DDD 4 Developers

Patterns für die Implementierung

DAS VORWORT

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.