_heuristics = abap_true.
    " use a slower but more robust algorithm....

  ELSE.
    " use a faster but less robust algorithm....

  ENDIF.

ENDMETHOD.
```

Listing 4.16 Verzweigte Implementierung

Wenn eine Methode einen Booleschen Eingabeparameter erhält, tut sie für gewöhnlich zwei Dinge statt einem. Wie Sie später in Abschnitt 4.3.1, »Eine Sache tun«, sehen werden, ist das für gewöhnlich keine gute Idee.

Legen Sie besser separate Methoden statt Boolescher Eingabeparameter an

Boolesche Parameter zu verwenden, um den Kontroll- und Verzweigungsfluss innerhalb einer Methode von außen zu steuern, macht die Methode komplexer, schwieriger zu verwenden und schwieriger zu warten. Implementieren Sie verschiedene Verhaltensweisen besser in separaten Methoden. Nutzen Sie die Namen der Methoden, um die Unterschiede hervorzuheben.

In unserem Beispiel sollten wir die Methode `calculate_credit_score` besser in zwei separate Methoden unterteilen, wie in Listing 4.17 zu sehen.

```
METHODS:

  calculate_simple_credit_score
    RETURNING VALUE(credit_score) TYPE int4,

  calculate_complex_credit_score
    RETURNING VALUE(credit_score) TYPE int4.
```

Listing 4.17 Separate Methoden anstatt einer einzelnen verzweigten

Die Trennung erlaubt es den beiden Methoden, sich voll und ganz auf ihre jeweilige Variante des Algorithmus konzentrieren, ohne auf die jeweilig andere Acht geben zu müssen. Auch für den Konsumenten ist über die genaueren Methodennahmen klarer, worin sich die Algorithmen unterscheiden.

Diese Trennung nach außen hin bedeutet keineswegs, dass Sie im Inneren duplizierten Code vorfinden werden. Sollten die beiden Algorithmen gleiche Teile haben, können Sie diese problemlos in private Methoden auslagern.

Sie könnten sogar noch einen Schritt weitergehen und für jeden der Algorithmen eine eigene Klasse anlegen. In dem Fall würden Sie ein Interface anlegen, das die Methode `calculate_credit_score` deklariert. Dies erspart dem Konsumenten die Entscheidung, ob er die eine oder andere Methode aufrufen soll: Er ruft einfach die Methode an demjenigen Objekt auf, dass ihm von außen bereitgestellt wurde.

4.2.5 EXPORTING-Parameter

Methoden können Aufrufern ihre Ergebnisse als `EXPORTING`- oder `RETURNING`-Parameter zurückgeben. Mit `EXPORTING` können Sie mehrere Rückgabewerte definieren, wie

in Listing 4.18 zu sehen, während Sie mit RETURNING exakt einen Rückgabewert liefern können.

```
METHODS check_business_partners
  IMPORTING
    business_partners TYPE business_partners
  EXPORTING
    result       TYPE result_type
    failed_keys  TYPE /bobf/t_frw_key
    messages     TYPE /bobf/t_frw_message.
```

Listing 4.18 Mehrere »EXPORTING«-Parameter

Die Methode check_business_partners validiert die im Eingabeparameter business_partners übergebene Liste von Geschäftspartnern. Das Ergebnis dieser Validierung wird in drei separaten Ausgabeparametern zurückgegeben: result liefert das Gesamtergebnis, failed_keys die Schlüssel der ungültigen Geschäftspartner und messages Details, was genau ungültig ist. EXPORTING-Parameter sind optional für den Aufrufer, d. h., Sie können also frei entscheiden, einige davon entgegenzunehmen und andere nicht, wie in Listing 4.19 zu sehen.

```
check_business_partners(
  EXPORTING
    business_partners = my_business_partners
  IMPORTING
    result            = DATA(business_partners_check_result)
    messages          = DATA(business_partner_messages) ).
```

Listing 4.19 Aufruf einer Methode mit mehreren EXPORTING-Parametern

Beachten Sie, dass sie beim Aufruf das gegenteilige Schlüsselwort verwenden müssen: Obwohl der Parameter aus Sicht der Methode EXPORTING ist, müssen Sie ihn im Aufruf als IMPORTING nennen.

[+]

Minimieren Sie die Anzahl der EXPORTING-Parameter

Zu viele EXPORTING-Parameter können ein Hinweis darauf sein, dass Ihre Methode mehrere Dinge auf einmal macht, was nicht im Sinne sauberen Codes ist, siehe auch Abschnitt 4.3.1, »Eine Sache tun«. Gute Methoden konzentrieren sich auf eine einzelne Tätigkeit, und das spiegelt sich für gewöhnlich in einer kleinen Anzahl von Ausgabeparametern wider.

Die Ausgabeparameter Ihrer Methode sollten eine logische Einheit bilden, also beispielsweise verschiedene Details derselben Sache darstellen. Haben Sie mehrere Ausgabeparameter, können Sie diese für gewöhnlich in Strukturen oder Klassen zusammenfassen und dadurch die Anzahl reduzieren. Wenn die Ausgabeparameter Ihrer Methode keine solche logische Einheit bilden, sollten Sie die Methode besser in mehrere separate Methoden unterteilen.

In unserer Beispielmethode check_business_partners bilden die Ausgabeparameter die logische Einheit »Prüfergebnis«, die wir als Struktur check_result mit Unterkomponenten für die drei Details abbilden können. Das erlaubt es, die drei Ausgabeparameter zu einem einzelnen Parameter zusammenzufassen, wie in Listing 4.20 zu sehen.

```
TYPES:
  BEGIN OF check_result,
    result      TYPE result_type,
    failed_keys TYPE /bobf/t_frw_key,
    messages    TYPE /bobf/t_frw_message,
  END OF check_result.

METHODS check_business_partners
  IMPORTING
    business_partners TYPE business_partners
  EXPORTING
    result TYPE check_result.
```

Listing 4.20 EXPORTING-Parameter reduzieren

Beachten Sie jedoch, dass selbst diese Reduktion noch nicht die höchste Stufe der Vereinfachung darstellt; diese lernen Sie im folgenden Abschnitt 4.2.6 mit RETURNING kennen.

4.2.6 RETURNING-Parameter

RETURNING-Parameter definieren exakt einen Rückgabewert für eine Methode. Diesen könnte man also als »das Ergebnis« des Methodenaufrufs interpretieren.

Unsere Vereinfachung der Methode check_business_partners in Listing 4.20 führte zu einer Methode, die nur noch einen einzigen EXPORTING-Parameter hat. Diese Methode wird wie in Listing 4.21 gezeigt aufgerufen.

```
check_business_partners(
  EXPORTING
    business_partners = my_business_partners
  IMPORTING
    result = DATA(business_partners_check_result) ).
```

Listing 4.21 Aufruf einer Methode mit einem einzelnen EXPORTING-Parameter

Nutzen Sie RETURNING statt EXPORTING

RETURNING-Parameter sind der beste Weg, um Rückgabewerte von Methoden zu deklarieren. Diese Form erlaubt es, den Aufruf deutlich zu verkürzen. Es ermöglicht auch das Verketten von Aufrufen, bei denen auf dem von einer Methode zurückgegebenen Objekt sofort die nächste Methode aufgerufen wird. Wenn Sie die Anzahl an EXPORTING-Parametern so weit reduzieren, dass nur noch einer übrig bleibt, sollten Sie diesen in einen RETURNING-Parameter umwandeln.

Einige ältere Performancetipps raten noch dazu, EXPORTING-Parameter zu verwenden, wenn sie große Tabellen mit Daten übergeben, siehe auch Abschnitt 4.2.8, »Übergabe per Wert und Übergabe per Referenz«. Dies ist in modernen ABAP-Versionen jedoch nicht mehr nötig und Sie können auch hier bedenkenlos zu RETURNING greifen. Weichen Sie von diesem Muster nur ab, wenn Sie messbare Vorteile davon haben.

Listing 4.22 zeigt das letzte Refactoring der Methode check_business_partners, in dem wir den EXPORTING-Parameter durch einen RETURNING-Parameter ersetzt haben.

```
METHODS check_business_partners
  IMPORTING
    business_partners TYPE string
  RETURNING
    VALUE(result) TYPE check_result.
```

Listing 4.22 Deklaration des RETURNING-Parameters

Nutzen Sie RETURNING, werden Aufrufe der Methode check_business_partners deutlich kürzer:

```
DATA(business_partners_check_result) =
  check_business_partners( my_business_partners ).
```

Nutzen Sie result als Namen des RETURNING-Parameters

Die meisten modernen Programmiersprachen nutzen das Konzept eines einzelnen RETURNING-Parameters. Die wenigsten allerdings bieten die Möglichkeit oder wie ABAP gar die Notwendigkeit, diesem Parameter einen Namen zu geben. Für gewöhnlich wird stattdessen ein Schlüsselwort, gefolgt vom Rückgabewert selbst, angegeben, z. B. return 42.

Gut benannte Methoden machen es für gewöhnlich überflüssig, dem Rückgabeparameter einen eigenständigen Namen zu geben. Der Parametername wiederholt in solchen Fällen zumeist nur den Methodennamen oder etwas ähnlich Offensichtliches und führt im ungeschicktesten Fall – z. B. bei Gettern – sogar zu Konflikten mit den Attributen der Klasse.

Konsumenten werden den Parameternamen in den wenigsten Fällen sehen. Zwar gibt es eine Syntax mit RECEIVING, die den Parameternamen wiederholt, diese wird aber (zu Recht) nur äußerst selten verwendet. Für gewöhnlich nutzen Konsumenten stattdessen die funktionale Schreibweise, bei der der Rückgabewert direkt einer Variablen zugewiesen wird – die dann wieder einen sprechenden Namen hat.

Aus all diesen Gründen empfehlen wir Ihnen, den RETURNING-Parameter von Methoden immer gleich zu nennen. Der Name result hat sich in nahezu allen Situationen als gut lesbare Wahl bewährt. Diese Gleichförmigkeit macht den Parameter im besten Sinne »unsichtbar«, er wird quasi zu einem Schlüsselwort.

Geben Sie dem Rückgabeparameter nur dann einen eigenständigen Namen, wenn aus dem aktuellen Kontext nicht ersichtlich ist, was die Methode zurückgibt, oder der Rückgabewert nicht exakt zum Methodennamen passt.

Bei einer Methode create_business_partner würde man beispielsweise erwarten, dass die Methode den erzeugten Geschäftspartner selbst zurückgibt. Liefert die Methode aber nur den Schlüssel des neuen Geschäftspartners, kann es eine gute Idee sein, den Rückgabeparameter key oder ähnlich zu nennen.

Manchmal hat eine Methode auch gar keinen echten Rückgabewert, gibt aber trotzdem me zurück, damit der Konsument weitere verkettete Methodenaufrufe anschließen kann. Auch in diesem Fall kann es sinnvoll sein, dies über den Parameternamen zu verdeutlichen.

4.2.7 CHANGING-Parameter

CHANGING-Parameter können Sie für Werte nutzen, die gleichzeitig Ein- und Ausgabeparameter sind. Die Methode kann den Inhalt dieser Parameter ändern, anders als bei reinen Eingabeparametern, die vor Änderungen geschützt sind. Dieser Zusatz ist vorrangig für Strukturen und internen Tabellen sinnvoll, also für Werte mit internen Unterkomponenten wie Feldern und Zeilen.

Stellen Sie sich beispielsweise vor, Sie wollten eine größere Menge von Entitäten validieren und alle gefundenen Ergebnisse in einer einzigen Tabelle zusammenfassen. In diesem Fall bietet sich eine Lösung mit CHANGING-Parameter an, wie in Listing 4.23 zu sehen.

```
METHODS validate
  CHANGING
    messages TYPE message_table.
```

Listing 4.23 Deklaration eines CHANGING-Parameters

Diese Instanzmethode validiert die aktuelle Entität und fügt alle Befunde dem Sammelparameter messages hinzu, einer internen Tabelle von Textnachrichten. Beim

Aufruf der Methode könnte der Parameter `messages` bereits Einträge enthalten, z. B. von früheren Aufrufen derselben Methode. Die Methode löscht diese nicht, sondern fügt ihre eigenen Einträge am Ende oder gemäß einer Tabellensortierung an den passenden Stellen in der Mitte hinzu. Die Methode kann dann wiederholt aufgerufen werden, um alles zu validieren, wie z. B. in Listing 4.24.

[!]

Nutzen Sie CHANGING-Parameter nicht zu oft

Sparen Sie sich `CHANGING`-Parameter für die Fälle auf, in denen Sie vorhandene Strukturen oder Tabellen modifizieren wollen, die bereits vor der Methodenanwendung Daten enthalten können, die erhalten bleiben sollen.

Möchten Sie hingegen den Inhalt des Parameters komplett vorgeben, d. h., die bereits darin enthaltenen Daten müssen nicht erhalten bleiben, so sollten Sie den Parameter besser als `EXPORTING` oder `RETURNING` deklarieren.

Möchten Sie hingegen in Ihrer Methode den Zustand eines Objekts verändern, können Sie dieses nach wie vor als `IMPORTING`-Parameter übergeben. Sie können dann trotzdem Methoden am Objekt aufrufen, die dessen Zustand ändern, oder sichtbare Attribute direkt modifizieren. Der Schutz des Eingabeparameters besteht in dieser Form darin, dass Sie die Objektreferenz selbst nicht abändern können, d. h., Sie können das vorliegende Objekt nicht komplett durch ein anderes ersetzen. Eine derartige Änderung ergibt allerdings auch nur in den allerwenigsten Fällen Sinn, sodass Sie `CHANGING`-Objektreferenzen in der Regel nicht antreffen werden.

```
DATA messages TYPE message_table.
LOOP AT entities INTO DATA(entity).
  entity->validate( CHANGING messages = messages ).
ENDLOOP.
```

Listing 4.24 CHANGING-Parameter verwenden

`CHANGING`-Parameter lassen sich gut durch Objekte ersetzen, die die entsprechenden Daten enthalten und lesende und ändernde Zugriffe darauf erlauben. Im vorliegenden Beispiel könnte das eine Klasse `message_container` sein, wie wir sie bereits früher in Listing 4.16 angetroffen und in Abschnitt 3.1.3, »Zustand«, besprochen haben. Die refaktorierte Methode `validate` könnte dann aussehen wie in Listing 4.25.

```
METHODS validate
  IMPORTING
    message_container TYPE ref to message_container.
```

Listing 4.25 CHANGING-Parameter durch IMPORTING-Objektreferenz ersetzen

Objekte können die Arbeit mit den Sammeldaten durch bequemere Methoden deutlich erleichtern. In unserem Fall könnten wir beispielsweise ein Interface verwenden,

das es erlaubt, Nachrichten hinzuzufügen, aber keine Möglichkeit bietet, Nachrichten zu löschen. So können wir versehentliche Programmierfehler vermeiden, wie sie beim Vollzugriff auf die darunter liegende Datenstrukturen passieren können.

Die Methoden können auch die Komplexität der Datenstrukturen verkleiden, sodass wir bedenkenlos eine Methode `add_message` aufrufen können, ohne uns Gedanken darüber machen zu müssen, ob wir es mit einer indizierten oder sortierten Tabelle zu tun haben. Auch der Aufruf der Methode wird etwas kürzer, denn wir müssen nun das Schlüsselwort `CHANGING` und den Parameternamen nicht mehr wiederholen, wie in Listing 4.26 zu sehen.

```
DATA(message_container) = NEW message_container( ).
LOOP AT entities INTO DATA(entity).
  entity->validate( message_container ).
ENDLOOP.
```

Listing 4.26 Verwendung der geänderten Methode

[+]

Vermeiden Sie es, verschiedene Arten von Ausgabeparametern zu mischen

Nutzen Sie eine der Varianten `EXPORTING`, `CHANGING` oder `RETURNING` in einer Methode, aber nicht mehrere dieser Arten zusammen.

Verschiedene Arten von Ausgabeparametern sind ein deutliches Zeichen, dass die Methode mehr als eine Aufgabe hat. Das ist verwirrend und der Aufruf der Methode wird unnötig kompliziert.

4.2.8 Übergabe per Wert und Übergabe per Referenz

Zusätzlich zur Art der Parameter können Sie auch festlegen, ob sie per Wert oder per Referenz übergeben werden.

Bei der Übergabe per Wert, englisch *pass by value*, arbeitet die Methode auf einer Kopie der ursprünglichen Variablen, die in einem eigenen Speicherbereich angelegt wird. Das passiert sogar, wenn Sie eine Objektreferenz übergeben, wobei hier allerdings nur die Referenz selbst kopiert wird, also der relativ kurze Zeiger auf den Speicherbereich, in dem dann wiederum das eigentliche Objekt liegt. ABAP verwendet einige Optimierungen, um den Aufwand dieser Kopiervorgänge zu minimieren. Insbesondere wird die Kopie typischerweise nach der *Copy-on-Write*-Strategie angelegt, also erst dann, wenn der Inhalt der Variablen oder des Parameters geändert wird.

Bei der Übergabe per Referenz, englisch *pass by reference*, hingegen wird ein Zeiger auf das ursprüngliche Objekt übergeben. Bekommt die Methode die Erlaubnis, den Parameter zu ändern, wird dadurch die außerhalb der Methode liegende Variable direkt mit verändert.

Für IMPORTING, EXPORTING und CHANGING ist die Übergabe per Referenz der Standard. Für RETURNING ist hingegen die Übergabe per Wert die einzig mögliche Wahl. Die RETURNING-Zusätze weisen deshalb stets das für diese Übergabeart prägende Schlüsselwort VALUE auf.

Die Übergabe per Referenz kann über das Schlüsselwort REFERENCE explizit gemacht werden. Dies wird allerdings nur selten genutzt. Unser Code aus Listing 4.10 könnte diese Übergabeart wie in Listing 4.27 verdeutlichen.

```
METHODS add_item
  IMPORTING
    REFERENCE(product)  TYPE REF TO product
    REFERENCE(quantity) TYPE quantity.
```

Listing 4.27 Explizite Syntax für Übergabe per Referenz

Weisen Sie Eingabeparametern, die per Wert übergeben werden, keine neuen Werte zu

Werden Eingabeparameter per Referenz übergeben – dies stellt den Standard dar –, können sie innerhalb der Methode *nicht* verändert werden. Eingabeparameter, die per Wert übergeben werden, *können* jedoch verändert werden. Da die Methode auf einer Kopie der Originalvariable agiert, ist diese Wertveränderung für den Aufrufer außerhalb der Methode nicht sichtbar.

Sie sollten es vermeiden, IMPORTING-Parameter zu verändern. Eingabeparameter kommen vom Aufrufer. So zu tun, als hätte der Aufrufer einen anderen Wert übergeben, provoziert Missverständnisse und Fehler.

Wollen Sie nicht direkt mit der Originaleingabe weiterarbeiten, was z. B. bei der Normalisierung oder Maskierung durchaus üblich ist, weisen Sie den modifizierten Wert einer neuen Variablen zu und arbeiten mit dieser weiter. Unter anderem wegen dieser Feinheit ist es eine gute Idee, Eingabeparameter eher per Referenz als per Wert zu übergeben.

Den Eingabeparameter quantity in Listing 4.27 könnten Sie wie in Listing 4.28 per Wert übergeben.

```
METHODS add_item
  IMPORTING
    REFERENCE(product) TYPE REF TO product
    VALUE(quantity)    TYPE quantity.
```

Listing 4.28 Eingabeparameter, der per Wert übergeben wird

Denken Sie an die Besonderheiten bei EXPORTING und erwägen Sie eine Übergabe per Wert

Parameter, die per Referenz übergeben werden, zeigen auf einen Speicherbereich, der bereits vor dem Methodenaufruf existiert und zum Zeitpunkt des Aufrufs deshalb bereits Werte beinhalten kann. Dies gilt auch für EXPORTING-Parameter, die sich dadurch tatsächlich so wie CHANGING-Parameter verhalten, siehe auch Abschnitt 4.2.7, »CHANGING-Parameter«.

Damit ein solcher EXPORTING-Parameter ein reiner *Ausgabe*parameter wird, muss ihm innerhalb der Methode unbedingt ein Wert zugewiesen werden. Eine gängige Vorgehensweise ist es, diese Parameter sofort am Beginn der Methode mit CLEAR zu leeren. Auf diese Weise vermeiden Sie, dass Methoden, die einen verzweigten Kontrollfluss haben oder mit Ausnahmen vorzeitig abbrechen können, versehentlich enden, ohne dass dem Ausgabeparameter ein Wert zugewiesen wurde.

Eine andere Möglichkeit ist es, EXPORTING-Parameter per Wert zu übergeben. Diese Kombination verhält sich wie ein RETURNING-Parameter. Bei diesen beiden Formen werden die Parameter mit einem neu angelegten und damit leeren Speicherbereich übergeben. Sowohl bei EXPORTING per Wert als auch bei RETURNING gibt es daher keine Notwendigkeit, sicherheitshalber mit CLEAR zu arbeiten. Erwägen Sie daher, die EXPORTING-Übergabe per Wert als Standard zu wählen.

In einigen Fällen kann die Übergabe per Wert zu einer schlechteren Performance führen. Wenn Sie beispielsweise extrem große Tabellen abschnittweise verarbeiten, erfordert die Übergabe per Wert, dass immer wieder größere Abschnitte der Tabellen kopiert werden, was zusätzlichen Speicher belegt. Bei der Übergabe per Referenz können Ober- und Untermethoden auf derselben internen Tabelle arbeiten, sodass kein Mehrverbrauch an Speicher anfällt. Diese Fälle sind jedoch eher selten und wir raten Ihnen wie bei allen Performancefragen dazu, erst ein sauberes Design anzustreben und nur bei nachweisbaren Problemen davon abzuweichen.

Listing 4.29 gibt ein Beispiel für EXPORTING-Parameter.

```
METHODS try_parse_int
  IMPORTING
    text    TYPE string
  EXPORTING
    success TYPE abap_bool
    result  TYPE int4.
```

Listing 4.29 Methode mit EXPORTING-Parametern

Die beiden EXPORTING-Parameter stehen hier ohne Zusatz, sodass ABAP sie standardmäßig per Referenz übergeben wird. Um sicherzustellen, dass sie keine alten Daten

enthalten, initialisiert die Methode `try_parse_int` sie sofort am Beginn der Methode, indem sie der einen `abap_false` als sinnvollen Standardwert gibt und die andere per `CLEAR` leert, wie in Listing 4.30 zu sehen.

```
METHOD try_parse_int.

  success = abap_false.
  clear result.

  ...

ENDMETHOD.
```

Listing 4.30 EXPORTING-Parameter initialisieren

Wenn Sie die `EXPORTING`-Parameter auf eine Übergabe per Wert ändern, wie in Listing 4.31 gezeigt, werden sie beim Starten der Methode automatisch initialisiert und sie können sich diese beiden Extrazeilen sparen.

```
METHODS try_parse_int
  IMPORTING
    text           TYPE string
  EXPORTING
    VALUE(success) TYPE abap_bool
    VALUE(result)  TYPE int4.
```

Listing 4.31 EXPORTING-Parameter mit Übergabe per Wert

Die Übergabe per Wert vermeidet einen weiteren typischen Fehler, der durch ein `CLEAR` am Beginn entstehen kann: Verwendet der Aufrufer ein- und dieselbe Variable gleichzeitig als `IMPORTING`- und `EXPORTING`-Parameter, funktioniert die Methode nicht. Sie leert dann nämlich beim Initialisieren des Ausgabeparameters versehentlich den Eingabeparameter gleich mit, und arbeitet am Ende auf leerem Input.

CHANGING-Parameter per Wert zu übergeben, ergibt wenig Sinn

`CHANGING`-Parameter dienen gleichzeitig als Eingabe und Ausgabe. Es ist zwar möglich, auch einen `CHANGING`-Parameter mit dem Zusatz `VALUE` als Übergabe per Wert zu deklarieren, dies wird im Alltag aber eher zu Verwirrung führen.

Solche Parameter behalten ihren ursprünglichen Eingabewert so lange, bis die Methode regulär beendet wird. Ändert die Methode also den Inhalt des Parameters, bricht dann aber mit einer Ausnahme ab, so enthält der Parameter den nichtaktualisierten Wert von vor der Methode. Dieses Verhalten ist nicht besonders intuitiv und kann zu unliebsamen Überraschungen führen. Wir raten Ihnen deshalb dazu, `CHANGING`-Parameter stets in ihrer ursprünglichen Rolle mit Übergabe per Referenz zu nutzen.

4.3 Methodeninhalt

Noch öfter als wir Code schreiben, lesen wir ihn. Ihnen geht es vermutlich genauso. Code sollte deshalb fürs Lesen optimiert sein. Leicht lesbare Methoden sind einfacher zu verwenden, zu ändern, anzupassen und zu testen. Es wird Ihnen leichter fallen, über solche Methoden zu sprechen, sie zu erklären, und Fehler darin zu erkennen. Dieser Abschnitt erkundet einige wichtige Richtlinien, die dafür sorgen, dass der Inhalt Ihrer Methoden klar verständlich und gleichzeitig flexibel bleibt.

4.3.1 Eine Sache tun

Damit eine Methode leicht zu lesen, testen und warten ist, sollte sie genau *eine* Aufgabe haben und diese auf die einfachste denkbare Weise erledigen.

Viele scheinbar unterschiedliche Richtlinien in diesem Buch verfolgen letztendlich dasselbe Ziel: Ihre Methoden fokussiert zu halten und sie dem Ziel, nur eine Sache zu tun, näher zu bringen. Eine Methode hat dann gute Chancen, genau eine Sache zu tun, wenn sie möglichst viele der folgenden Bedingungen erfüllt:

- Sie hat nur wenige Eingabeparameter.
- Sie hat keine Booleschen Eingabeparameter.
- Sie hat genau einen Ausgabeparameter.
- Sie löst genau eine Art von Ausnahme aus.
- Sie ist kurz.
- Ihr Inhalt bewegt sich auf demselben Abstraktionsniveau.
- Sie finden keinen Weg, sinnvoll weitere Methoden zu extrahieren.
- Der Inhalt der Methode erweckt in Ihnen nicht das Bedürfnis, ihn in Abschnitte zu untergliedern.

Schauen wir uns zur Veranschaulichung die Definition einer Methode zum Loggen von Ausnahmen an, wie in Listing 4.32 zu sehen.

```
METHODS
  log_exception
    IMPORTING
      exception_instance TYPE REF TO cx_root.
```

Listing 4.32 Definition einer Methode, die Ausnahmen loggt

Wir wollen diese Methode `log_exception` in einem Hintergrundprozess nutzen. Sie soll die auftretende Ausnahme `exception_instance` fangen, in ein Log schreiben und den Hintergrundprozess dann beenden. Unsere Log-Dateien vertragen nur eine begrenzte Anzahl von Zeichen pro Zeile. Unsere Endanwenderinnen und Endanwen-

der werden diese Logs später lesen, um mehr über die aufgetretenen Fehler zu erfahren. Unsere erste Implementierung dieser Methode sieht aus wie in Listing 4.33.

```
METHOD log_exception.
  DATA(current_exception) = exception_instance.
  WHILE current_exception IS BOUND.
    " log a header with the exception class name
    DATA(class_name) =
      cl_abap_classdescr=>get_class_name( current_exception ).
    log_line(
      |---- Exception occurred: { class_name }| ).

    " split the exception text into lines and log them
    DATA(exception_text) = current_exception->get_text( ).
    DATA(lines) = split_text_into_lines( exception_text ).
    LOOP AT lines into data(line).
      log_line( line ).
    ENDLOOP.

    " move to the previous exception
    current_exception = current_exception->previous.
    IF current_exception IS BOUND.
      log_line( `---- Exception has previous exception` ).
    ENDIF.
  ENDWHILE.
ENDMETHOD.
```

Listing 4.33 Erste Implementierung der Methode log_exception

Die Hilfsmethode `log_line` schreibt eine einzelne Zeile Text in die Log-Datei. Die Methode `split_text_into_lines` zerlegt einen längeren Text in mehrere Zeilen mit einer begrenzten Anzahl von Zeichen und gibt diese als Tabelle von Strings zurück.

Die Methode `log_exception` umfasst zwar nur eine Aufgabe, um diese zu erledigen, sind aber viele Teilschritte nötig:

1. Lege die Ausnahme in einer lokalen Variablen ab.
2. Wiederhole, solange die Variable eine Ausnahme enthält, folgende Schritte:
   - Ermittle den Namen der Ausnahmeklasse.
   - Logge einen Kopfeintrag mit dem Klassennamen.
   - Zerlege den Text der Ausnahme in Zeilen.
   - Logge diese Zeilen, Zeile für Zeile.
   - Ermittle die vorangegangene Ausnahme, und lege sie in die lokale Variable.

Listet man die Schritte derart in Umgangssprache auf, wird deutlich, wie viele Dinge diese Methode tut: Sie fragt iterativ vorangegangene Ausnahmen ab, ermittelt Klassennamen und zerlegt Text in Zeilen. Im nächsten Abschnitt werden Sie eine Richtlinie kennenlernen, die Ihnen hilft, derartige Methoden besser zu strukturieren.

4.3.2 Eine Abstraktionsstufe absteigen

Eine Methode sollte eine Geschichte erzählen, der Sie leicht folgen und die Sie mühelos verstehen können. Dazu sollte sich der Inhalt der Methode auf einem einheitlichen Abstraktionsniveau bewegen. Dieses Niveau sollte nur wenig unter dem der zu erledigenden Aufgabe liegen.

Wenn Sie nach der Lektüre des Codes einer Methode mehr Details sehen wollen, können Sie in die darin verwendeten Untermethoden abtauchen. Das können Sie immer wieder tun, bis Sie an eine von zwei natürlichen Grenzen stoßen: Entweder gelangen Sie in eine Methode, die nur noch aus grundlegenden Elementen der Programmiersprache besteht und in der es daher keine weiteren Unterebenen mehr gibt. Oder Sie gelangen an eine Stelle, an der das Interface einer anderen Komponente aufgerufen wird.

Wenn Sie an eine solche Grenze stoßen, sollten die Methoden Ihnen bis dahin Ebene für Ebene klar vermittelt haben, was sie tun und an welchen Stellen sie Aufgaben an andere Abhängigkeiten delegieren. Mehr Informationen, wie Sie solche Schnittstellen sauber entwerfen, finden Sie in Kapitel 3, »Klassen und Interfaces«.

Hier zeigt sich, wie sich wohldurchdachte Abstraktionen zu einem eleganten Gesamtkonstrukt zusammenfügen. Jeder Bestandteil des komplexen Designs ist so vom Rest entkoppelt, dass Sie immer nur den kleinen, gerade vor Ihnen liegenden Teil verstehen müssen, um einen Schritt weiter zu kommen. Sie müssen nicht allen Code lesen, um einen kleinen Teil des Systems zu verstehen.

Steigen Sie eine Abstraktionsstufe ab

Die Anweisungen in einer Methode sollten eine Abstraktionsstufe unter dem Namen der Methode liegen. Alle Anweisungen in der Methode sollten das gleiche Abstraktionsniveau haben.

Dieses Prinzip ist unter verschiedenen Namen bekannt, unter anderem als *Stepdown Rule* im Buch *Clean Code* und als *Single Level of Abstraction Principle* (SLAP) im Buch *The Productive Programmer* (O'Reilly, 2008). Das Konzept selbst ist aber vermutlich älter und taucht beispielsweise auch bereits in *Smalltalk Best Practice Patterns* (Prentice Hall, 1996) als *Composed Method Pattern* auf.

Obwohl die Richtlinie einleuchtend klingt, ist sie in der Praxis mitunter überraschend schwer umzusetzen. Die Herausforderung besteht darin zu erkennen, wo ein Abstrak-

tionsniveau aufhört und das nächste beginnt. Als Hilfestellung hat sich folgende Technik bewährt: Erklären Sie in kurzen Worten, was die Methode tut, um an ihr Ziel zu kommen. Die Schritte, die Sie dabei aufzählen, können Sie oft eins zu eins in Untermethoden übersetzen. Alles, was Sie bei Ihrer Zusammenfassung auslassen, liegt auf einem niedrigeren Abstraktionsniveau und sollte nicht in der Methode vorkommen.

Der Refactoring-Schritt »Extrahiere Methode«

Eine Methode zu extrahieren, ist einer der grundlegenden Schritte beim Refactoring, die bereits im Buch *Refactoring: Wie Sie das Design bestehender Software verbessern* (mitp, 2020) vorgestellt wurden.

Dabei schneiden Sie ein zusammenhängendes Stück Code aus einer Methode heraus, legen eine neue Methode an und fügen das Stück Code dort ein. An der Stelle, wo Sie ausgeschnitten haben, rufen Sie nun diese neue Methode auf.

Diesen Refactoring-Schritt können Sie in den ABAP Development Tools (ADT) auch per Tastenkombination automatisch ausführen lassen. Wählen Sie ein zusammenhängendes Stück Code in einer Methode aus und drücken Sie Alt+⇧+M (unter Windows) bzw. Cmd+Alt+M (unter macOS), um den Refactoring-Wizard zu starten.

Die Methode `log_exception` in Listing 4.33 ist durch Leerzeilen und Kommentare in mehrere Abschnitte zerlegt, die jeweils einzelne Teilaufgaben erledigen. Folgt man den Kommentaren, ergeben sich folgende große Schritte:

1. Logge einen Eintrag mit dem Namen der Ausnahmeklasse.
2. Zerlege den Text der Ausnahme in Zeilen und logge sie.
3. Fahre mit der vorangegangenen Ausnahme fort (und wiederhole das).

Beachten Sie, wie diese Schritte auf einer einheitlichen Abstraktionsstufe stehen. Wir wollen in dieser Methode nur grob erkennen, was zum Loggen einer Ausnahme zu tun ist. Details, wie der Name einer Ausnahmeklasse ermittelt oder wie Text in Zeilen zerlegt wird, interessieren uns hier (noch) nicht. Diese Schritte liegen auf einer anderen Abstraktionsstufe und gehören in die Untermethoden.

Decken Sie das Listing 4.33 noch einmal kurz mit der Hand ab und versuchen Sie, in kurzen Worten zusammenzufassen, was die Methode erreichen will, kommen Sie vermutlich auf eine Kurzbeschreibung wie die folgende:

1. Logge einen Kopfeintrag für die Ausnahme.
2. Logge einen Detailabschnitt für die Ausnahme.
3. Mache mit der vorangegangenen Ausnahme dasselbe.

Diese Schritte übersetzen wir nun eins zu eins in Methoden, wie in Listing 4.34 gezeigt.

```
METHOD log_exception.
  log_exception_header( exception_instance ).
  log_exception_body( exception_instance ).
  log_previous_exception( exception_instance->previous ).
ENDMETHOD.

METHOD log_exception_header.
  DATA(class_name) =
    cl_abap_classdescr=>get_class_name( exception_instance ).
  log_text(
    |---- Exception occurred: { class_name }| ).
ENDMETHOD.

METHOD log_exception_body.
  DATA(exception_text) = exception_instance->get_text( ).
  log_text( exception_text ).
ENDMETHOD.

METHOD log_previous_exception.
  CHECK exception_instance IS BOUND.
  log_text( `---- Exception has previous exception` ).
  log_exception( exception_instance ).
ENDMETHOD.

METHOD log_text.
  DATA(lines) = split_text_into_lines( text ).
  log_lines( lines ).
ENDMETHOD.

METHOD log_lines.
  LOOP AT lines INTO DATA(line).
    log_line( line ).
  ENDLOOP.
ENDMETHOD.
```

Listing 4.34 Mehrere Methoden mit einheitlichem Abstraktionsniveau

Jede dieser Methoden bewegt sich nun auf einer einheitlichen Abstraktionsstufe, die wiederum eine Stufe unter der Abstraktionsstufe der Aufgabe der jeweiligen Methode liegt.

Bei einer solchen Zerlegung werden die Methoden nicht nur leichter verständlich. Sie werden auch ganz neue Möglichkeiten im Design erkennen. Plötzlich werden weitere Klassen und Features sichtbar, die vorher im Wust der Details versteckt waren. Wenn Sie diese losen Enden weiter verfolgen und in eigenständige Konzepte überführen, bereichern Sie Ihr Projekt mit wiederverwendbaren Komponenten.

Im überarbeiteten Listing 4.34 könnten Sie z. B. erkennen, dass es nun leicht ist, neben dem Ausnahmetext noch weitere Informationen ins Log zu schreiben, z. B. wo genau im Code die Ausnahme ausgelöst wurde, oder gleich den ganzen Stack-Trace. Sie könnten sogar das Formatieren des Texts in einen eigenständigen `exception_formatter` auslagern, den Sie auch ohne Log verwenden können.

4.3.3 Methoden klein halten

ABAP ist tendenziell eine wortreiche, langatmige Sprache. In Kombination mit schlecht strukturiertem Legacy-Code führt das zu gigantischen Methoden und Funktionsbausteinen, die Tausende Zeilen Code umfassen können. Solche ausufernden Methoden sind schwer zu verstehen. Ohne Debuggen sind Sie hier verloren. Vom Ändern wollen wir erst gar nicht reden. Bereits ein Blick auf den Datendeklarationsabschnitt in Listing 4.35 zeigt, dass diese Methode vermutlich sehr viel mehr als nur eine Sache tut.

```
DATA:
  class           TYPE vseoclass,
  attributes      TYPE seoo_attributes_r,
  methods         TYPE seoo_methods_r,
  events          TYPE seoo_events_r,
  types           TYPE seoo_types_r,
  aliases         TYPE seoo_aliases_r,
  implementings   TYPE seor_implementings_r,
  inheritance     TYPE vseoextend,
  friendships     TYPE seof_friendships_r,
  typepusages     TYPE seot_typepusages_r,
  clsdeferrds     TYPE seot_clsdeferrds_r,
  intdeferrds     TYPE seot_intdeferrds_r,
  attribute       TYPE vseoattrib,
  method          TYPE vseomethod,
  event           TYPE vseoevent,
  type            TYPE vseotype,
  alias           TYPE seoaliases,
  implementing    TYPE vseoimplem,
  friendship      TYPE seofriends,
```



```
  typepusage      TYPE vseotypep,
  clsdeferrd      TYPE vseocdefer,
  intdeferrd      TYPE vseoidefer,
  new_clskey_save TYPE seoclskey.
```

Listing 4.35 Zu viele Datendeklarationen: Auftakt für eine gigantische Methode

Halten Sie Methoden klein

Wie klein genau eine Methode sein sollte, ist nicht ganz einfach in Zahlen auszudrücken. Methoden sollten so klein wie möglich sein. Allerdings auch nicht kleiner.

Merken Sie sich als Faustregel: Methoden sollten weniger als 20 Anweisungen umfassen. Optimal sind Methoden mit nur fünf Anweisungen oder weniger. Zählen Sie die *Anweisungen*, nicht die *Textzeilen*.

Wenn Sie die Richtlinien in diesem Abschnitt befolgen, z. B. die Berücksichtigung gleicher Abstraktionsstufen, und wiederholt Methoden extrahieren, werden Sie sich ganz von alleine auf das Ziel kurzer Methoden zubewegen.

Schauen Sie sich etwa noch einmal die Methode `log_exception` aus Listing 4.33 an. Auf den ersten Blick sieht sie aus, als hätte sie eine vernünftige Größe. Das spätere Listing 4.34 zeigt jedoch, dass wir sie mühelos in viele kleinere Methoden zerlegen konnten. Jede dieser Methoden war leichter verständlich und damit »sauberer« im Sinne von Clean Code. Die ursprüngliche Methode `log_exception` liest sich sehr viel leichter als eine Folge einfacher Schritte. Sie können sie verstehen, ohne in Untermethoden abtauchen zu müssen.

Schauen Sie sich zum Vergleich die Methode `log_exception_body` aus Listing 4.36 an.

```
METHOD log_exception_body.
  DATA(exception_text) = exception_instance->get_text( ).
  log_text( exception_text ).
ENDMETHOD.
```

Listing 4.36 Die Methode log_exception_body

Auch hier könnte man auf die Idee kommen, dass sie sich in weitere Untermethoden zerlegen ließe. Der Versuch könnte wie in Listing 4.37 aussehen.

```
METHOD log_exception_body.
  log_exception_text( exception_instance ).
ENDMETHOD.

METHOD log_exception_text.
```

```
  DATA(exception_text) = exception_instance->get_text( ).
  log_text( exception_text ).
ENDMETHOD.
```

Listing 4.37 Die extrahierte Methode log_exception_text ist redundant.

Eine solche Methode `log_exception_text` könnte durchaus sinnvoll sein, z. B., wenn die ursprüngliche Methode `log_exception_body` noch mehr täte, als nur den Text der Ausnahme zu loggen. In diesem Fall sieht die neue Methode jedoch exakt so aus wie die alte. Wir haben uns gewissermaßen im Kreis gedreht. Die neue Methode trägt nicht zu einem besseren Verständnis bei.

Hören Sie auf, Methoden zu extrahieren, wenn Sie sich mit den vorhandenen Methoden wohl fühlen und Sie und Ihr Team der Meinung sind, ihren Inhalt problemlos und schnell erfassen zu können.

4.3.4 Früh scheitern

Oftmals lassen sich Ausnahmen oder Bugs darauf zurückführen, dass eine Methode einen fehlerhaften Eingabeparameter enthielt. Wenn die Methode diese Situation früh erkennt und per Ausnahme abbricht, haben Sie es leichter, die Quelle des Fehlers – vermutlich die Aufrufstelle – zu identifizieren. Das kann Ihnen wertvolle Zeit bei der Analyse sparen und erlaubt es Ihnen, Programmfehler schneller zu beheben.

Scheitern Sie früh

Prüfen Sie die Eingabeparameter in einer Methode frühzeitig und lassen Sie die Methode schnellstmöglich kontrolliert abbrechen.

Diese Empfehlung richtet sich vor allem an `PUBLIC` und `PROTECTED` Methoden. Diese Methoden erhalten ihre Eingaben meist von außerhalb der Klasse. Da nicht immer klar ist, wo der Aufruf herkommt, ob also letztendlich ein selbst verschuldeter Programmierfehler oder etwas Unkontrollierbares wie fehlerhafte Benutzereingaben die Quelle sind, sollte die Methode kontrollierbar per Ausnahme abbrechen.

`PRIVATE` Methoden würden prinzipiell ebenso von Eingabeprüfungen profitieren. Allerdings können diese ausschließlich von anderen Methoden derselben Klasse aufgerufen werden. Wenn diese bereits alle Eingaben validieren, würden Sie dieselbe Eingabe wieder und wieder validieren. Sie können sich in diesen »inneren« Methoden deshalb zumeist darauf zurückbesinnen, dass fehlerhafte Eingaben bereits in den äußeren Methoden geprüft und abgefangen wurden und komplett auf Validierungen verzichten.

Späte Validierungen verschleiern nicht nur die Quelle der Fehler, sondern können auch wichtige Ressourcen verschwenden. Überhaupt nicht zu validieren, kann zu inkonsistenten Daten und unerwünschtem Verhalten führen.

Die Klasse `step_fan`, die wir Ihnen bereits in Listing 4.8 gezeigt hatten, validiert den Eingabeparameter `step` ihres Konstruktors nicht. Dieser Wert wird verwendet, um die Intensität des Ventilators nach oben oder unten zu regulieren. Negative Werte oder Null ergeben hier ebenso wenig Sinn wie ein Wert, der größer als die maximal mögliche Intensität ist. Der Code sollte den möglichen Wertebereich validieren und so früh wie möglich scheitern. Hier ist das bereits im Konstruktor, wie in Listing 4.38 zu sehen.

```
METHOD constructor.
  super->constructor( ).
  IF step < 1 OR step > max_intensity.
    RAISE EXCEPTION NEW invalid_argument_exception(
      argument_name  = `step`
      argument_value = step ).
  ENDIF.
  me->step = step.
ENDMETHOD.
```

Listing 4.38 Validierung im Konstruktor

Der Konstruktor von `step_fan` ruft zunächst den Konstruktor der Oberklasse. Dann validiert er den Eingabeparameter und löst eine Ausnahme aus, falls der Wert ungültig ist. In Ausnahmen sollten Sie so viele Informationen wie möglich bereitstellen. In diesem Fall haben wir uns für eine selbst angelegte `invalid_argument_exception` entschieden, die den Namen des Parameters und seinen Wert beinhaltet.

Konzentrieren Sie sich auf den Erfolgspfad oder auf die Fehlerbehandlung

Eine Methode sollte sich auf den Erfolgspfad konzentrieren, also auf den Pfad, der auf direktem Weg zum Erreichen der Aufgabe der Methode führt. Wenn eine Methode die Fehlerbehandlung durchführen soll, sollte der Code dafür nicht von der Hauptaufgabe ablenken. Wenn die Fehlerbehandlung in der Methode zu viel Raum einnimmt, sollten Sie sie in eine eigene Methode auslagern. Mehr Informationen dazu finden Sie in Kapitel 11, »Fehlerbehandlung«.

Der überarbeitete Konstruktor wird nun vom Code für die Fehlerbehandlung dominiert. Die Methode wird lesbarer, wenn wir diesen Code in eine andere Methode auslagern, wie in Listing 4.39 zu sehen.

```
METHOD constructor.
  super->constructor( ).
  validate_step( step ).
  me->step = step.
ENDMETHOD.
```

```
METHOD validate_step.
  IF step < 1 OR step > max_intensity.
    RAISE EXCEPTION NEW invalid_argument_exception(
      argument_name  = `step`
      argument_value = step ).
  ENDIF.
ENDMETHOD.
```

Listing 4.39 Fehlerbehandlung in eigene Methode auslagern

Beachten Sie, dass derartige Validierungen für andere Entwickler und Entwicklerinnen gedacht sind, um ihnen zu helfen, fehlerhafte Aufrufstellen zu korrigieren. Endanwenderinnen und Endanwender können mit derlei Ausnahmen nichts anfangen, da sie rein technischer Natur sind. Diesen sollten Sie sprechende Rückmeldungen mit Texten, die sich des Vokabulars der Domäne, Benutzeroberfläche und Dokumentation bedienen, präsentieren. In Listing 4.26 aus Abschnitt 4.2.7, »CHANGING-Parameter«, hatten wir Ihnen Code gezeigt, der eine Liste von Entitäten validiert. Wenn dieser Code Teil einer größeren Methode ist, die mit den Entitäten arbeitet, könnte ein frühes Scheitern bedeuten, dass die Methode sofort beim ersten Auffinden eines Fehlers abbrechen sollte, nicht erst, nachdem alle Entitäten untersucht wurden. Dies können wir mit kleinen Umstellungen erreichen, was uns zu Listing 4.40 führt.

```
DATA(message_container) = NEW message_container( ).
validate_entities( message_container ).
message_container->raise_if_in_error( ).
...
```

Listing 4.40 Eine Ausnahme auslösen, wenn ein Nachrichtencontainer Fehlermeldungen enthält

In diesem Fall hat die Klasse `message_container` eine Methode, die eine Ausnahme auslöst, wenn sie eine Fehlermeldung enthält. Die Fehlermeldungen werden der Ausnahme hinzugefügt. Diese wiederum könnte dann an einer anderen Stelle im Code abgefangen werden und zur Präsentation an die Endanwenderinnen und Endanwender weitergeleitet werden. Wir könnten Sie dort z. B. in eine OData-Entität verpacken und an unsere SAPUI5-App weiterleiten, die sie der Benutzerin bzw. dem Benutzer in einem Pop-over-Control darstellt.

4.3.5 CHECK oder RETURN

Statt Ausnahmen auszulösen, können Sie Methoden oft auch einfach normal beenden, wenn die Eingabeparameter nicht passen. Genauer hingeschaut bedeutet dieses

»nicht passen« hier nicht, dass ein Programmierfehler vorliegt, sondern einfach, dass ganz regulär nichts zu tun ist.

In Listing 4.34 haben Sie in der Methode `log_previous_exception` bereits ein solches frühes Beenden kennengelernt. Wenn dort der Eingabeparameter `exception_instance` nicht gebunden ist, d. h., die Referenz nirgendwohin zeigt, kann die Methode sofort aufhören.

Ein weiteres Beispiel, wo wir hiervon profitieren könnten, ist die Hauptmethode `log_exception`. Wenn die `exception_instance` nicht gebunden ist, kann die Methode auch hier sofort aufhören, ohne weitere Schritte durchzuführen, wie in Listing 4.41 zu sehen.

```
METHOD log_exception.
  CHECK exception_instance IS BOUND.
  log_exception_header( exception_instance ).
  log_exception_body( exception_instance ).
  log_previous_exception( exception_instance->previous ).
ENDMETHOD.
```

Listing 4.41 Prüfen, ob die Ausnahmeninstanz gebunden ist

Beide Methoden benutzen die Anweisung `CHECK`. Dieses Schlüsselwort liest sich an dieser Stelle flüssig, kann aber an anderen Stellen zu Verwirrung führen. Es zeigt deutlich, dass hier etwas geprüft wird, aber nicht, was die Reaktion darauf sein wird.

Ein stets verständlicher allgemeingültiger Weg zum sofortigen Beenden ist eine `IF`-Prüfung, gefolgt von einer `RETURN`-Anweisung, wie in Listing 4.42 zu sehen.

```
METHOD log_exception.
  IF exception_instance IS NOT BOUND.
    RETURN.
  ENDIF.
  log_exception_header( exception_instance ).
  log_exception_body( exception_instance ).
  log_previous_exception( exception_instance->previous ).
ENDMETHOD.
```

Listing 4.42 Sofortiges Beenden, wenn Eingabeparameter nicht gebunden sind

Dieser Schritt fügt eine Einrückung hinzu und durchbricht damit den visuellen Fluss. Besprechen Sie am besten mit Ihrem Team, ob Sie sich mit der `CHECK`-Variante wohlfühlen oder ob Sie die `IF-RETURN`-Variante verwenden wollen.

Wie auch immer Sie sich hier entscheiden, vermeiden Sie es, `CHECK` jenseits des Initialisierungsbereichs einer Methode zu verwenden. Die Anweisung hat sehr unter-

schiedliche Auswirkungen, je nachdem, wo sie verwendet wird, was zu unerwartetem Verhalten führen kann. Beispielsweise beendet `CHECK` innerhalb einer `LOOP`-Schleife den laufenden Schleifendurchgang und fährt mit dem nächsten Schleifendurchgang fort. Viele ABAP-Entwicklerinnen und -Entwickler erwarten intuitiv, dass `CHECK` die Schleife komplett abbrechen würde. An dieser Stelle ist es z. B. nachvollziehbarer, die `IF-RETURN`-Variante zu benutzen, um die Schleife abzubrechen. Um zum nächsten Schleifendurchgang zu springen, empfiehlt sich hier eine `IF`-Bedingung, bei deren Eintreten Sie mit der eindeutigeren `CONTINUE`-Anweisung den aktuellen Schleifendurchgang beenden.

4.4 Methoden aufrufen

Wir haben bereits einige Aspekte von Methodenaufrufen in ABAP angeschaut. Wir haben z. B. bereits besprochen, dass statische Methoden mit dem Doppelpfeil `=>` aufgerufen werden, während der Einfachpfeil `->` für Instanzmethoden verwendet wird.

Sie haben auch bereits gesehen, dass Interface-Methoden direkt genannt werden, wenn die Variable auf das Interface selbst deutet. Wenn die Variable jedoch einen anderen Typ hat, müssen Sie den Namen des Interface wiederholen, gefolgt von einer Tilde `~`. Haben Sie z. B. wie in Listing 4.7 eine Variable `my_thermal_switch` mit dem Typ `thermal_switch`, können Sie die Methode `is_on` des Interface `switchable` wie folgt aufrufen:

```
DATA(is_on) = my_thermal_switch->switchable~is_on( ).
```

Klassen können Aliase definieren, um dieses Wiederholen des Interface-Namens zu vermeiden. Ein Beispiel haben Sie bereits in Abschnitt 3.1.2, »Klassen und Objekte«, gesehen. Für die Übergabe von Parametern haben Sie ebenfalls verschiedene Varianten kennengelernt. Diese richten sich danach, ob Sie `IMPORTING`-, `EXPORTING`-, `CHANGING`- oder `RETURNING`-Parameter verwenden. Im Folgenden betrachten wir nun einige weitere Richtlinien, wie Sie Methodenaufrufe möglichst einfach und verständlich gestalten.

4.4.1 Eingabeparameter übergeben

Listing 4.43 zeigt eine mögliche Definition der Methode `split_text_into_lines`, die wir in Listing 4.33 verwenden.

```
METHODS:
  split_text_into_lines
    IMPORTING
      text TYPE string
      columns_per_line TYPE int4 DEFAULT 40
    RETURNING
      VALUE(lines) TYPE string_table.
```

Listing 4.43 Definition der Methode split_text_into_lines

Wenn Sie eine Methode ausschließlich mit Eingabeparametern versorgen und ihren `RETURNING`-Parameter einer lokalen Variablen zuweisen möchten, können Sie die Methode einfach wie in Listing 4.44 gezeigt aufrufen.

```
DATA(lines) =
  split_text_into_lines(
    text             = text
    columns_per_line = 80 ).
```

Listing 4.44 Eine Methode mit IMPORTING- und RETURNING-Parameter aufrufen

Wenn Sie ausschließlich `IMPORTING`-Parameter an eine Methode übergeben und den `RETURNING`-Parameter in eine neue lokale Variable legen wollen, bietet sich die Aufrufvariante in Listing 4.45 an.

```
DATA(lines) =
  split_text_into_lines(
    EXPORTING
      text             = text
      columns_per_line = 80 ).
```

Listing 4.45 Explizite Nennung von EXPORTING

Vermeiden Sie das optionale Schlüsselwort EXPORTING

Wenn alle eingehenden Parameter einer Methode den Typ `IMPORTING` haben, ist das Schlüsselwort `EXPORTING` rein optional. Wir raten Ihnen dazu, es wegzulassen, denn es macht den Aufruf nur länger, ohne irgendwelche wissenswerte Information hinzuzufügen.

4.4.2 Ausgabeparameter empfangen

`RETURNING`-Parameter können Sie einfach einer Variablen zuweisen, wie in Listing 4.44 zu sehen.

ABAP erlaubt es allerdings auch, diese Parameter innerhalb der Klammern zu empfangen, wenn Sie das entsprechende Schlüsselwort `RECEIVING` verwenden, wie in Listing 4.46 zu sehen.

```
split_text_into_lines(
  EXPORTING
    text             = text
    columns_per_line = 80
  RECEIVING
    result           = DATA(lines) ).
```

Listing 4.46 Verwendung des Schlüsselworts RECEIVING

Vermeiden Sie das Schlüsselwort RECEIVING

Die Aufrufvariante mit `RECEIVING` ist unnötig lang und zwingt Sie, auch das eigentliche optionale `EXPORTING` für Eingabeparameter einzusetzen. Ihr Code wird einfacher lesbar, wenn Sie auf `RECEIVING` verzichten und den `RETURNING`-Parameter wie das Ergebnis einer Berechnung direkt empfangen.

Die `EXPORTING`-Parameter einer Methode werden beim Aufruf mithilfe des Schlüsselworts `IMPORTING` zugewiesen. Sie haben solche Aufrufe bereits früher gesehen, z. B. in Listing 4.19. Ein weiteres Beispiel ist in Listing 4.47 zu sehen, wo wir die Methode `try_parse_int` aus Listing 4.29 aufrufen.

```
try_parse_int(
  EXPORTING
    text    = `42`
  IMPORTING
    success = DATA(parse_succeeded)
    result  = DATA(parsed_int) ).
```

Listing 4.47 EXPORTING-Parameter mithilfe des IMPORTING-Schlüsselworts zuweisen

Wann immer Sie einen `EXPORTING`- oder `CHANGING`-Parameter benutzen, müssen Sie beim Aufruf die entsprechenden Schlüsselwörter `IMPORTING` und `CHANGING` verwenden, wie bereits in Abschnitt 4.2, »Parameter«, beschrieben.

4.4.3 Das Konstrukt CALL METHOD

ABAP kennt mit der Anweisung `CALL METHOD` einen weiteren Weg, Methoden aufzurufen. Dessen Struktur ist angelehnt an die Anweisung `CALL FUNCTION` zum Aufrufen von Funktionsbausteinen. Listing 4.48 zeigt ein Beispiel.

Diese Variante des Methodenaufrufs umschließt die Parameter nicht mit Klammern und zwingt Sie, die Schlüsselwörter `IMPORTING`, `EXPORTING`, `CHANGING` und `RECEIVING` immer zu verwenden.


```
CALL METHOD split_text_into_lines
  EXPORTING
    text   = text
  RECEIVING
    result = DATA(lines).
```

Listing 4.48 Die Verwendung von CALL METHOD

Nutzen Sie CALL METHOD nur für dynamische Aufrufe

Die Anweisung `CALL METHOD` ist nicht nur unnötig langatmig, sondern auch seit geraumer Zeit auf der Liste der obsoleten ABAP-Anweisungen zu finden. Verwenden Sie stattdessen besser die funktionsartige Variante mit Klammern, um Ihrem Code eine einheitliche und leicht lesbare Form zu geben.

Nutzen Sie `CALL METHOD` ausschließlich bei dynamischen Methodenaufrufen, also solchen, bei denen der Name der Klasse und/oder Methode erst zum Ausführungszeitpunkt feststeht, wie z. B. in Listing 4.49. Für solche dynamischen Aufrufe gibt es keine funktionsartige Notation.

```
CALL METHOD modify->(method_name)
  EXPORTING
    node           = my_bo_c=>node-item
    key            = item->key
    data           = item
    changed_fields = changed_fields.
```

Listing 4.49 CALL METHOD für dynamische Methodenaufrufe

Dynamische Methodenaufrufe wiederum sollten Sie nur dort anwenden, wo sie unumgänglich sind. Typensichere Interfaces sollten stets Ihre bevorzugte Wahl sein.

4.4.4 Optionale Parameternamen

Wenn eine Methode nur einen einzigen Eingabeparameter hat, oder mehrere optionale Eingabeparameter, von denen einer als `PREFERRED PARAMETER` gekennzeichnet ist, so kann die aufrufende Stelle den Parameternamen weglassen. Das verkürzt den Code sehr stark, geht aber bei guten Variablen- und Konstantennamen keineswegs zu Lasten der Verständlichkeit:

```
DATA(lines) = split_text_into_lines( text ).
```

Den Parameter in solchen Varianten zu nennen, ist möglich, aber oft sinnlos redundant:

```
DATA(lines) = split_text_into_lines( text = text ).
```

Manchmal lassen der Methodenname und der Wert allerdings keine Rückschlüsse zu, was da gerade wofür übergeben wird. In diesen Fällen kann es sinnvoll sein, den Parameter ausdrücklich zu nennen:

```
car->drive( speed = 50 ).
```

Nennen Sie optionale Parameter nur, wenn es die Verständlichkeit erhöht

Stellen Sie sicher, dass der Methodenaufruf verständlich ist. Nennen Sie Parameter nur, wenn es der Verständlichkeit des Codes dient. Vermeiden Sie die Wiederholung, wenn sie redundant ist und den Code nur unnötig aufbläht.

4.4.5 Selbstreferenzen

In jeder Instanzmethode stellt Ihnen ABAP mit der Selbstreferenz `me` automatisch einen Zeiger auf die aktuelle Klasseninstanz zur Verfügung. ABAP folgt damit dem Muster anderer objektorientierter Programmiersprachen, beispielsweise der Selbstreferenz `this` in Sprachen wie Java und C#.

Diese Selbstreferenz können Sie wiederum benutzen, um Methoden der aktuellen Klasse aufzurufen. Die Methode `log_exception` aus Listing 4.41 könnten Sie also unter Verwendung von `me` auch wie in Listing 4.50 schreiben.

```
METHOD log_exception.
  CHECK exception_instance IS BOUND.
  me->log_exception_header( exception_instance ).
  me->log_exception_body( exception_instance ).
  me->log_previous_exception( exception_instance->previous ).
ENDMETHOD.
```

Listing 4.50 Verwendung der Selbstreferenz me zum Aufrufen von Methoden

Vermeiden Sie me, wo nicht unbedingt nötig

Sie sollten die Selbstreferenz `me` nicht generell zum Aufrufen von Instanzkomponenten der Klasse nutzen. `me` ist oft redundant und bläht den Code auf, ohne Wesentliches zum Verständnis beizutragen.

Gelegentlich kommt es vor, dass eine lokale Variable denselben Namen trägt wie eine Instanzkomponente der Klasse. In diesem Fall müssen Sie `me` benutzen, um zu unterscheiden, ob Sie die Klassenkomponente oder die lokale Variable meinen.

Dieses »Überschatten« gleichnamiger Teile führt gerne zu unliebsamen Missverständnissen. In den meisten Methoden sollten Sie daher versuchen, solche Namenskonflikte zu vermeiden, indem Sie einfach andere passende Namen wählen. Eine Ausnahme

bilden Methoden, die ausdrücklich zum Setzen der Klassenkomponenten da sind, also Konstruktoren und Setter. Dort ist diese Gleichbenennung sogar förderlich, da sie unmissverständlich klar macht, welche Komponente gerade gesetzt wird.

Ein Beispiel sehen Sie im Konstruktor der Klasse `step_fan` in Listing 4.51.

```
METHOD constructor.
  super->constructor( ).
  validate_step( step ).
  me->step = step.
ENDMETHOD.
```

Listing 4.51 Verwenden Sie me, um lokale Variablen von Klassenkomponenten zu unterscheiden.

4.5 Zusammenfassung

In diesem Kapitel haben wir einige wichtige Richtlinien betrachtet, wie Sie Methoden in ABAP lesbar entwerfen und benutzen können. Fassen wir die Punkte noch einmal zusammen.

Wir haben mit generellen Richtlinien zum Umgang mit objektorientierten Methoden begonnen:

- Nutzen Sie Instanzmethoden als Standard.
- Rufen Sie statische Methoden nicht über Instanzvariablen auf.
- Öffentliche Instanzmethoden sollten Teil eines Interface sein.
- Lassen Sie Vorsicht walten beim Redefinieren von Methoden.

Unsere Tour durch die Welt der Parameter hat uns zu den folgenden Punkten geführt:

- Minimieren Sie die Anzahl der Eingabeparameter.
- Teilen Sie Methoden eher auf, statt optionale Parameter hinzuzufügen.
- Nutzen Sie `PREFERRED PARAMETER` eher selten.
- Splitten Sie Methoden lieber, statt Boolesche Parameter einzuführen.
- Minimieren Sie die Anzahl der `EXPORTING`-Parameter.
- Nutzen Sie `RETURNING` statt `EXPORTING`.
- Verwenden Sie `result` als Standardname für `RETURNING`-Parameter.
- Nutzen Sie `CHANGING`-Parameter eher selten.
- Vermeiden Sie es, verschiedene Arten von Ausgabeparametern zu kombinieren.

- Weisen Sie `IMPORTING`-Parametern, die per Wert übergeben werden, keine neuen Werte zu.
- Beachten Sie die Besonderheiten von `EXPORTING`-Parametern; nutzen Sie Übergabe per Wert.
- `CHANGING`-Parameter mit Übergabe per Wert sind nicht sinnvoll.

In Abschnitt 4.3, »Methodeninhalt«, haben wir uns angeschaut, wie der Inhalt von Methoden aussehen sollte:

- Eine Methode sollte exakt eine Sache tun.
- Steigen Sie eine Abstraktionsstufe ab.
- Halten Sie Methoden klein.
- Scheitern Sie früh.
- Konzentrieren Sie sich entweder auf den Erfolgspfad oder auf die Fehlerbehandlung.

Zuletzt haben Sie in diesem Kapitel verschiedene Arten kennengelernt, wie Sie in ABAP Methoden aufrufen können:

- Vermeiden Sie das optionale Schlüsselwort `EXPORTING`.
- Lassen Sie `RECEIVING` weg.
- Nutzen Sie `CALL METHOD` nur für dynamische Aufrufe.
- Nennen Sie optionale Parameter nur, wenn es die Bedeutung klarer macht.
- Lassen Sie die Selbstreferenz `me` weg, wenn möglich.

Im nächsten Kapitel werden wir uns um ein essenzielles Thema kümmern: die Namensgebung. Sie werden lernen, wie wichtig gute Namen sind und wie stark sie die Lesbarkeit von ABAP-Code verbessern können.