1 DDD für mich

Sie wollen Ihr Handwerk verbessern und Ihren Projekterfolg vergrößern? Sie brennen darauf, Ihrem Unternehmen dabei zu helfen, mit der von Ihnen entwickelten Software neue Höhen zu erklimmen? Sie wollen nicht nur Software implementieren, die die Anforderungen Ihres Business korrekt abbildet, sondern die auch skalierbar ist und die modernste Softwarearchitektur einsetzt? All das und mehr können Sie erreichen, wenn Sie *Domain-Driven Design* (DDD) lernen, je schneller desto besser.

DDD ist ein Satz von Werkzeugen, die beim Entwerfen und Implementieren von hochwertiger Software helfen, und das sowohl auf strategischer wie auch auf taktischer Ebene. Ihre Organisation kann nicht in allen Bereichen exzellent sein, deshalb sollte sie umsichtig wählen, womit sie herausstechen will. Die strategischen Werkzeuge von DDD helfen Ihnen und Ihrem Team dabei, die aus Wettbewerbssicht besten Entwurfs- und Integrationsentscheidungen für Ihr Geschäft zu treffen. Ihre Organisation wird am meisten von Softwaremodellen profitieren, die ausdrücklich ihre Kernkompetenzen reflektiert. Die taktischen Werkzeuge von DDD wiederum können Sie und Ihr Team dabei unterstützen, nützliche Software zu entwerfen, die das für den Geschäftserfolg notwendige Vorgehen exakt modelliert. Ihre Organisation sollte von den breiten Deployment-Möglichkeiten in einer Vielzahl von Infrastrukturen profitieren, sei es inhouse oder in der Cloud. Mit DDD können Sie und Ihr Team diejenigen sein, die die effektivsten Softwareentwürfe und -implementationen abliefern, wie sie in der heutigen vom harten Wettbewerb geprägten Geschäftswelt gebraucht werden.

In diesem Buch habe ich DDD für Sie zusammengefasst. Dabei habe ich die Beschreibung sowohl der strategischen als auch der taktischen Modellierungswerkzeuge auf das Wesentliche gestrafft. Mir ist bewusst, unter welchen Bedingungen Softwareentwicklung stattfindet. Außerdem kenne ich die Herausforderung, sein Handwerkszeug in einer sich rasant entwickelnden Branche zu verbessern, sehr gut. Deshalb weiß ich, dass viele von Ihnen DDD so schnell wie möglich in der täglichen Arbeit anwenden möchten, ohne mehrere Monate in das Einlesen in ein Thema wie DDD zu investieren.

Ich bin Autor des Buches *Implementing Domain-Driven Design (IDDD,* [Vernon 2013]) und führe regelmäßig den von mir entwickelten dreitägigen IDDD-Workshop durch. Und nun habe ich dieses Buch geschrieben, um DDD in einer stark verdichteten Form zu präsentieren. Das alles ist Teil meines Engagements, DDD dorthin zu bringen, wo es hingehört – in jedes Softwareentwicklungsteam. Das beinhaltet natürlich auch, DDD Ihnen nahezubringen.

Wird DDD wehtun?

Vielleicht haben Sie gehört, dass DDD ein komplizierter Ansatz zur Softwareentwicklung ist. Kompliziert? Es ist sicherlich nicht aus sich selbst heraus kompliziert. Tatsächlich besteht es aus einer Reihe von fortgeschrittenen Techniken, die für den Einsatz in komplexen Softwareprojekten vorgesehen sind. DDD ist mächtig, und man muss viel lernen, um es zu verstehen. Deshalb kann es entmutigend

sein, DDD ohne Hilfe von Experten in der Praxis einzusetzen. Wahrscheinlich haben Sie schon festgestellt, dass die meisten Bücher über DDD viele Hundert Seiten lang sind und sich keineswegs leicht lesen und anwenden lassen. Auch ich habe DDD schon sehr detailliert mit vielen Worten erläutert und ein umfassendes Implementationshandbuch über mehr als ein Dutzend DDD-Themen und -Werkzeuge geschrieben. Diese Arbeit führte zu dem Buch *Implementing Domain-Driven Design* [Vernon 2013]. Das hier vorliegende neue und verdichtete Buch hat zum Ziel, Sie so schnell und so einfach wie möglich mit den wichtigsten Teilen von DDD bekannt zu machen. Warum? Zum einen können die längeren Texte manche Leser einschüchtern. Dann hilft ein komprimierter Leitfaden für die ersten Schritte zur Anwendung. Ich habe beobachtet, dass die Anwender von DDD immer wieder in die einschlägige Literatur schauen. Die Schlussfolgerung könnte sein, dass man nie auslernt. Bei der Vertiefung Ihrer Kenntnisse kann dieses Buch folglich als schnelle Referenz dienen, während andere Bücher für tiefer gehende Informationen Verwendung finden. Außerdem kann es schwierig sein, DDD den Kollegen und dem immer wichtigen Management zu verkaufen. Dieses Buch kann dabei helfen; nicht nur, weil es DDD in einem knappen Format präsentiert, sondern auch, weil es zeigt, dass Werkzeuge vorhanden sind, um seinen Nutzen zu steigern.

Natürlich ist es nicht möglich, DDD mit diesem Buch vollständig zu vermitteln, schon weil ich mit Absicht einige DDD-Techniken weggelassen habe. Für ein deutlich tieferes Verständnis schauen Sie einfach in mein Buch *Implementing Domain-Driven Design* [Vernon 2013] und überlegen Sie sich, an meinem dreitägigen IDDD-Workshop teilzunehmen. Der dreitägige Intensivkurs, den ich bereits rund um den Globus vor einem breiten Publikum von Entwicklern gehalten habe, hilft schnell mit DDD vertraut zu werden. Außerdem biete ich DDD-Trainings online unter *http://ForComprehension.com* an.

Die gute Nachricht ist: DDD muss nicht wehtun. Da Sie wahrscheinlich in Ihren Projekten schon mit genug Komplexität zu kämpfen haben, sollten Sie DDD lernen, um die Komplexität einfacher zu beherrschen.

Gutes, schlechtes und effektives Design

Oft wird über gutes und schlechtes Design gesprochen. Welche Art von Design machen Sie? Viele Entwicklungsteams verschwenden nicht den geringsten Gedanken an Design. Stattdessen veranstalten sie das, was ich »Taskboard-Lotterie« (engl. task-board shuffle) nenne. Auf dem Taskboard hat das Team eine Liste von Entwicklungsaufgaben stehen (wie zum Beispiel ein Scrum Product Backlog) und dann verschiebt das Team einen Klebezettel aus der »To Do«-Spalte in die »In Progress«-Spalte. Ein Backlog Item erfinden und dann »Taskboard-Lotterie« spielen, ist schon alles, was an Überlegungen und Einsichten stattfindet. Der Rest wird Programmier-Heldentaten überlassen, die dann erfolgen, wenn die Entwick-

ler den Quellcode in den Rechner nur so hacken. Das Ergebnis ist selten so gut, wie es sein könnte. Die heftigste Auswirkung hat dieses nicht vorhandene Design auf das Business.

Das Ganze passiert häufig durch den hohen Druck, neue Versionen der Software unter einem unerbittlichen Zeitplan abliefern zu müssen. Dieser Zeitdruck entsteht, wenn das Management Scrum in erster Linie zur Kontrolle von Terminen verwendet, statt einer der wichtigsten Lehren von Scrum zu folgen: dem *Wissenserwerb* (engl.: *knowledge acquisition*).

Wo auch immer ich bei Unternehmen berate oder lehre, finde ich mehr oder weniger die gleiche Situation vor. Softwareprojekte sind in Gefahr und ganze Teams werden eingestellt, um die Systeme am Leben und Laufen zu halten; Code und Daten werden täglich geflickt. Daraus entstehen die folgenden heimtückischen Probleme, die interessanterweise durch den Einsatz von DDD vermieden werden können. Ich beginne mit den übergeordneten Geschäftsproblemen und arbeite mich dann zu den eher technischen vor:

- Softwareentwicklung wird als Kostenstelle und nicht als Profitcenter betrachtet. Hauptgrund dafür ist, dass Computer und Software vom Management als notwendiges Übel statt als strategischer Vorteil angesehen werden. (Unglücklicherweise kann man dieses Problem nicht lösen, wenn die Unternehmenskultur zu starr ist und keine Veränderungen zulässt.)
- Entwickler sind besessen von Technologie und davon, Probleme mit Technologie zu lösen, statt sie sorgfältig zu durchdenken. Das verführt Entwickler zur ewig andauernden Jagd nach den neuesten Technologietrends und »new shiny objects«.
- Der Datenbank wird zu hohe Priorität zugesprochen und die meisten Diskussionen drehen sich um die Datenbank und das Datenmodell statt um Geschäftsprozesse und -vorgänge.
- Entwickler geben sich nicht genug Mühe, die Objekte und Operationen so zu benennen, dass die Namen zu den fachlichen Aufgaben passen, die sie erfüllen. Das führt zu einer großen Lücke zwischen dem mentalen Modell, das die Anwender haben, und der Software, die die Entwickler den Anwendern liefern.
- Das vorhergehende Problem ist in der Regel das Ergebnis von schlechter Zusammenarbeit mit dem Fachbereich. Oft verbringen Fachexperten zu viel Zeit damit, alleine an Spezifikationen zu arbeiten, die niemand nutzt oder die nur zum Teil von Entwicklern gelesen und damit berücksichtigt werden.
- Schätzungen über den Projektfortschritt werden zu oft verlangt. Sie regelmäßig zu erstellen, verschlingt signifikant viel Zeit und Aufwand, was zu einer späteren Fertigstellung der Softwareinkremente führt. Entwickler verwenden daraufhin, um Zeit zu sparen, die »Taskboard-Lotterie« statt eines durchdachten Designs. Sie erzeugen dadurch einen *Big Ball of Mud* (dt.: *große*

Matschkugel, großes verworrenes Knäuel, dazu mehr in den folgenden Kapiteln), statt die Modelle passend zur Fachlichkeit aufzutrennen.

- Entwickler platzieren Fachlogik in Komponenten, die zur Benutzerschnittstelle oder zur Datenhaltung gehören. Außerdem werden Operationen zum Speichern in die Datenbank mitten in der Fachlogik ausgeführt.
- Defekte, langsame und sich gegenseitig behindernde Datenbankabfragen blockieren die Anwender, wenn sie zeitkritische fachliche Operationen ausführen.
- Die Entwickler schaffen falsche Abstraktionen, um alle gegenwärtigen und möglichen zukünftigen Bedürfnisse mit übermäßig verallgemeinerten Lösungen abzudecken, anstatt die tatsächlichen konkreten fachlichen Bedürfnisse zu befriedigen.
- Services sind untereinander stark gekoppelt, wobei eine Operation in einem Service ausgeführt wird und dieser Service direkt eine Operation in einem anderen Service aufruft. Diese Kopplung führt oft zu zersplitterten Geschäftsprozessen und nicht abgestimmten Daten, ganz zu schweigen von Systemen, die sehr schwer zu pflegen sind.

Das alles scheint im Sinne von »kein Design führt zu billigerer Software« zu geschehen. Die Ursache ist allzu oft, dass Fachbereich und Entwickler nicht wissen, dass es eine viel bessere Alternative gibt. »Software frisst die Welt auf« (engl.: »Software is eating the world«) [Andreessen 2011], und es sollte Ihnen nicht egal sein, dass Software auch entweder Ihren Profit auffressen oder aber Ihrem Profit ein Festmahl bescheren kann.

Es ist wichtig, zu verstehen, dass die erhoffte Geldeinsparung durch Nicht-Design ein Irrweg ist. In diese vermeintlich kostengünstigere Sackgasse werden diejenigen gelockt, die dem Druck nachgeben, Software ohne gut durchdachtes Design zu produzieren. Das ist so, weil Design immer noch aus den Hirnen der einzelnen Entwickler durch ihre Finger fließt, während sie einsam und völlig ohne Anregungen von anderen (inklusive der Fachexperten) mit dem Code ringen. Ich finde, das folgende Zitat fasst die Situation gut zusammen:

> *Die Frage, ob Design nötig oder bezahlbar ist, trifft nicht den Punkt. Design ist unumgänglich. Die Alternative zu gutem Design ist schlechtes Design, nicht überhaupt kein Design.*
>
> *Book Design: A Practical Introduction von Douglas Martin [Martin 1990]*

Obwohl sich Martins Kommentar nicht auf Softwaredesign bezieht, kann man ihn direkt auf unser Handwerk übertragen, denn auch hier gibt es keinen Ersatz für gut durchdachtes Design. Lässt man in der oben beschriebenen Situation fünf Entwickler in einem Projekt arbeiten, dann wird Nicht-Design tatsächlich ein Kuddelmuddel von fünf verschiedenen Designs in einem Modell produzieren. Das heißt, man bekommt eine Mischung aus fünf verschiedenen Übersetzungen

von ausgedachten Fachsprachen, die ohne Mitwirkung von echten *Domain Experts* (dt.: *Domänenexperten*, *Fachexperten*) entwickelt werden.

Fazit: Wir modellieren immer, unabhängig davon, ob wir akzeptieren, dass wir modellieren oder nicht. So ähnlich hat sich unser Straßennetz über Jahrhunderte hin entwickelt. Die ersten antiken Straßen begannen als Trampelpfade, die irgendwann zu gut ausgetretenen Wegen wurden. Sie hatten überraschende Kurven und gabelten sich an Stellen, die nur für einige wenige von Relevanz waren. Eines Tages wurden die ausgetretenen Wege eingeebnet und gepflastert, um sie für die steigende Zahl von Reisenden bequemer zu machen. Diese behelfsmäßigen Verkehrsstraßen werden heute nicht deshalb befahren, weil sie gut entworfen wurden, sondern einfach, weil es sie gibt. Wenige unserer Zeitgenossen können nachvollziehen, warum es so unbequem ist, auf einer von ihnen zu reisen. Moderne Straßen hingegen werden am Reißbrett geplant und entworfen. Sie berücksichtigen sorgfältige Untersuchungen der Bevölkerungsdichte, der Umwelt und des vorhersehbaren Verkehrsflusses. Beide Arten von Straßen sind modelliert worden. Das eine Modell entstand ohne viel planerischen Aufwand. Das andere Modell verlangte maximalen Einsatz von Wissen und Verstand. Auch Software kann aus beiden Perspektiven modelliert werden.

Wenn Sie befürchten, dass das Produzieren von Software mit gut durchdachtem Design teuer ist, denken Sie daran, wie viel teurer es werden wird, mit einem schlechten Design zu leben oder gar es zu reparieren. Das gilt insbesondere, wenn es sich um Software handelt, mit der sich Ihre Organisation von anderen Firmen unterscheiden und damit einen erheblichen Wettbewerbsvorteil verschaffen will.

Ein Wort, das eng mit dem Wort *gut* verwandt ist, ist das Wort *effektiv*. *Effektiv* beschreibt wahrscheinlich präziser, wonach wir beim Softwareentwurf streben sollten: *effektives Design*. Effektives Design erfüllt die Anforderungen einer Firma bis zu dem Grad, der notwendig ist, damit sie sich von ihren Wettbewerbern mithilfe ihrer Software abheben kann. Effektives Design zwingt die Organisation dazu, zu erkennen, worin sie sich hervortun muss. Dann wird effektives Design eingesetzt, um die Erzeugung des korrekten Softwaremodells zu leiten.

In Scrum wird *Wissen* durch Experimentieren und gemeinsames Lernen *erworben* (engl.: *knowledge acquisition*), was auch als »Informationen kaufen[1]« bezeichnet wird [Rubin 2012]. Wissen bekommt man nie kostenlos, aber in diesem Buch zeige ich Wege auf, es sich schneller zu erarbeiten.

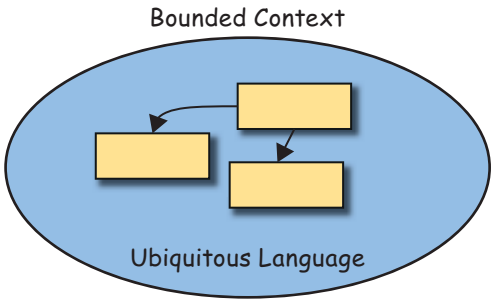
Falls Sie immer noch bezweifeln, dass effektives Design entscheidend ist, möchte ich Sie mit den Einsichten von jemandem vertraut machen, der die Wichtigkeit von effektivem Design verstanden zu haben scheint:

1. Anm. d. Übersetzer: In der deutschen Ausgabe von *Essential Scrum* [Rubin 2014] wird »buying information« mit »Informationen einholen« übersetzt, hier braucht es aber die Kauf-Metapher.

> *Die meisten Menschen machen den Fehler zu denken, dass es bei Design nur darum geht, wie es aussieht. Die Leute denken, es ist diese Fassade – dass man den Designern einen Kasten übergibt und sagt: »Macht den hübsch!« Das ist nicht unser Verständnis von Design. Es geht nicht nur darum, wie etwas aussieht und sich anfühlt. Design ist, wie etwas funktioniert.*
>
> *Steve Jobs*

Bei Software ist effektives Design am wichtigsten. Da es keine andere Alternative gibt, empfehle ich Ihnen effektives Design.

Strategisches Design



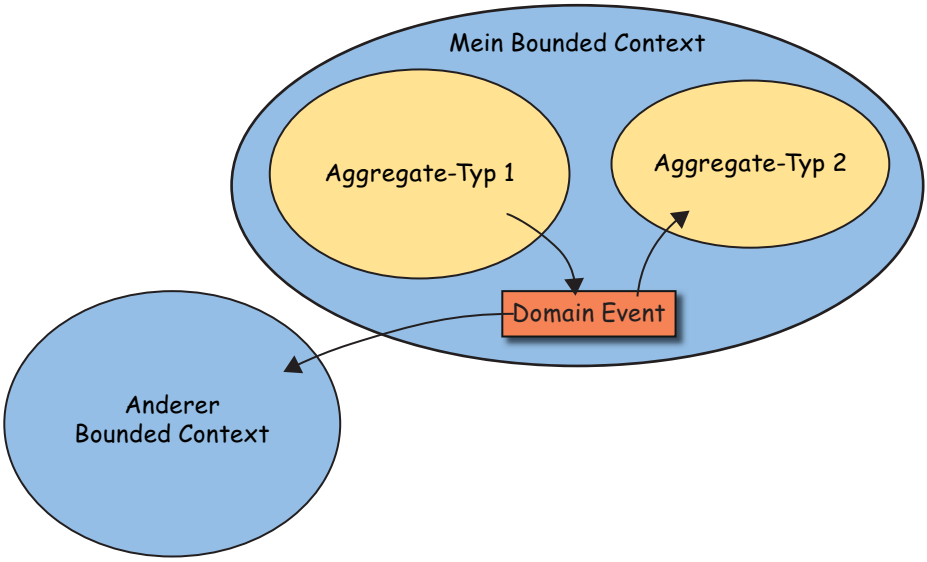
Wir fangen mit strategischem Design an, weil es besonders wichtig ist. Man kann taktisches Design nicht effektiv anwenden ohne vorheriges strategisches Design. Mit strategischem Design setzt man die groben Pinselstriche, bevor es in die Details der Implementierung geht. Strategisches Design hebt hervor, was strategisch für Ihr Geschäft wichtig ist. Wie man die Arbeit anhand von Prioritäten aufteilt und wo man am besten Dinge zusammenfasst.

Als Erstes werden Sie lernen, wie man Domänenmodelle voneinander abgrenzt. Dazu benutzt man das strategische Entwurfsmuster *Bounded Context* (dt.: *begrenzter Kontext*). Wenn Sie sich damit vertraut gemacht haben, werden wir sehen, wie eine *Ubiquitous Language* (dt.: *allgegenwärtige Sprache*) innerhalb eines expliziten *Bounded Context* entwickelt wird.

Sie werden lernen, wie wichtig es ist, nicht nur Entwickler miteinzubeziehen, sondern auch *Domain Experts*, wenn Sie die *Ubiquitous Language* Ihres Modells entwickeln. Sie werden sehen, wie ein Team von sowohl Softwareentwicklern als auch *Domain Experts* zusammenarbeitet. Diese Zusammenarbeit zwischen intelligenten und motivierten Menschen ist unerlässlich, damit DDD die besten Ergebnisse erzeugen kann. Die Sprache, die Sie durch die Zusammenarbeit gemeinsam entwickeln, wird allgegenwärtig (engl.: ubiquitous) sein, d.h., sie wird die gesamte Kommunikation im Team und das Softwaremodell durchdringen.

Während wir tiefer in das strategische Design eindringen, lernen Sie *Subdomains* (dt.: *Teildomänen*, *Subdomänen*) kennen. *Subdomains* helfen uns, mit der grenzenlosen Komplexität von Altsystemen (engl.: legacy systems) umzugehen. Sie verbessern außerdem die Ergebnisse von Projekten auf der grünen Wiese. Zusätzlich werden Sie sehen, wie man mehrere *Bounded Contexts* mithilfe einer Technik namens *Context Mapping* (dt.: *Abbilden von Kontexten*) integriert. *Context Maps* (dt.: *Kontextlandkarten*) definieren sowohl Beziehungen zwischen Teams als auch technische Mechanismen, die zwischen zwei verbundenen *Bounded Contexts* bestehen.

Taktisches Design



Nachdem Sie mit strategischem Design eine solide Grundlage erhalten haben, werden Sie die wichtigsten Werkzeuge im taktischen Design von DDD kennenlernen. Taktisches Design ist der dünne Pinselstrich, um die feinen Details des Domänenmodells zu zeichnen. Eines der wichtigeren Werkzeuge wird eingesetzt, um *Entities* (dt.: *Entitäten*, *Dinge*) und *Value Objects* (dt.: *Wertobjekte*, *Fachwerte*, *Werte*) im geeigneten Umfang zusammenzufassen – im Muster *Aggregate* (dt.: *Aggregat*).

DDD hat zum Ziel, Ihre Fachlichkeit so ausdrucksstark wie möglich zu modellieren. Die Verwendung von *Domain Events* (dt.: *Domänenereignisse*, *fachliche Ereignisse*) wird Ihnen dabei helfen, einerseits ausdrucksstark zu modellieren und andererseits Ereignisse, die in Ihrem Modell aufgetreten sind, an andere Systeme weiterzugeben, die diese Information benötigen. Die interessier-

ten anderen Systeme können der eigene lokale *Bounded Context* oder fremde entfernte *Bounded Contexts* sein.

Lernprozess und Wissensvertiefung

DDD lehrt auf eine Art und Weise zu denken, die Ihnen und Ihrem Team hilft, Ihr Wissen zu vertiefen, während Sie die Kernkompetenzen Ihres Unternehmens kennenlernen. Dieser Lernprozess erfolgt durch Konversation und Ausprobieren in einer Gruppe. Wenn Sie den Status quo und Ihre Annahmen über das Softwaremodell infrage stellen, werden Sie viel über das Modell lernen, und dieses so wichtige neu erworbene Wissen wird sich über das ganze Team verteilen. Das ist eine entscheidende Investition in Ihr Geschäft und Ihr Team. Das Ziel sollte nicht nur darin bestehen, etwas zu lernen und ein bisschen zu vertiefen, sondern so schnell wie möglich zu lernen und zu vertiefen. In Kapitel 7, »Beschleunigungs- und Managementtechniken«, finden Sie weitere Informationen, wie man diese Ziele erreichen kann.

Legen wir los!

Selbst in einer komprimierten Darstellung gibt es eine Menge über DDD zu lernen. Also legen wir los mit Kapitel 2, »Strategisches Design mit Bounded Contexts und der Ubiquitous Language«.