Christopher Rudoll

DDD 4 Developers

Patterns für die Implementierung

dpunkt.verlag

Vorwort

Dieses Buch ist die Summe zahlreicher (und unermüdlicher) Versuche, gute Software zu entwickeln. Es versucht Ideen zu vermitteln, die ich aus Erfolgen und Misserfolgen im Projektalltag gewonnen habe. Die Überzeugung, dass es gerade die Prinzipien des Domain-Driven Design (DDD) sind, die – auch wenn sie unabsichtlich angewandt werden – letztlich ein Chaos im Entstehen von Software verhindern, ist Ausgangspunkt des Buches geworden.

Das Schreiben von Software ist eine merkwürdige Sache. Man braucht zunächst nicht sehr viel – eine IDE (Integrated Development Environment), einen Compiler, eine kurze Anleitung zu if-Blöcken und Schleifen und schon entsteht Software. Kommen nun noch einige mächtige Frameworks hinzu, kann sehr schnell der Eindruck von Reife entstehen. Softwarearchitektur wird manchmal mit der Architektur eines Hauses verglichen. Anders aber als beim Bauen eines Hauses fällt Software, die ungeplant durch das bloße Aufeinanderschichten von Bauteilen entsteht, nicht plötzlich in sich zusammen. Auch die Wände sehen auf den ersten Blick oft gerade aus. Doch täuscht dieser Eindruck: Software, die keinem organisierenden Prinzip folgt, hat eine ähnliche Tendenz zum Kollaps – nur dass sich der Zusammenbruch von Software vor allem darin äußert, dass schlicht kein Feature mehr gebaut werden kann, ohne ein unkalkulierbares Risiko von Regression zu erzeugen. Die Kosten explodieren.

Meine Faszination für DDD hat ihren Ursprung in einer Frage, die unterschiedlich erfolgreiche Softwareprojekte aufwarfen – möglicherweise eine Frage nach dem, was später in diesem Buch der »glückliche Zufall« genannt wird: Warum scheint bei mancher Software jede neue Anforderung ein neues Problem darzustellen, während andere Software sich stets leicht um neue Funktionalität erweitern lässt und manchmal sogar erwünschte Effekte geradezu unbeabsichtigt zu erzeugen scheint? Was ist der Unterschied zwischen diesen Arten von Software? Für mich waren es am Ende die Ideen, die Evans' DDD-Buch zugrunde liegen, die mir einen Anhaltspunkt zur Beantwortung dieser Frage gaben. Software hat eine Tendenz, zum Luftschloss zu verkommen, und sie braucht eine Verankerung in der realen Welt. Domain-Driven Design stellt Prinzipien zur Verfügung, die einen solchen Anker bieten können. Von dieser Verankerung der Software in der realen Welt handelt dieses Buch.

Mein Dank gilt den Kolleginnen und Kollegen der iteratec GmbH, ohne deren Engagement und Diskussionsbereitschaft ich weder Softwareprojekte durchführen könnte, noch dieses Buch hätte schreiben können. Besonders danke ich Stefan Rauch und Wolfgang Strunk für

ihre Unterstützung, Lars Orta für die Intuitionen, Fabian Knoll für hilfreiche Anmerkungen, Andreas Feldschmid für Korrekturen und ansteckenden Enthusiasmus sowie Franziska Gilbert und Konstantin Schlagbauer für Debatten und Abstraktionen.

Darüber hinaus möchte ich dem dpunkt.verlag und insbesondere Christa Preisendanz für die Unterstützung bei diesem Buchprojekt danken. Den Reviewern und vor allem Carola Lilienthal danke ich für hilfreiche Anregungen.

Es liegt in der Natur von Büchern, dass sie als abgeschlossenes Ganzes erscheinen. Doch gerade eine Sammlung von Patterns, die aus der Praxis gewonnen wurden, kann nie völlig abgeschlossen sein. Die Praxis verändert sich, neue Ideen und Erkenntnisse kommen auf und die Herausforderungen sind im Wandel. Ich hoffe, dass dieses Buch für die Leserinnen und Leser von Nutzen sein wird – und freue mich über jedes Feedback und jeden Erfahrungsbericht, auf welchem Kanal er mich auch erreicht!

Christopher Rudoll
München, im Februar 2025

1 Einleitung

Always remember that the model is not the diagram.

— Eric Evans

1.1 Wovon dieses Buch handelt

Dieses Buch handelt von Domain-Driven Design (DDD). Genauer gesagt: davon, was Domain-Driven Design für Softwareentwickler bedeutet. Die überwiegende Mehrzahl der Studien zu DDD richtet sich (implizit oder explizit) entweder an Softwarearchitekten oder an Business-Analysten (BAs). Das ist nicht überraschend, da Domänenmodellierung und Prozessdesign – die zentralen Tätigkeiten des DDD – häufig im Rahmen dieser Rollen durchgeführt werden. Architekten und BAs sind geübt darin, die Fragen zu stellen, die entscheidende Unterschiede in der Modellierung ausmachen, und die Prinzipien des DDD erleichtern dies. Umgekehrt scheinen diese Prinzipien für Entwickler zunächst nicht unmittelbar relevant zu sein, da sie gerade in ihrer Abstraktheit keinen direkten Bezug zur täglichen Arbeit in der Implementierung aufweisen. Doch gibt es mindestens zwei Gründe, warum DDD auch für Entwickler relevant ist:

- DDD endet nicht auf der architekturellen Ebene. In der Implementierung stößt man immer wieder auf Problemstellungen, die zunächst als reine »Implementierungsdetails« erscheinen und sich bei näherer Betrachtung als Fragen von Domänenmodellierung entpuppen – und spätestens hier ist ein Verständnis von Regeln, die Architekten aus ihrer Erfahrung vertraut sind, auch für Entwickler relevant.
- Obwohl es immer noch die Regel zu sein scheint, dass Teams aus Softwareentwicklern und -architekten bestehen, deren Tätigkeiten sich zwar überschneiden, aber letztlich doch sehr unterschiedlich sind, entscheiden sich immer mehr Teams, die Rolle des »Architekten« nicht durch eine Person zu besetzen, sondern

architekturelle Arbeit im Entwicklungsteam durchführen zu lassen. Dieses Vorgehen passt sicherlich besser zu den agilen Vorgehensmodellen, die die Industrie inzwischen fast flächendeckend adaptiert hat. Es stellt aber auch neue Herausforderungen an die Mitglieder des Entwicklungsteams, wenn es nicht am Ende dazu führen soll, dass »Architektur-Storys« von De-facto-Architekten umgesetzt werden, die das nur dem Namen nach nicht mehr sind. Wenn aber umgekehrt auch architekturelle Aufgaben durch Entwickler gelöst werden sollen, ist ein Verständnis von DDD für diese zumindest sehr hilfreich.

Nun bedeutet DDD auf der Implementierungsebene nicht genau dasselbe wie auf der architekturellen Ebene. Vielmehr werden allgemeine Prinzipien hier oft sehr konkret. Dies ermöglicht es, wiederkehrende Fragestellungen zu identifizieren und sie in Patterns zu gießen, die einen hohen Wiedererkennungswert haben. Solche DDD-Patterns unterscheiden sich von den klassischen Design-Patterns dadurch, dass sie nicht so sehr darauf abzielen, ein Implementierungsproblem möglichst einfach oder effizient zu lösen. Vielmehr geht es darum, die Zieldomäne möglichst akkurat in Code zu modellieren, um Divergenzen zwischen Code und realer Welt zu vermeiden. DDD befasst sich mit der Beziehung zwischen Code und mentaler Welt der Domänenexperten (und letztlich der Endanwender) – und nicht primär mit den verschiedenen Möglichkeiten, den resultierenden Code zu strukturieren, Abhängigkeiten zu entkoppeln oder testbar zu machen.

1.2 Wer dieses Buch lesen sollte

Geschrieben wurde dieses Buch für Entwickler, die sich bereits die Frage gestellt haben, »was DDD eigentlich konkret bedeutet«. Ich werde versuchen, diese Frage so konkret wie möglich zu beantworten. DDD ist in der Implementierung genau so wichtig wie in der Architektur und die Grundsätze sauberen domänenorientierten Designs auf den Code anzuwenden, erfordert einige Übung. Die Beispiele, die ich gewählt habe, sind so einfach wie möglich gehalten, sind aber von der realen Praxis abstrahiert. Die zugrunde liegenden Prinzipien sind auf komplexere Kontexte übertragbar, auch wenn dies in einigen Fällen eine gewisse Abstraktionsleistung erfordert.

Was die BAs und Architekten betrifft, so hoffe ich, dass der Text ihnen eine interessante Perspektive eröffnet, die etwas abseits der ausgetretenen Pfade einschlägiger DDD-Einführungen liegt.

1.3 Wovon dieses Buch *nicht* handelt

Dieses Buch ist keine klassische Einführung in Domain-Driven Design. Es gibt viele gute Bücher dieser Art und ich glaube nicht, dass es ein weiteres braucht. Insbesondere handelt dieses Buch daher nicht von der Unterscheidung zwischen strategischem und taktischem Design, nicht von Event Storming und nicht einmal primär von Aggregate Roots und Repositories. Das soll nicht bedeuten, dass ich diese Dinge nicht für wichtig halte – ganz im Gegenteil! Sie sind nur nicht das Thema dieses Buches. Mir geht es zugleich um etwas Abstrakteres und etwas Konkreteres: Ich glaube, dass DDD insofern abstrakter ist, als es völlig unwichtig ist, in welcher Struktur, Architektur, Programmiersprache, Framework oder in welchen Design-Patterns es implementiert ist. Es wurde oft betont, dass DDD und funktionale Programmierung gut zusammenpassen, und ich stimme dem zu. Wahrscheinlich ließe sich DDD sogar in Assembler implementieren, wenngleich ich hoffe, dass mich niemand darum bittet, den Beweis anzutreten. Ebenso handelt dieses Buch *nicht* von hexagonaler oder »cleaner« Architektur. Es ist in den letzten Jahren in Mode gekommen, diese Gruppe von Architekturmustern so eng mit DDD zu verknüpfen, dass sie wie zwei Seiten einer Medaille erscheinen. Das ist meines Erachtens schlicht nicht der Fall. Ich glaube nicht, dass DDD eine Frage des Projektsetups ist oder sich darin entscheidet, ob vertikale Package-Strukturen oder Infrastruktur-Adapter oder gar (kein) JPA (Jakarta (früher: Java) Persistence API) verwendet wird. (Umgekehrt heißt das nicht, dass diese architekturellen Überlegungen falsch oder nur unwichtig wären, sie sind nur nicht die Punkte, um die es bei DDD primär geht.)

Andererseits glaube ich, dass es eine Seite von DDD gibt, die sich auf einem höheren Level von Konkretion abspielt, als es die meisten Einführungen erreichen: In der Implementierung tauchen immer wieder sehr ähnliche *ganz konkrete* Fragestellungen auf, die man mit einiger Erfahrung auf gemeinsame Nenner bringen kann. Um solche »gemeinsame Nenner« geht es hier und dies schlägt sich in der Struktur des Buches nieder.

1.4 Struktur

Der Text ist in Patterns unterteilt. Allerdings ist diese Art der Strukturierung mittlerweile stark abgenutzt und die Patterns sind keineswegs analog zu den bekannten Design-Patterns der *Gang of Four* [Gamma et al. 2015] zu verstehen. Sie sind insofern Patterns, als sie

wiederkehrende und wiedererkennbare Situationen beschreiben, die analoge Probleme erzeugen und deren Analyse regelmäßig zu ähnlichen Ergebnissen führt. In der Praxis habe ich gute Erfahrungen damit gemacht, solchen Patterns eingängige Namen zu geben, die dann als eine Art Abkürzung für Eingeweihte fungieren – etwa wenn ein Entwickler bei der Behandlung einer konkreten Designfrage von einem »2x2=3«-Problem spricht und alle anderen nur noch wissend nicken. Solches Erfahrungswissen lässt sich abstrahieren und kann im Idealfall verhindern, dass alte Fehler zu oft gemacht werden.

Die Codebeispiele sind in Kotlin geschrieben. In Einzelfällen werden die bekannten JPA-Annotationen verwendet. Dies soll keinesfalls suggerieren, dass JPA oder auch nur objektorientierte Sprachen wesentlich für DDD sind. Es soll auch kein anämisches Domänenmodell nahelegen und soll auch keinen Widerspruch gegen »cleane« Architekturprinzipien signalisieren. Es ist nur so, dass sowohl objektorientierte Sprachen als auch JPA sehr häufig in Business-Kontexten verwendet werden und ihre explizite Struktur eine verständliche Darstellung erleichtert.

Schließlich ist das Buch in »Levels« unterteilt, vom Anfänger bis zum Experten. Die zugrunde liegenden Patterns sind nicht etwa unterschiedlich schwer zu verstehen, die Einordnung in die Levels ergibt sich vielmehr aus den zugrunde liegenden Prinzipien. Ein Pattern anzuwenden ist das eine – zu wissen *warum*, ist etwas anderes. Die Prinzipien hinter den Patterns werden dabei zunehmend komplexer – was natürlich nicht heißen soll, dass DDD-Einsteiger nach Kapitel 3 aufhören sollten zu lesen. Ganz im Gegenteil.

1.5 Ergänzende Lektüre

In den letzten Jahren sind viele sehr gute Bücher über *Domain-Driven Design* geschrieben worden, die das Thema aus ganz unterschiedlichen Blickwinkeln beleuchten. Neben Evans' »Blue Book« [Evans 2004] und dem »Red Book« von Vernon [Vernon 2013], das sich konkret mit der Implementierung von DDD befasst, sind viele weitere Bücher als ergänzende Lektüre sehr zu empfehlen, die Aspekte beleuchten, die im Folgenden weniger im Fokus stehen. Zu nennen sind hier insbesondere die DDD-Einführung von Vlad Khononov [Khononov 2022], das Buch von Stefan Hofer und Henning Schwentner zu *Domain Storytelling* [Hofer & Schwentner 2023] sowie *Domain-Driven Transformation* von Carola Lilienthal und Henning Schwentner [Lilienthal & Schwentner 2023]. Das Thema DDD ist in letzter Zeit stark in den Fokus gerückt und obwohl viele Fragen innerhalb der DDD-

Community noch kontrovers diskutiert werden, zeichnet sich in vielen Bereichen ein Konsens ab. Ohne die aktive Diskussion der komplexen Konzepte wären die enormen Fortschritte in den Techniken des Domain-Driven Design undenkbar.

Am Ende der Kapitel finden sich jeweils Hinweise auf weiterführende Literatur zu den vorgestellten Patterns, die als Referenz genutzt werden kann.

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.

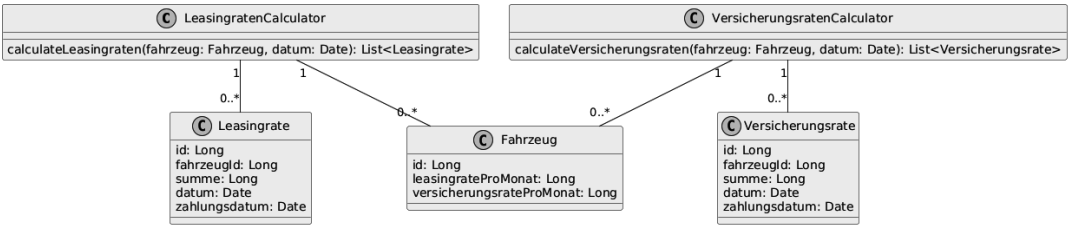

Abb. 2–1
UML-Diagramm der Raten-Kalkulation

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.

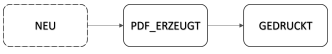

Abb. 3–2
2x2=3

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.

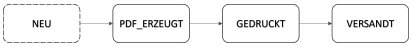

Abb. 3–3
Statusübergänge

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.

Inhaltsverzeichnis

DDD 4 Developers

- Fokus auf Implementierungsdetails von DDD für komplexe Anwendungen
- Praxisnahe Muster für die Umsetzung mit anschaulichen Beispielen
- Behandlung realer Frage- und Problemstellungen in Softwareentwicklungsprojekten

In den letzten Jahrzehnten hat sich Domain-Driven Design (DDD) als Technik der Wahl etabliert, um der enormen und zunehmenden Komplexität der Fachdomänen in der Softwaremodellierung zu begegnen.

Christopher Rudoll zeigt, was die Prinzipien des Domain-Driven Design über die zentralen Tätigkeiten der Domänenmodellierung und des Prozessdesigns hinaus in der ganz konkreten Implementierungspraxis bedeuten und wie sie sich auf Fragestellungen in der täglichen Arbeit von Softwareentwicklern anwenden lassen. Dabei wird deutlich, dass DDD nicht nur mit Event Storming und der Identifikation von Bounded Contexts zu tun hat, sondern auch in Detailfragen der Implementierung sehr hilfreiche Leitlinien bieten kann. Solche Leitlinien werden in Form von Patterns und Antipatterns anhand von Code und UML-Beispielen ausführlich erläutert.

Aus dem Inhalt:

- Supple Design – Evans' Prinzipien
- Ontologie – Abbildung der Welt in Software
- Semantik – Abbildung sprachlicher Konzepte in Software
- Konzeptuelle Räume – kognitive Grundlagen der Konzeptbildung
- Die (lästige) Realität – Fallstricke und Stolpersteine

Das Buch bietet sowohl für DDD-Enthusiasten und Softwaremodellierer als auch für Business-Analysten und Architekten neue, spannende Konzepte. Es erweitert den Werkzeugkasten eines jeden Entwicklers.

Christopher Rudoll ist Softwarearchitekt bei der iteratec GmbH in München und arbeitet an Projekten in den Branchen Automotive und Logistik. Sein Tätigkeitsfeld reicht vom Entwurf und der Umsetzung von Webanwendungen bis zur Konzeption und Implementierung komplexer Microservice-Landschaften, wobei sein besonderes Interesse den Ideen des Domain-Driven Design gilt.

€ 32,90 (D)

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

ISBN 978-3-98889-041-2