DDD 4 Developers

Patterns für die Implementierung

2 Supple Design

2.1 Evans' Prinzipien

Bevor wir tiefer in die Materie einsteigen, müssen wir einen Blick auf ein Kapitel in Evans' Buch werfen, das eine besonders deutliche Brücke zu den Patterns dieses Buches bildet. Evans behandelt Fragen der Implementierung insbesondere unter dem Begriff »Supple Design«, also *flexibles* Design. Er betont, dass es keine Formel für flexibles Design gibt, führt aber einige Patterns auf, die einen Eindruck davon vermitteln sollen, wie er sich ein solches flexibles Design vorstellt. Diese Patterns, die er als »complement to deep modelling« [Evans 2004, S. 244] begreift, sind die folgenden:

- Intention-Revealing Interfaces
- Assertions
- Side-Effect-Free Functions
- Conceptual Contours
- Standalone Classes
- Closure of Operations

Wir wollen nicht alle diese Patterns im Detail beleuchten, aber um ein Gefühl für sie zu gewinnen, werden wir uns zunächst drei von ihnen näher ansehen.

Intention-Revealing Interfaces

Das »Intention-Revealing Interfaces«-Pattern besteht darin, Klassen und Operationen so zu bezeichnen, dass ihr Name ihre Wirkung und ihren Zweck bereits verrät. Das entsprechende Antipattern, das es zu zweifelhaftem Ruhm gebracht hat, ist der Methodenname `doIt()`. Die Klassen- und Methodennamen sollten die *Intention* des Entwicklers explizit machen und nicht kaschieren.

Dieses Pattern gehört freilich zu den Best Practices der Softwareentwicklung. Evans selbst verweist darauf, dass Kent Beck ein ähnli-

ches Prinzip vertreten habe. Auch Robert Martins »Bibel« des Clean Code enthält ein entsprechendes Antipattern namens »G 16: Obscured Intent« [Martin 2009, S. 295] (Beispiel im Original in Java). Er illustriert dieses Antipattern durch folgendes Beispiel:

Listing 2.1 *Obscured Intent*

```
fun m_otCalc() : Int {
    return iThsWkd * iThsRte +
      round(0.5 * iThsRte *
        max(0, iThsWkd - 400)
      ).toInt()
}
```

Diese Methode berechnet eine Überstundenzahlung, was aber ohne Martins explizite Erklärung nahezu unmöglich zu verstehen wäre. Die Intention des Entwicklers wird durch den Namen der Methode (hier insbesondere auch durch die ungarische Notation), aber auch durch die Implementierung sehr weitgehend verborgen.

*Im Folgenden werden eingeführte Begriffe, die im Domain-Driven Design eine besondere Bedeutung haben, **fett** gesetzt.*

Martins und Becks Beispiel zeigen, dass das Prinzip der Intention-Revealing Interfaces keineswegs ein exklusives DDD-Prinzip ist. Warum stellt Evans es dann so explizit vor?

Ein Teil der Antwort auf die Frage ist: weil das Prinzip im Kontext der **Ubiquitous Language** eine neue Bedeutung erlangt. Wir werden darauf später zurückkommen, doch zunächst gehen wir noch weiter auf das Supple Design ein.

Assertions

Ein weiteres Prinzip, das Evans vorstellt, ist die Verwendung von *Assertions*. Eine Assertion ist bekanntlich eine Konstatierung eines Sachverhaltes, der zu einem gegebenen Zeitpunkt gelten muss, im Programmcode. Primär dienen solche Assertions der programmatischen Garantie solcher Sachverhalte. Evans betont allerdings, dass sie zugleich als *Dokumentation* dieser Sachverhalte gedacht sind. Er erwähnt, dass einige Sprachen Assertions als Sprachkonstrukt beinhalten, konzentriert sich in seinem Beispiel allerdings hauptsächlich auf Unit Tests (in denen Assertions natürlich eine zentrale Rolle spielen).

Vaughn Vernon behandelt Assertions insbesondere im Kontext von Validierungen und geht dabei einen Schritt weiter. Im Kontext des *design-by-contract* werden Assertions, die nicht nur in Unit Tests,

sondern im Programmcode selbst stehen, relevant, wie etwa folgendes – vereinfachtes – Beispiel Vernons [Vernon 2013, S. 209] zeigt:

Listing 2.2
Assertion im Kontext von Validierungen

```
public final class EmailAddress {

    public EmailAddress(String anAddress) {
        if (anAddress == null) {
            throw new IllegalArgumentException("Address " +
                    "must not be null.");
        }
    }
}
```

Eine fachliche Invariante ist eine Regel oder ein Sachverhalt, der stets gelten muss.

Nun stellt sich auch hier unmittelbar die Frage, ob Validierungen nicht ein ganz allgemeines Prinzip der Softwareentwicklung sind und warum sie gerade im DDD-Kontext eine besondere Rolle spielen sollten. Erneut ist Teil der Antwort: weil die Vor- und Nachbedingungen, die Evans und Vernon sicherstellen wollen, sogenannte **fachliche Invarianten** sind, die im DDD-Kontext von besonderer Bedeutung sind – und auch hierauf werden wir zurückkommen.

Side-Effect-Free Functions

Ein drittes Prinzip, das Evans vorstellt, besteht darin, möglichst viel Logik des Programms in Funktionen zu verorten, die keine Seiteneffekte haben. Auch dieses Konzept wird den meisten Entwicklern als Best Practice der Softwareentwicklung an sich geläufig sein. Die Popularität der funktionalen Programmierung schuldet dem Konzept viel und »Uncle Bob« bezeichnet Seiteneffekte gar als *Lügen* [Martin 2009, S. 44]. Evans stellt Side-Effect-Free Functions in Kombination mit Intention-Revealing Interfaces und **Value Objects** – auf die wir zurückkommen werden – vor und betont die Komplexitätsreduktion und Testbarkeit. Spätestens hier wird die Frage sehr wichtig: Warum scheinen so viele der Konzepte, die Evans zur Illustration des Supple Design vorstellt, auf den ersten Blick nicht unbedingt mit DDD zu tun zu haben?

Ein Value Object repräsentiert einen Wert. Es ist unveränderlich und hat keine eigene Identität.

2.2 DDD und die Prinzipien der Softwareentwicklung

Conceptual Contours

Um die Frage aus dem vorhergehenden Abschnitt zu beantworten, werfen wir zuletzt einen Blick auf das wohl tiefste Pattern, das Evans vorstellt, das aber leider zugleich auch ein wenig vage bleibt: das »Conceptual Contours«-Pattern.

Das Beispiel, das Evans verwendet, um dieses Konzept vorzustellen, erfordert einiges Wissen über buchhalterische Fachlichkeit, daher betrachten wir ein vereinfachtes Beispiel, das sich an Evans anlehnt. Es handelt sich um das Objektgeflecht, das in Abbildung 2–1 dargestellt ist.

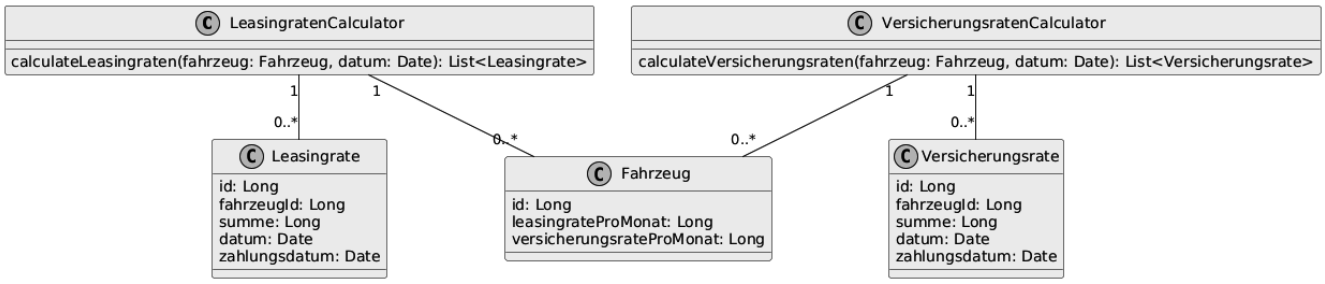

Abb. 2–1
UML-Diagramm der Raten-Kalkulation

In diesem Beispiel sind in einer bereits existierenden Anwendung, die mit der Verwaltung von Leasingfahrzeugen befasst ist, zwei Anforderungen implementiert worden, die einander sehr ähnlich sind. Einerseits werden dem Kunden für sein geleastes Fahrzeug Leasingraten in Rechnung gestellt, die von einem $LeasingratenCalculator$ berechnet werden. Andererseits muss der Kunde gegebenenfalls auch Versicherungsraten zahlen, die von einem $VersicherungsratenCalculator$ berechnet werden. In Evans' Beispiel wird nun die Logik eines Calculators so komplex, dass sie refaktoriert werden soll. Ein Gespräch zwischen Entwicklern und Fachseite führt zu einem tieferen Verständnis der Fachlichkeit und es entsteht ein nur leicht abgewandeltes Modell, das aber mehrere entscheidende Vorteile bietet (siehe Abbildung 2–2).

Im neuen Modell ist die Erkenntnis aufgehoben,

- dass eine Forderung und eine Zahlung zwei unterschiedliche Dinge sind. Das kann aus buchhalterischer Sicht bisweilen sehr wichtig sein, weil etwa für die Bilanzierung oft entscheidend ist, wann eine Forderung entstanden ist, und nicht, wann die entsprechende Zahlung tatsächlich geleistet wurde.

3.2 Pattern: 2x2=3

So viel zur abstrakten Theorie. Unser Ziel ist es aber, solche abstrakten Prinzipien konkret anwendbar zu machen. Wir entwickeln daher nun unser erstes Pattern und beginnen mit einer ersten, sehr einfachen Struktur, die uns jedoch sehr oft in der Modellierung begegnet.

Betrachten wir folgendes Codebeispiel aus dem Rechnungsstellungsmodul der Caravaggio Leasing:

```
data class Rechnung(
    ...
    val pdfErzeugt: Boolean,
    val pdfGedruckt: Boolean
)
```

Listing 3.1 *Rechnung (vereinfacht)*

Auf den ersten Blick erscheint die Modellierung klar, sparsam und prägnant. Die Domänenobjekte sind Rechnungen. Diese Rechnungen haben verschiedene Eigenschaften – selbstverständlich haben sie auch eine ID, ein Rechnungsdatum etc., die hier der Einfachheit halber weggelassen wurden. Insbesondere haben die Rechnungen aber die booleschen Eigenschaften, ob für die Rechnung ein PDF erzeugt und ob ein PDF bereits gedruckt wurde. Zu dieser Modellierung können wir auf verschiedenen Wegen gelangt sein. Möglicherweise haben wir in der Anforderungsanalyse erfahren, dass auf der UI (User Interface) für die Rechnungsbearbeitung »ein Haken angezeigt werden soll, ob das PDF schon erzeugt wurde«. Später kam dann die Anforderung, ein zweiter Haken müsse anzeigen, ob das PDF bereits gedruckt worden sei. In einem Worst-Case-Szenario haben die Domänenexperten selbst von »Flags« gesprochen, die eine Rechnung haben könne. Wie auch immer das Modell zustande gekommen ist – es ist höchstwahrscheinlich falsch. Das Problem an »Flags« ist, dass sich annähernd jeder Sachverhalt der Welt als Flag beschreiben lässt: Er trifft zu oder eben nicht. (Letztlich ist das natürlich sogar der Grund dafür, warum Entwickler überhaupt imstande sind, die Welt in Bits abzubilden.) Doch heißt das nicht, dass eine Modellierung, die jede mögliche Eigenschaft eines Objekts als Boolean abbildet, sinnvoll ist. Nun scheint für den vorliegenden Fall zunächst einmal kein Problem mit den Flags vorzuliegen: Schließlich handelt es sich tatsächlich um Eigenschaften der Rechnung, die von hoher Relevanz für den Nut-

zer sind, und die Eigenschaften sind auch »realweltlich« boolesche Zustände. Wo liegt also das Problem?

Kaffeepause 1

Es ist Zeit für eine erste Kaffeepause – und Kaffeepausen lassen sich stets gut zur Reflexion auf Designfragen nutzen. Betrachte das obige Rechnungsmodell: Was wäre ein alternatives Modell?

Die Antwort lautet: Es hat niemand die Frage gestellt, ob denn jede Konstellation der Flags überhaupt vorliegen könne. Hätte man die Frage gestellt, hätte man wohl einen verständnislosen Blick geerntet und erfahren, dass »man natürlich keine Rechnung drucken kann, die man noch nicht erzeugt hat«. Und dies ist der Grund für den Namen des »2x2=3«-Patterns: Die beiden booleschen Eigenschaften ergeben natürlich vier mögliche Kombinationen, doch in der Realität ist es schlicht nicht möglich, dass eine Rechnung zwar noch nicht gerendert, aber bereits gedruckt ist. Alle anderen Zustände konnten vorkommen (`false/false` meinte, dass die Rechnung bislang nur in Form strukturierter Daten erfasst war, aber noch nicht als PDF vorlag), nur dieser eine nicht. Dies legt nahe, dass die Modellierung zumindest suboptimal war. Man kann nun argumentieren, dass das doch nicht so schlimm sei, schließlich lässt sich die entsprechende Validierung schnell einbauen und jede Fehlersituation ist ausgeschlossen – aber das behandelt nur die Symptome und löst das eigentliche Problem nicht.

Zeichnet man den tatsächlichen Ablauf einmal auf (Abbildung 3–2), dann erkennt man schnell die eigentliche Struktur.

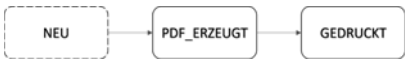

Abb. 3–2
2x2=3

Nehmen wir darüber hinaus einmal an, als nächste Anforderung käme auf, dass die UI auch als Haken anzeigen solle, ob die Rechnung bereits *versandt* wurde. Eine Nachfrage ergäbe hier, dass man natürlich keine Rechnungen versenden könne, die man nicht zuvor gerendert und gedruckt habe. Das resultierende Bild sähe aus wie in Abbildung 3–3.

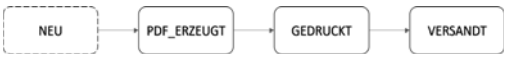

Abb. 3–3
Statusübergänge

Hier wird ersichtlich, dass das, was wir eigentlich gerade modellieren, ein klassischer Zustandsautomat (*state machine*) ist. Freilich sind

die Statusübergänge in diesem Fall sehr geradlinig – aber bereits diese Beobachtung kann uns zu der Frage verleiten, ob es einmal vorkommen kann, dass man solche Statusübergänge rückgängig machen muss. Man kann hier guten Mutes Wetten eingehen. Ist man aber einmal so weit, fällt eine angemessene Implementierung nicht mehr schwer:

Listing 3.2
Rechnung mit Status als Enum

```
data class Rechnung(
    ...
    val status: RechnungStatus
)

enum class RechnungStatus {
    NEU,
    PDF_ERZEUGT,
    GEDRUCKT,
    VERSANDT
}
```

Das wäre sicherlich eine angemessenere Modellierung. In einem nächsten Gespräch mit den Domänenexperten könnte nun geklärt werden, ob diese bereits von einem »Status«, »Zustand« oder Ähnlichem einer Rechnung sprechen, sodass gegebenenfalls das Wording angepasst werden kann. Es kann dabei vorkommen, dass den Domänenexperten selbst erst im Gespräch klar wird, dass sie, ohne es explizit getan zu haben, im Hinterkopf immer ein Statusmodell modelliert hatten, à la »wie weit die Rechnung gerade ist«. Ist das der Fall, kann die Domänenanalyse dabei helfen, implizite Annahmen und allgemein vorhandenes, aber nicht formuliertes Wissen explizit zu machen.

Fazit

Dieses erste Pattern mag noch sehr trivial erscheinen, doch die Praxis zeigt, dass die entsprechende Situation regelmäßig auftritt und dann sehr häufig zu Fehlern führt oder Feature-Erweiterungen erschwert. Was das Pattern zunächst vermitteln soll, ist, zu hinterfragen, ob alle Kombinationen von Eigenschaften tatsächlich auftreten können, oder ob hier verborgene Restriktionen vorliegen, die sich auf

den ersten Blick als Detail darstellen, in Wirklichkeit aber auf Fehler im zugrunde liegenden Modell hinweisen. Dabei weist die Formel von »2x2=3« darauf hin, dass diese Situationen in sehr unterschiedlichen Varianten auftreten können – aber stets eine strukturelle Analogie aufzeigen. »2x2=3«-Situationen zu erkennen, ist reine Übungssache, und hat man das Pattern einmal verinnerlicht, reicht es oft, im Gespräch die plakative Formel zu verwenden, um den reichlich abstrakten Sachverhalt völlig klarzumachen. Denn auf den zweiten Blick handelt es sich nicht um einen bloßen Warnhinweis, sich keine booleschen Eigenschaften von der Fachseite diktieren zu lassen. Vielmehr ist dieses Pattern eine sehr einfache Verdeutlichung eines oft zu wenig beachteten Umstands: Domänenmodellierung findet ebenso sehr in den »Kleinigkeiten« und den »Implementierungsdetails« statt wie auf architektureller Ebene. Modellierungs-Know-how für Entwickler sieht anders aus als das Know-how der Facharchitekten – es liegt im Coding vergraben.

Zuletzt ist hier der rechte Ort für eine erste Warnung: Man kann versucht sein zu argumentieren, dass doch beide Versionen – Enum und Booleans – letztlich funktionieren und dass es doch im Grunde »egal sei, wie rum man das implementiert«. Das ist zunächst nicht richtig, weil das kompliziertere Design oft Fehler auslösen wird. Außerdem verpassen wir, wenn wir die boolesche Variante wählen, den wesentlichen Punkt von DDD: die abstrakte Fachwelt korrekt zu modellieren. Das scheint in diesem einfachen Beispiel noch keine großartigen Konsequenzen zu haben, aber wir werden später Beispiele sehen, in denen die Auswirkungen immens werden.

Zweimal zwei ist manchmal drei!

3.3 Antipattern: Reified Relation

Das zweite Pattern, das wir beleuchten wollen, ist tatsächlich ein Antipattern: Es kommt vor, dass Designmuster oder Implementierungsarten sich so sehr anbieten, dass sie uns geradezu an jeder Ecke begegnen – obwohl sie vielleicht inkorrekt oder zumindest nicht für jede Situation angemessen sind. Solche Antipatterns wiederzuerkennen, ist genau so wichtig, wie ihre positiven Gegenstücke zu kennen. Dies gilt insbesondere, da es Patterns gibt, die zunächst nicht in sich falsch sind, aber die Tendenz haben, zu unglücklichen Auswüchsen zu verkommen. Das ist auch der Fall bei dem *Reified Relation*-Antipattern.