# Tidy First?

Mini-Refactorings für besseres Software-Design

KAPITEL 1

# Guard Clauses

Sie sehen Code wie den folgenden:

```
if (Bedingung)
    ... Code ...
```

Oder noch besser:

```
if (Bedingung)
    if (not andere Bedingung)
        ... Code ...
```

Beim Lesen verliert man sich leicht in verschachtelten Bedingungen. Räumen Sie das Ganze wie folgt auf:

```
if (not Bedingung) return
if (andere Bedingung) return
... Code ...
```

Das lässt sich leichter lesen. Der Code sagt so: »Bevor wir in die Details des Codes einsteigen, gibt es ein paar Vorbedingungen, die wir berücksichtigen müssen.«

(Aber was ist mit MuLTipLen ReTuRns? Die »Regel« bezüglich eines einzelnen Returns pro Routine stammt aus den Tagen von FORTRAN, als eine einzelne Routine mehrere Einstiegs- *und* Ausstiegspunkte haben konnte. Solcher Code ließ sich so gut wie gar nicht debuggen. Niemand konnte sagen, welche Anweisungen ausgeführt wurden. Code mit Guard Clauses lässt sich leichter analysieren, weil die Vorbedingungen explizit sind.)

Treiben Sie es mit den Guard Clauses aber nicht zu weit. Eine Routine mit sieben oder acht Guard Clauses (die mir durchaus in der freien Wildbahn schon begegnet ist) lässt sich *nicht* leichter lesen. Stattdessen muss man sich dann dringender darum kümmern, die Komplexität aufzuteilen.

Räumen Sie mit einer Guard Clause nur auf, wenn die Anforderung genau erfüllt wird:

```
if (Bedingung)
    ... der gesamte Rest des Codes in der Routine ...
```

Ich sehe Code, den ich aufräumen will, aber nicht kann:

```
if (Bedingung)
    ... Code ...
... anderer Code ...
```

Vielleicht lassen sich die ersten beiden Zeilen in eine Hilfsmethode auslagern, die dann durch eine Guard Clause aufgeräumt wird, aber gehen Sie wirklich *immer* nur in kleinen Schritten vor.

Hier ein Beispiel: *https://github.com/Bogdanp/dramatiq/pull/470*

KAPITEL 2

# Toter Code

Löschen Sie ihn. Das ist alles. Wenn der Code nicht ausgeführt wird, löschen Sie ihn einfach.

Das Löschen von totem Code kann sich erstaunlich seltsam anfühlen. Schließlich hat jemand Zeit und Aufwand in sein Schreiben investiert. Die Organisation hat dafür bezahlt. Er ist jetzt da. Man muss ihn nur aufrufen, damit er Werte schafft. Wenn wir ihn wieder benötigen, wäre es doch töricht, ihn zu löschen.

Ich überlasse es Ihnen, all die kognitiven Vorurteile herauszufinden, die ich gerade demonstriert habe.

Manchmal ist es einfach, toten Code zu identifizieren. Manchmal – insbesondere bei intensiver Nutzung von Reflection – ist es nicht so einfach. Haben Sie den Verdacht, dass Code nicht genutzt wird, treffen Sie vorbereitende Aufräummaßnahmen, indem Sie seine Verwendung loggen. Bringen Sie die vorbereitenden Aufräumereien in die Produktivumgebung und warten Sie so lange, bis Sie sicher genug sind.

Sie fragen sich vielleicht: »Und wenn wir den Code später brauchen?« Nun, dafür ist die Versionsverwaltung da. Wir löschen ja nicht wirklich etwas. Wir müssen es uns jetzt nur nicht mehr anschauen. Wenn wir (und das sind wirklich viele Wenns) 1. viel Code haben, der 2. jetzt nicht genutzt wird, den wir aber 3. in Zukunft verwenden wollen – und zwar 4. genau so, wie er ursprünglich geschrieben wurde –, und wenn er dann 5. immer noch funktioniert, können wir ihn ja zurückholen. Oder ihn neu schreiben, dieses Mal besser. Aber wenn es hart auf hart kommt, bekommen wir ihn immer wieder zurück.

Löschen Sie wie immer mit jeder Aufräumerei nur ein bisschen Code. Stellt sich dabei heraus, dass Sie falsch lagen, lässt sich die Änderung recht einfach wieder rückgängig machen (siehe Kapitel 28). »Ein bisschen« ist ein subjektiver Wert, keine harte Anzahl von Codezeilen. Es kann eine Klausel in einer Bedingung sein (weil Sie zum Beispiel sehen, dass sie sich auf true eindampfen lässt), eine Routine, eine Datei, ein Verzeichnis.

KAPITEL 5

# Lesereihenfolge

Nehmen wir an, Sie lesen eine Datei (wir können ein andermal darüber diskutieren, ob Quellcode in Dateien gehört). Sie lesen die gesamte Datei, gelangen an deren Ende, und da ist es – das Detail, das Ihnen dabei geholfen hätte, den ganzen Rest der Datei zu verstehen.

Ordnen Sie den Code in der Datei in der Reihenfolge an, in der eine Leserin oder ein Leser es beim Lesen bevorzugen würde (und denken Sie daran – auf jeden Schreibenden kommen viele Leser).

Sie sind ein Leser oder eine Leserin. Sie haben es gerade gelesen. Also wissen Sie es.

Widerstehen Sie der Versuchung, gleichzeitig irgendwelche anderen Aufräumereien anzuwenden. Vermutlich werden Sie beim Lesen über andere Details gestolpert sein, die ein Verstehen und Ändern schwerer machen, als sie sein sollten. Gehen Sie solche Details später an. Alternativ kümmern Sie sich jetzt um diese Details und verschieben das Anpassen der Lesereihenfolge auf später. Aber vermischen Sie nicht beides.

Manche Sprachen reagieren beim Deklarieren von Elementen empfindlich auf deren Reihenfolge. Ein Vertauschen der Reihenfolge beim Deklarieren der Funktionen A und B kann dann andere Ergebnisse bei der Ausführung hervorbringen. Seien Sie bei solchen Sprachen vorsichtig. Ordnen Sie vielleicht nicht die ganze Datei neu an, sondern nur die für Lesende relevanten Abschnitte.

Die eine perfekte Reihenfolge gibt es nicht. Manchmal wollen Sie erst die Primitive verstehen und dann sehen, wie sie sich zusammensetzen. Manchmal wollen Sie zuerst die API verstehen und dann die Details der Implementierung. Sie sind der Leser oder die Leserin – nutzen Sie also Ihre Einschätzung und Ihre (aktuelle) Erfahrung. Welche Reihenfolge hätten Sie gern gesehen? Geben Sie diese als Geschenk an die nächste Person, die den Code lesen wird.