Eric Evans

Domain-Driven Design Referenz

Definitionen & Muster

Übersetzt von
Michael Plöd
Christian Stettler
& Eberhard Wolff

INNOQ

Domain-Driven Design Referenz

Eric Evans, übersetzt von Michael Plöd, Sonja Scheungrab, Christian Stettler und Eberhard Wolff

Dieses Buch wird verkauft unter http://leanpub.com/ddd-referenz

Diese Version wurde veröffentlicht am 2019-05-04

Dies ist ein Leanpub-Buch. Leanpub bietet Autoren und Verlagen, mit Hilfe von Lean-Publishing, neue Möglichkeiten des Publizierens. Lean Publishing bedeutet die wiederholte Veröffentlichung neuer Beta-Versionen eines eBooks unter der Zuhilfenahme schlanker Werkzeuge. Das Feedback der Erstleser hilft dem Autor bei der Finalisierung und der anschließenden Vermarktung des Buches. Lean Publishing unterstützt den Autor darin ein Buch zu schreiben, das auch gelesen wird.

This work is licensed under a Creative Commons Attribution 4.0 International License

Inhaltsverzeichnis

Vorwort der Übersetzer

Domain-driven Design spielt in unserer täglichen Arbeit beim Entwerfen und Umsetzen von fachlich anspruchsvollen Systemen eine sehr wichtige Rolle. Unserer Meinung nach ist ein grundlegendes Verständnis von DDD in jedem Projekt, das ein fachlich komplexes System umsetzen soll, notwendig. Es kann daher gar nicht genug Literatur auf Deutsch dazu geben. Eine der wichtigsten Quellen für besseres Verständnis ist die DDD-Referenz des DDD-Erfinders Eric Evans. Was liegt also näher, als sie zu übersetzen? Da das englische Werk kostenlos erhältlich ist und unter Creative-Common-Lizenz steht, war es problemlos möglich, dieses Projekt anzugehen. Das hat uns auch dazu motiviert, die Übersetzung unter dieselbe Lizenz zu stellen und kostenlos als eBook anzubieten.

Für uns ist es wichtig, dass die Übersetzung sich möglichst eng am ursprünglichen Text orientiert. So ist sichergestellt, dass die genaue Bedeutung der Muster nicht verloren geht. Für dieses Ziel mussten wir teilweise Kompromisse bei der einfachen Lesbarkeit eingehen.

Bedanken wollen wir uns bei der gesamten DDD-Gemeinde, bei Eric Evans und bei unserem Arbeitgeber INNOQ, der dieses Projekt unterstützt hat.

Wir freuen uns natürlich über Feedback - zum Beispiel als Issue im Github Projekt: https://github.com/ddd-referenz/ddd-referenz[1].

[1] https://github.com/ddd-referenz/ddd-referenz

Die Übersetzer

Michael Plöd ist Fellow bei INNOQ. Seine aktuellen Interessengebiete sind Microservices, Domain-driven Design, Alternativen zu alt eingewachsenen Softwarearchitekturen, Event Sourcing und Präsentationstechniken für Entwickler und Architekten.

Christian Stettler ist Senior Consultant und Architekt bei INNOQ. Er entwickelt fachlich relevante Systeme mit verteilten, skalierbaren Architekturen. Sein Interesse gilt dabei den Ansätzen von Domain-driven Design, um ein umfassendes fachliches Verständnis zu erlangen und dieses durchgängig in Code zu gießen.

Eberhard Wolff arbeitet als Fellow bei INNOQ und berät Kunden in Bezug auf Architekturen und Technologien. Sein Schwerpunkt liegt auf modernen Architektur-Ansätzen. Er ist Autor von über hundert Artikeln und Büchern u.a. zu Microservices[2], Microservice Technologien[3] und Continuous Delivery[4].

Cover und Review

Sonja Scheungrab arbeitet, nach Stationen bei mehreren Verlagen, bei INNOQ als Allrounderin für Kreatives. Nebenbei studiert sie Informatik an der Fernuni Hagen.

Bilder

Out at Home Plate, 1925, Library of Congress Prints & Photographs

Fingerprints, Wanted Poster, US Department of Justice

[2] http://microservices-buch.de/
[3] http://microservices-praxisbuch.de/
[4] http://continuous-delivery-buch.de/

Danksagungen

Es ist nun über zehn Jahre her, dass mein Buch “Domain-Driven Design, Tackling Complexity in the Heart of Software” (oder *The Big Blue Book* – das große blaue Buch, wie einige Leute es nennen) veröffentlicht wurde. In diesen zehn Jahren haben sich die im Buch diskutierten Grundlagen nicht wesentlich verändert, aber es hat sich *viel* daran geändert, wie wir Software entwickeln. DDD ist relevant geblieben, weil intelligente und innovative Leute die Dinge immer wieder drastisch geändert haben. Ich möchte diesen Leuten danken.

Beginnen möchte ich mit einem Dank für CQRS und Event Sourcing an Greg Young, Udi Dahan und den Leuten, die sich von ihnen haben inspirieren lassen. Das sind heute recht weit verbreitete Ansätze für die Architektur eines DDD-Systems. Dies war die erste gelungene große Abkehr von der engen Sichtweise auf die Architektur, die zur Jahrhundertwende vorherrschte.

Seitdem gab es mehrere interessante Technologien und Frameworks, die das Ziel hatten, DDD in der Implementierung zu konkretisieren (neben anderen Zielen ihrer Designer), mit unterschiedlichem Erfolg. Dazu gehören unter anderem Qi4J, Naked Objects, Roo. Auch wenn sie keine breite Verbreitung finden, sind solche Experimente sehr wertvoll.

Ich möchte auch den Leuten und der Community danken, die unser technisches Ökosystem in den letzten Jahren so revolutioniert haben, dass DDD viel mehr Spaß macht und besser anwendbar ist. Die meisten dieser Leute haben ein geringes Interesse an DDD, aber ihre Arbeit hat uns enorm geholfen. Ich denke insbesondere an die Freiheit, die NoSQL uns bringt, die kompaktere Ausdrucksweise neuer Programmiersprachen (einige davon funktional) und der unerbittliche Drang zu einfacheren technischen Frameworks und unaufdringlichen, entkoppelten Bibliotheken. Die Technologie von vor einem Jahrzehnt war kompliziert und schwergewichtig und machte DDD noch schwieriger. Es gibt natürlich auch schlechte neue Technologien, aber der Trend insgesamt ist positiv. Deshalb möchte ich mich ganz besonders bei allen

bedanken, die zu diesem Trend beigetragen haben, obwohl sie vielleicht noch nie von DDD gehört haben.

Als nächstes möchte ich denen danken, die Bücher über DDD geschrieben haben. Das erste Buch über DDD nach meinem hat Jimmy Nilsson geschrieben. Mit einem Buch hast du ein Buch. Mit zweien hast du ein Thema. Als nächstes veröffentlichte InfoQ "DDD Quickly", das aufgrund seiner Kürze, seiner Verfügbarkeit als kostenloser Download und der Reichweite von InfoQ vielen Leuten einen ersten Eindruck von diesem Thema gab. Die Jahre vergingen, und es gab viele wertvolle Blogartikel und andere kurze Artikel. Es gab auch Fachbücher wie "DDD with Naked Objects". Und ich möchte mich besonders beim unentbehrlichen Martin Fowler bedanken, der geholfen hat, die Konzepte von DDD klar zu kommunizieren und oft auch die endgültige Dokumentation der entstehenden Muster zu liefern. Erst im vergangenen Jahr veröffentlichte Vaughn Vernon das ehrgeizigste Buch seit meinem eigenen, "Implementing Domain-Driven Design" (das einige scheinbar "The Big Red Book" nennen, das große rote Buch).

Ich bedauere sehr, dass ich bestimmt viele Leute, die wichtige Beiträge geleistet haben, nicht erwähnen werde. Lassen Sie mich zumindest ein pauschales Dankeschön an die Leute richten, die DDD in die Öffentlichkeit getragen haben und an diejenigen, die DDD in die stillen Winkel von Unternehmen geschoben haben. Es braucht Tausende von Vorkämpfern, bis eine Softwarephilosophie Wirkung zeigt.

Obwohl es sich um die erste Druckausgabe der DDD-Referenz handelt, ist ihr erster Entwurf tatsächlich vor der Veröffentlichung meines Buches aus dem Jahr 2004 entstanden. Auf Anraten von Ralph Johnson habe ich die kurzen Zusammenfassungen der einzelnen Muster extrahiert und in Workshops verwendet, wobei jedes Muster von den Teilnehmern laut vorgelesen und anschließend diskutiert wurde. Ich habe diese Dokumente mehrere Jahre lang in Trainingskursen verwendet.

Dann, ein paar Jahre nachdem mein Buch veröffentlicht wurde, schlug Ward Cunningham als Teil seiner Arbeit an einem Verzeichnis von Mustern einigen Autoren vor, dass wir kurze Zusammenfassungen unserer Muster unter Creative Commons lizensieren. Martin Fowler und ich haben mit Zustimmung von Pearson Education, unserem Verleger, genau das getan, was Möglichkeiten für erweiterte Werke wie dieses eröffnet hat.

Danke an Euch alle.

Eric Evans, Juni 2014

Definitionen

Domain

Ein Bereich von Wissen, Einfluss oder Aktivität. Das Fachgebiet, auf das der Benutzer ein Programm anwendet, ist die *Domain* (Dt.: Domäne) der Software.

Model

Ein *Model* (Dt.: Modell) ist ein System von Abstraktionen, das ausgewählte Aspekte einer Domäne beschreibt und zur Lösung von Problemen im Zusammenhang mit dieser Domäne verwendet werden kann.

Ubiquitous Language

Die *Ubiquitous Language* (Dt.: allgegenwärtige Sprache) ist eine Sprache, die um das Domänenmodell herum gruppiert ist und von allen Teammitgliedern in einem *Bounded Context* verwendet wird, um alle Aktivitäten des Teams mit der Software zu verbinden.

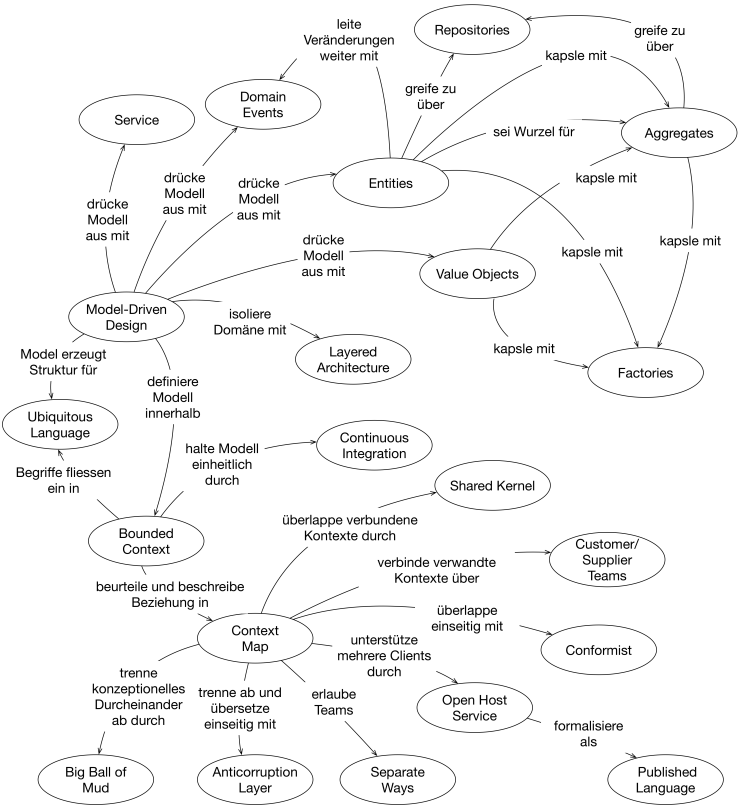
Context

Der Bereich, in dem ein Wort oder eine Aussage auftaucht und der dabei ihre Bedeutung bestimmt. *Aussagen über ein Modell können nur in einem Context (Dt.: Kontext) verstanden werden.*

Bounded Context

Der *Bounded Context* (Dt.: begrenzter Kontext) ist eine Beschreibung einer Grenze (typischerweise ein Subsystem oder die Arbeit eines bestimmten Teams), innerhalb derer ein bestimmtes Modell definiert und anwendbar ist.

Überblick über die Muster-Sprache



Überblick über die Muster-Sprache

I. Das Modell zum Einsatz bringen

Domain-driven Design ist ein Ansatz für die Entwicklung komplexer Software, bei dem wir:

1. Uns auf die Core Domain konzentrieren.
2. Modelle in einer kreativen Zusammenarbeit von Domänen-Praktikern und Software-Praktikern erkunden.
3. Eine Ubiquitous Language in einem expliziten Bounded Context sprechen.

Diese Zusammenfassung von DDD in drei Punkten stützt sich auf die Definition der Begriffe, die in dieser Broschüre definiert sind.

Viele Projekte betreiben Modellierung, ohne am Ende einen echten Nutzen zu erzielen. Die Muster von DDD stellen erfolgreiche Praktiken aus Projekten dar, bei denen die Modellierung bedeutende Vorteile gebracht hat. Zusammen angewendet stellen sie eine ganz andere Herangehensweise an Modellierung und Softwareentwicklung vor, die von kleinen Details bis hin zur übergreifenden Vision reicht. Starre Konventionen für die Modellierung müssen gegen freies Erforschen von Modellen in Zusammenarbeit mit nicht-technischen Personen abgewogen werden. Für einen Erfolg müssen Taktik und Strategie kombiniert werden; DDD befasst sich sowohl mit taktischem als auch mit strategischem Design.

Bounded Context

Dt.: Begrenzter Kontext

Bei jedem großen Projekt sind mehrere Modelle im Spiel. Sie entstehen aus vielen Gründen. Zwei Subsysteme bedienen in der Regel sehr unterschiedliche Benutzergruppen, mit unterschiedlichen Aufgaben, für die unterschiedliche Modelle sinnvoll sein können. Voneinander unabhängig arbeitende Teams lösen möglicherweise das gleiche Problem auf andere Weise, weil sie nicht miteinander kommunizieren. Auch die Werkzeuge im Einsatz sind vielleicht unterschiedlich, so dass Programmcode nicht gemeinsam genutzt werden kann.

Die Präsenz mehrerer Modelle ist unvermeidlich, aber wenn Code, der auf verschiedenen Modellen basiert, kombiniert wird, wird Software fehlerhaft, unzuverlässig und schwer verständlich und die Kommunikation zwischen den Teammitgliedern konfus. Oft ist unklar, in welchem Zusammenhang ein Modell nicht angewendet werden soll.

Ausdrücke aus dem Modell haben, wie jede andere Formulierung auch, nur im Kontext eine Bedeutung.

Daher:

Definiere explizit den Kontext, in dem ein Modell Anwendung findet. Lege explizit die Grenzen bezüglich Teamorganisation, Verwendung innerhalb bestimmter Teile der Anwendung und physischen Umsetzungen wie Codebasen und Datenbankschemata fest. Wende Continuous Integration an, um Modellkonzepte und -begriffe innerhalb dieser Grenzen strikt konsistent zu halten, lass dich aber nicht von Problemen außerhalb ablenken oder verwirren. Standardisiere einen einzigen Entwicklungsprozess innerhalb des Kontextes, der anderswo nicht verwendet werden muss.

Ubiquitous Language

Dt.: allgegenwärtige Sprache

Zuerst schreibst du einen Satz,
Und dann hackst du ihn klein;
Dann mische die Stücke und ordne sie so an
wie sie zufällig gefallen sind:
Die Reihenfolge der Ausdrücke macht
überhaupt keinen Unterschied.

— Lewis Carroll, "Poeta Fit, Non Nascitur"

Um ein Design zu schaffen, das flexibel und reich an Wissen ist, bedarf es einer vielseitigen, gemeinsam im Team genutzten Sprache und lebhaftem Experimentieren mit der Sprache, wie es in Softwareprojekten selten vorkommt.

Innerhalb eines einzigen Bounded Context kann die Sprache so kaputt sein, dass sie die Bemühungen um eine anspruchsvolle Modellierung untergräbt. Wenn das Modell nur dazu dient, UML-Diagramme für die Techniker im Team zu erstellen, dann trägt es nicht zur kreativen Zusammenarbeit, dem Kern von DDD, bei.

Domänenexperten verwenden ihren Fachjargon, während die Techniker im Team ihre eigene Sprache haben, um über die Domäne hinsichtlich Design zu diskutieren. Die Terminologie in den täglichen Diskussionen ist getrennt von der im Code verwendeten (letztendlich dem wichtigsten Produkt eines Softwareprojekts). Und selbst die gleiche Person verwendet unterschiedliche Sprachen in Wort und Schrift, so dass die prägnantesten Ausdrücke der Domäne oft in einer flüchtigen Form entstehen, die nie im Code oder auch nur schriftlich erfasst wird.

Übersetzung stört die Kommunikation und macht das Erarbeiten des Wissens blutleer.

Doch keiner dieser Dialekte kann eine gemeinsame Sprache werden, weil keiner alle Bedürfnisse erfüllt.

Domänenexperten sollten gegen Begriffe oder Strukturen protestieren, die ungeschickt oder unzureichend sind, um das Domänenverständnis zu vermitteln; Entwickler sollten auf Mehrdeutigkeit oder Inkonsistenz achten, die das Design stören.

Spielt mit dem Modell, während ihr über das System sprecht. Beschreibt Szenarien *laut* mit Hilfe der Elemente und Interaktionen aus dem Modell und kombiniert dabei Konzepte so, wie es das Modell erlaubt. Findet einfachere Wege, um zu sagen, was ihr sagen wollt, und übertragt diese neuen Ideen dann wieder zurück in die Diagramme und den Code.

Mit einer Ubiquitous Language ist das Modell nicht nur ein Design-Artefakt. Es wird zu einem integralen Bestandteil von allem, was die Entwickler und Fachexperten gemeinsam tun.

Daher:

Verwende das Modell als Rückgrat einer Sprache. Verpflichte das Team dazu, diese Sprache unermüdlich in der gesamten Kommunikation innerhalb des Teams und im Code anzuwenden. Verwende in einem Bounded Context die gleiche Sprache in Diagrammen, beim Schreiben und insbesondere beim Sprechen.

Begreife, dass eine Änderung der Sprache eine Änderung des Modells ist.

Beseitige Schwierigkeiten, indem du mit alternativen Ausdrücken experimentierst, die alternative Modelle widerspiegeln. Dann überarbeite den Code und benenne Klassen, Methoden und Module um, so dass sie dem neuen Modell entsprechen. Löse Verwirrung über Begriffe in einem Gespräch auf, und zwar in der Art und Weise, wie wir uns über die Bedeutung gewöhnlicher Wörter einigen.

Continuous Integration

Sobald ein Bounded Context definiert ist, müssen wir ihn intakt halten.

Wenn mehrere Personen am gleichen Bounded Context arbeiten, besteht eine starke Tendenz dazu, dass das Modell fragmentiert wird. Je größer das Team, desto größer das Problem, aber schon drei oder vier Personen können auf ernsthafte Probleme stoßen. Doch die Zerlegung des Systems in immer kleinere Bounded Contexts führt letztendlich dazu, das sie kein ausreichendes Maß an Integration und Kohärenz mehr haben.

Daher:

Etabliere einen Prozess, bei dem alle Code- und alle andere Implementierungsartefakte regelmäßig zusammengeführt werden, und nutze automatisierte Tests, um Fragmentierungen schnell zu finden. Wende ständig die Ubiquitous Language an, um eine gemeinsame Sicht auf das Modell auszuarbeiten, während sich die Konzepte in den Köpfen der verschiedenen beteiligten Personen weiterentwickeln.

Model-driven Design

Dt.: modellgetriebener Entwurf

Die enge Verknüpfung des Codes mit einem zugrunde liegenden Modell gibt dem Code einen Sinn und macht das Modell relevant.

Wenn das Design oder zumindest ein zentraler Teil davon nicht auf das Domänenmodell abgebildet werden kann, hat dieses Modell nur einen geringen Wert, und die Korrektheit der Software ist fraglich. Gleichzeitig sind komplexe Abbildungen zwischen Modellen und den Funktionen des Designs schwer verständlich und in der Praxis nicht wartbar, sobald sich das Design ändert. Eine tödliche Kluft öffnet sich zwischen Analyse und Design, so dass die in den beiden Aktivitäten gewonnenen Erkenntnisse nicht in die jeweils andere einfließen.

Leite aus dem Modell die beim Design verwendete Terminologie und die grundlegende Zuordnung von Verantwortlichkeiten ab. Der Code wird zu einem Ausdruck des Modells, so dass eine Änderung des Codes eine Änderung des Modells bedeuten kann. Seine Wirkung muss sich entsprechend auf den Rest der Projektarbeit auswirken.

Um die Implementierung eng an ein Modell zu binden, sind in der Regel Softwareentwicklungswerkzeuge und -sprachen erforderlich, die ein solches Modellierungsparadigma unterstützen, wie beispielsweise die objekt-orientierte Programmierung.

Daher:

Entwirf einen Teil des Softwaresystems so, dass es das Domänenmodell möglichst genau widerspiegelt, so dass die Umsetzung offensichtlich ist. Betrachte das Modell immer wieder und modifiziere es, um es natürlicher in Software zu implementieren, sogar während du versuchst, das Modell den tieferen Einblick in die Domäne besser reflektieren zu lassen. Erhebe den Anspruch, ein einziges Modell zu finden, das beiden Zwecken gerecht wird und zudem eine flüssige Ubiquitous Language unterstützt.

Hands-on Modelers

Dt.: Praktizierende Modellierer

Wenn sich die Personen, die den Code schreiben, nicht für das Modell verantwortlich fühlen oder nicht verstehen, wie das Modell für eine Anwendung funktionieren soll, dann hat das Modell mit der Software nichts zu tun. Wenn Entwickler nicht erkennen, dass Code-Änderungen das Modell verändern, dann wird ihr Refactoring das Modell eher schwächen als stärken. Wenn ein Modellierer vom Implementierungsprozess getrennt ist, gewinnt er oder sie nie ein Gefühl für die Restriktionen bei der Implementierung oder verliert dieses schnell. Die grundlegende Bedingung des Model-driven Designs - dass das Modell eine effektive Implementierung unterstützt und wichtige Erkenntnisse über die Domäne abstrahiert - geht dadurch nahezu verloren und die daraus resultierenden Modelle werden unpraktisch sein. Schließlich werden die Kenntnisse und Fähigkeiten erfahrener Designer nicht auf andere Entwickler übertragen, wenn die Arbeitsteilung die Art der Zusammenarbeit verhindert, welche die Feinheiten der Programmierung eines modellgetriebenen Entwurfs vermittelt.

Daher:

Jede technische Person, die am Modell mitwirkt, muss einige Zeit damit verbringen, am Code zu arbeiten, unabhängig davon, welche Rolle sie im Projekt primär hat. Alle, die für das Ändern von Code verantwortlich sind, müssen lernen, ein Modell durch Code auszudrücken. Jeder Entwickler muss auf der einen oder anderen Ebene an der Diskussion über das Modell beteiligt sein und Kontakt zu Domänenexperten haben. Diejenigen, die auf andere Weisen zum Projekt beitragen, müssen diejenigen, die am Code arbeiten, bewusst in einen dynamischen Austausch von Ideen zum Modell mithilfe der Ubiquitous Language einbeziehen.

Refactoring Toward Deeper Insight

Dt.: Refactoring zum tieferen Verständnis

Die Verwendung eines bewährten Satzes von Grundbausteinen zusammen mit einer konsistenten Sprache bringt ein wenig Vernunft in die Entwicklungsarbeit. Aber es bleibt eine Herausforderung, tatsächlich ein prägnantes Modell zu finden, das die subtilen Belange der Fachexperten aufgreift und ein praktisches Design vorantreiben kann. Ein Modell, welches das Oberflächliche abstreift und das Wesentliche erfasst, ist ein tiefgreifendes Modell. Dadurch sollte die Software der Denkweise der Domänenexperten besser entsprechen und besser auf die Bedürfnisse der Benutzer eingehen.

Traditionell wird Refactoring als Code-Transformation aus technischen Gründen beschrieben. Refactoring kann aber auch durch einen tieferen Einblick in die Domäne und eine entsprechende Verfeinerung des Modells oder dessen Ausdruck im Code motiviert werden.

Anspruchsvolle Domänenmodelle erweisen sich selten als nützlich, außer wenn sie durch einen iterativen Prozess mit Refactoring entwickelt werden, in der engen Zusammenarbeit von Domänenexperten mit Entwicklern, die daran interessiert sind, etwas über die Domäne zu erfahren.

II. Bestandteile eines modellgetriebenen Entwurfs

Diese Muster stellen weit verbreitete Best Practices für objekt-orientiertes Design im Hinblick auf Domain-driven Design dar. Sie leiten Entscheidungen, die das Modell präzisieren und das Modell und die Implementierung aufeinander abgestimmt halten, so dass das eine die Effektivität des anderen verstärkt. Das sorgfältige Ausarbeiten der Details der einzelnen Modellelemente gibt den Entwicklern eine solide Plattform, um Modelle zu erforschen und in enger Abstimmung mit der Implementierung zu halten.

Layered Architecture

Dt.: Geschichtete Architektur

In einem objekt-orientierten Programm werden Benutzeroberfläche, Datenbank und anderer unterstützender Code oft direkt in die Business-Objekte implementiert. Zusätzliche Geschäftslogik ist in das Verhalten von UI-Widgets und in Datenbankskripten eingebettet. Dies geschieht, weil es der einfachste Weg ist, die Dinge kurzfristig schnell zum Laufen zu bringen.

Wenn der domänenbezogene Code durch eine so große Menge an weiterem Code vermischt wird, wird es extrem schwierig, ihn zu identifizieren und zu verstehen. Scheinbar kleine Änderungen an der Benutzeroberfläche verändern möglicherweise in Wirklichkeit die Geschäftslogik. Um eine Geschäftsregel zu ändern, kann das akribische Prüfen von UI-Code, Datenbankcode oder anderen Programmteilen erforderlich sein. Die Implementierung kohärenter, modellgetriebener Objekte wird unpraktikabel. Automatisiertes Testen ist umständlich. Mit all den Technologien und der Logik, die in jede einzelne Aktivität involviert sind, muss ein Programm sehr einfach gehalten werden, oder es wird unmöglich, es zu verstehen.

Daher:

Isoliere die Umsetzung des Domänenmodells und der Geschäftslogik und eliminiere alle Abhängigkeiten zur Infrastruktur, zur Benutzeroberfläche und sogar zur Anwendungslogik, die keine Geschäftslogik ist. Unterteile ein komplexes Programm in Schichten. Entwickle innerhalb jeder Schicht ein Design, das kohärent ist und nur von den darunterliegenden Schichten abhängt. Verwende gängige Architekturmuster, um eine lose Kopplung zu den darüberliegenden Schichten zu erreichen. Konzentriere den gesamten Code, der sich auf das Domänenmodell bezieht, in einer Schicht und isoliere ihn von der Benutzeroberfläche, der Anwendungslogik und dem Infrastrukturcode. Die Domänenobjekte können sich auf die Umsetzung des Domänenmodells konzentrieren und sind

frei von der Verantwortlichkeit, sich anzuzeigen, sich zu speichern, Aufgaben in der Anwendung zu verwalten usw. Dies ermöglicht es dem Modell, sich so zu entwickeln, dass es reichhaltig genug und klar genug sein kann, um grundlegendes Geschäftswissen zu erfassen und umzusetzen.

Das Hauptziel dabei ist die Isolation. Verwandte Muster, wie z.B. die "Hexagonal Architecture", können ebenso gut oder besser dazu beitragen, dass unsere Umsetzungen des Domänenmodells Abhängigkeiten auf und Referenzen zu anderen Systembelangen vermeiden.

Entities

Dt.: Entitäten


IDENTIFICATION
ORDER NO. 1211
October 24, 1933

DIVISION OF INVESTIGATION
U. S. DEPARTMENT OF JUSTICE
WASHINGTON, D. C.

Fingerprint Classification
13 29 W MO 9
26 U 00 9

WANTED

CLYDE CHAMPION BARROW, aliases CLYDE BARROW, ELVIN WILLIAMS, ELDIN WILLIAMS, JACK HALE, ROY BAILEY.

NATIONAL MOTOR VEHICLE THEFT ACT

DESCRIPTION

Age, 23 years
Height, 5 feet, 7 inches, bare feet
Weight, 150 pounds
Build, medium; Hair, dark brown, wavy, reported dyed black; Eyes, hazel; Complexion, light; Marks, shield and anchor with U.S.N., on right forearm outer; Girl's bust, left inner forearm; Residence, West Dallas, Texas.

RELATIVES:

Mrs. Cumie F. Barrow, mother, Rural Route 6, Dallas, Texas
I. C. Barrow, brother, Rural Route 6, Dallas, Texas
Mrs. Ortis Wilber, sister, Denison, Texas
Mrs. Clyde Barrow, aliases Bonnie Barrow, Bonnie Parker, Mrs. Roy Harding, wife, probably accompanying Barrow
Mrs. Emma Parker, mother of Mrs. Clyde Barrow, Dallas, Texas.

le Clyde Barrow

CRIMINAL RECORD

As Clyde Champion Barrow, No. 6048, arrested police department, Dallas, Texas, December 3, 1926; charge, auto theft; indictment dismissed.

As Clyde Champion Barrow, No. 4316, arrested police department, Fort Worth, Texas, February 22, 1928; charge, investigation; dismissed.

As Clyde Barrow, No. 6048, arrested police department, Dallas, Texas, October 13, 1929; charge attempted burglary and safe burglary; indictment quashed.

As Clyde Barrow, alias Elvin Williams, No. 414, arrested police department, Waco, Texas, March 2, 1930; charge, burglary; sentence, 14 years Texas State Penitentiary; escaped County Jail, Waco, Texas, March 11, 1930.

As Clyde Barrow, No. 765, arrested police department Middletown, Ohio, March 18, 1930; charge, fugitive.

As Clyde Barrow, No. 63527, received Texas State Penitentiary, Huntsville, Texas, April 21, 1930, from McLennan County; charge, burglary, theft over and from the person; sentence 14 years (7-2's, cum.); paroled February 2, 1932.

This man is very dangerous and extreme caution must be exercised by arresting officers as he is wanted in connection with assault and murder of officers.

Fahndungsposter für Clyde Barrow (von Bonnie & Clyde)

Viele Objekte stellen eine Kontinuität und Identität dar und durchlaufen einen Lebenszyklus, obwohl sich ihre Attribute ändern können.

Einige Objekte sind nicht in erster Linie über ihre Attribute definiert. Sie stellen eine Identität dar, welche die Zeit und oft verschiedene Darstellungen durchläuft. Manchmal stimmt ein solches Objekt mit einem anderen Objekt überein, obwohl die Attribute unterschiedlich sind. Oder ein Objekt muss von anderen Objekten unterschieden werden, selbst wenn sie die gleichen Attribute haben können. Eine falsche Identität kann zu Datenkorruption führen.

Daher:

Wenn ein Objekt durch seine Identität und nicht durch seine Attribute unterschieden wird, dann mache das zu einem wichtigen Teil seiner Definition im Modell. Halte die Klassendefinition einfach und konzentriere dich auf die Kontinuität des Lebenszyklus und die Identität.

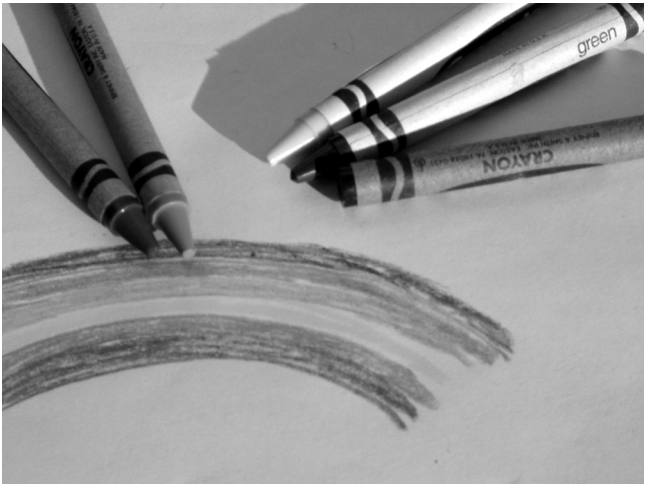
Definiere eine Möglichkeit zur Unterscheidung jedes Objekts unabhängig von seiner Form oder Geschichte. Achte auf Anforderungen, die einen Abgleich von Objekten über Attribute erfordern. Definiere eine Operation, die garantiert ein eindeutiges Ergebnis für jedes Objekt liefert, möglicherweise durch Ergänzen eines garantiert eindeutigen Attributes. Dieses Attribut zur Identifikation kann von außen kommen, oder es kann ein beliebiger Identifikator sein, der vom und für das System erstellt wird, muss aber zu den Identitätsunterschieden im Modell passen.

Das Modell muss definieren, was es bedeutet, das Gleiche zu sein.

(auch bekannt als Reference Objects, Dt.: Referenzobjekte)

Value Objects

Dt.: Wertobjekte



Einige Objekte beschreiben oder berechnen eine Eigenschaft eines Dings.

Viele Objekte haben keine konzeptionelle Identität.

Die Identität von Entities zu verfolgen ist unerlässlich, aber anderen Objekten eine Identität zuzuweisen, kann die Systemleistung beeinträchtigen, mehr Analysetätigkeiten erzwingen und das Modell durcheinander bringen, weil alle Objekte gleich aussehen. Softwaredesign ist ein ständiger Kampf gegen Komplexität. Wir müssen Unterscheidungen treffen, damit eine besondere Behandlung nur dann erfolgt, wenn sie notwendig ist.

Wenn wir diese Kategorie von Objekten jedoch nur als die Abwesenheit von Identität verstehen, haben wir nicht viel zu unserem Werkzeugkasten oder Vokabular hinzugefügt. Tatsächlich haben diese

Objekte eigene Eigenschaften und eine eigene Bedeutung für das Modell. Dies sind Objekte, die Dinge beschreiben.

Daher:

Wenn es dir nur um die Attribute und die Logik eines Elements des Modells geht, klassifiziere es als Value Object. Lass es die Bedeutung der Attribute ausdrücken, die es enthält, und gib ihm die zugehörige Funktionalität. Behandle das Value Object als unveränderlich. Mache alle Operationen zu Side-effect-free Functions, die nicht von veränderlichem Zustand abhängig sind. Gib dem Value Object keine Identität und vermeide die Komplexität, die zur Wartung von Entities notwendig sind.

Domain Event

Dt.: Domänenereignis

Drinnen oder im Aus?

Etwas ist passiert, was für Domänenexperten wichtig ist.

Eine Entity ist dafür verantwortlich, ihren Zustand und die Regeln für ihren Lebenszyklus zu erfassen. Aber wenn man die tatsächlichen Ursachen für die Änderung des Zustands kennen muss, ist das in der Regel nicht explizit gemacht, und es kann schwierig sein zu erklären, wie das System in den Zustand gekommen ist, in dem es ist. Änderungsprotokolle können eine Nachverfolgung ermöglichen, sind aber in der Regel nicht dazu geeignet, für die Logik des Programms selbst verwendet zu werden. Änderungshistorien von Entities können den Zugriff auf frühere Zustände ermöglichen, ignorieren aber die Bedeutung dieser Änderungen, so dass jede Manipulation der Informationen prozedural ist und oft aus der Domänenebene herausgedrängt wird.

In verteilten Systemen entsteht ein anderer, wenn auch verwandter Themenkomplex. Der Zustand eines verteilten Systems kann nicht immer vollständig konsistent gehalten werden. Wir halten die Ag-

gregates intern jederzeit konsistent, während wir andere Änderungen asynchron vornehmen. Da sich Änderungen über die Knoten eines Netzwerks ausbreiten, kann es schwierig sein, mehrere Änderungen zu verarbeiten, die nicht in der richtigen Reihenfolge oder aus verschiedenen Quellen kommen.

Daher:

Modelliere Informationen über die Aktivität in der Domäne als eine Reihe von diskreten Ereignissen. Stelle jedes Ereignis als Domänenobjekt dar. Diese unterscheiden sich von Systemereignissen, welche die Aktivität innerhalb der Software selbst widerspiegeln, obwohl ein Systemereignis oft mit einem Domain Event verbunden ist, entweder als Teil einer Reaktion auf das Domain Event oder als Möglichkeit, Informationen über das Domain Event in das System zu übertragen.

Ein Domain Event ist ein vollwertiger Teil des Domänenmodells, eine Darstellung von etwas, das in der Domäne passiert ist. Ignoriere irrelevante Domänenaktivitäten, während du diejenigen Ereignisse explizit festlegst, die Domänenexperten überwachen oder bei denen sie benachrichtigt werden möchten oder die mit Zustandsänderungen in den anderen Modellobjekten verbunden sind.

In einem verteilten System kann der Zustand einer Entity aus den Domain Events abgeleitet werden, die einem bestimmten Knoten derzeit bekannt sind, was ein kohärentes Modell ermöglicht, wenn keine vollständigen Informationen über das System als Ganzes vorliegen.

Domain Events sind normalerweise unveränderlich, da sie eine Aufzeichnung von etwas in der Vergangenheit sind. Zusätzlich zu einer Beschreibung des Ereignisses enthält ein Domain Event typischerweise einen Zeitstempel für den Zeitpunkt des Auftretens des Ereignisses und die Identitäten der an dem Ereignis beteiligten Entities. Außerdem hat ein Domain Event oft einen separaten Zeitstempel, der angibt, wann das Ereignis in das System gelangt ist und die Identität der Person, die es ausgelöst hat. Wenn das nützlich ist, kann eine Identität für das Domain Event auf Basis einiger dieser Eigenschaften definiert werden.

Wenn dann also beispielsweise zwei Instanzen desselben Domain Events an einem Knoten ankommen, können sie als das gleich Ereignis erkannt werden.

Domain Event ist ein neuer Begriff, der seit dem Buch von 2004 entstanden ist.

Services

Dt.: Dienste

Manchmal ist es einfach kein Ding.

Einige Konzepte aus der Domäne können nicht geeignet als Objekte modelliert werden. Der Zwang, dass die notwendige Domänenfunktionalität in die Verantwortung einer Entity oder eines Value Objects fallen soll, verzerrt entweder die Definition eines modellbasierten Objekts oder fügt bedeutungslose künstliche Objekte hinzu.

Daher:

Wenn ein wichtiger Prozess oder eine signifikante Transformation in der Domäne nicht in der natürlichen Verantwortung einer Entity oder eines Value Objects liegt, füge dem Modell eine Operation als eigenständige Schnittstelle hinzu, die als Service deklariert ist. Definiere einen Servicevertrag, eine Reihe von Assertions zu Interaktionen mit dem Service. Drücke diese Assertions in der Ubiquitous Language eines bestimmten Bounded Context aus. Gib dem Service einen Namen, der ebenfalls Teil der Ubiquitous Language wird.

Modules

Dt.: Module

Jeder verwendet Module, aber nur wenige behandeln sie als vollwertigen Teil des Modells. Code wird in alle möglichen Kategorien unterteilt, von Aspekten der technischen Architektur bis hin zu den Aufgaben der Entwickler. Selbst Entwickler, die viel refaktorisieren, begnügen sich oft mit den Modulen, die frühzeitig im Projekt entstanden sind.

Erklärungen von Kopplung und Kohäsion lassen diese Begriffe tendenziell wie technische Metriken wirken, die mechanisch anhand der Verteilungen von Beziehungen und Interaktionen beurteilt werden sollen. Dabei geht es nicht nur um die Aufteilung von Code in Module, sondern auch in Konzepte. Es gibt eine Grenze, an wie viele Dinge eine Person auf einmal denken kann (daher niedrige Kopplung). Unzusammenhängende Fragmente von Ideen sind so schwer zu verstehen wie eine undifferenzierte Ideensuppe (daher hohe Kohäsion).

Daher:

Wähle Module, welche die Geschichte des Systems erzählen und einen zusammenhängenden Satz von Konzepten enthalten. Gib den Modulen Namen, die Teil der Ubiquitous Language werden. Module sind Teil des Modells und ihre Namen sollten das Verständnis der Domäne widerspiegeln.

Dies führt oft zu einer geringen Kopplung zwischen den Modulen, und falls nicht: suche nach Wegen, um das Modell zu ändern und so die Konzepte zu entwirren, oder nach einem bisher unbemerkten Konzept, das die Grundlage für ein Modul sein kann, welches die Elemente sinnvoll zusammenbringen würde. Suche eine niedrige Kopplung im Sinne von Konzepten, die unabhängig voneinander verstanden und begründet werden können. Verfeinere das Modell, bis es nach High-Level-Domänenkonzepten partitioniert ist und auch der Code entsprechend entkoppelt ist.

(auch bekannt als Packages, Dt.: Pakete)

Aggregates

Dt.: Aggregate

Es ist schwierig, die Konsistenz von Änderungen an Objekten in einem Modell mit komplexen Beziehungen zu gewährleisten. Objekte sollen ihren eigenen internen konsistenten Zustand beibehalten, können aber bei Änderungen in anderen Objekten, die konzeptionell Bestandteile dieser ersten Objekte sind, übergangen werden. Vorsichtige Datenbank-Sperrschemata führen dazu, dass sich mehrere Benutzer sinnlos gegenseitig stören, und können ein System unbrauchbar machen. Ähnliche Probleme entstehen bei der Verteilung von Objekten auf mehrere Server, oder beim Entwurf asynchroner Transaktionen.

Daher:

Gruppiere die Entities und Value Objects zu Aggregates und definiere Grenzen um jede dieser Gruppen herum. Wähle eine Entity als Wurzel eines jeden Aggregates und erlaube externen Objekten, nur Referenzen auf die Wurzel zu halten (Referenzen auf interne Elemente werden nur für die Verwendung innerhalb einer einzigen Operation ausgegeben). Definiere Eigenschaften und Invarianten für das Aggregate als Ganzes und übertrage die Verantwortung für deren Durchsetzung an die Wurzel oder an einen ausgewiesenen Mechanismus im Framework.

Verwende die gleichen Aggregate-Grenzen für die Steuerung von Transaktionen und Verteilung.

Wende innerhalb der Grenze eines Aggregates die Konsistenzregeln synchron an. Behandle Änderungen über Grenzen hinweg asynchron.

Halte ein einzelnes Aggregate als Ganzes auf einem Server. Erlaube die Verteilung verschiedener Aggregates auf unterschiedliche Knoten.

Wenn diese Entwurfsentscheidungen nicht gut von den Grenzen der Aggregates geleitet werden, überdenke das Modell. Deutet das Szenario in der Domäne auf eine wichtige neue Erkenntnis hin? Solche Änderungen verbessern oft die Aussagekraft und Flexibilität des Modells sowie die Lösung der Transaktions- und Verteilungsproblematik.

Repositories

Zugriff auf Aggregates durch Anfragen, die in der Ubiquitous Language ausgedrückt sind.

Die Verbreitung von navigierbaren Beziehungen, die nur dazu dienen, Dinge zu finden, bringen das Modell durcheinander. In ausgereiften Modellen drücken Abfragen oft Domänenkonzepte aus. Dennoch können Abfragen zu Problemen führen.

Die schiere technische Komplexität bei der Verwendung der meisten Infrastrukturen für den Datenbankzugriff beeinträchtigt schnell den Client-Code, was Entwickler dazu bringt, die Domänenebene zu reduzieren, was wiederum das Modell irrelevant macht.

Ein Query-Framework kann den größten Teil dieser technischen Komplexität kapseln, so dass Entwickler genau die Daten, die sie benötigen, automatisiert oder deklarativ aus der Datenbank beziehen können, aber das löst nur einen Teil des Problems.

Uneingeschränkte Abfragen können nur bestimmte Felder aus Objekten auslesen und so die Kapselung durchbrechen oder bestimmte Objekte aus dem inneren Zustand eines Aggregates instanziieren und dabei die Wurzel eines Aggregates umgehen und es so diesen Objekten unmöglich machen, die Regeln des Domänenmodells durchzusetzen. Die Domänenlogik verlagert sich in Abfragen und in Code der Anwendungsschicht, und die Entities und Value Objects werden zu reinen Datencontainern.

Daher:

Erstelle für jeden Aggregate-Typ, auf den global zugegriffen werden muss, einen Dienst, der die Illusion einer Sammlung aller Objekte des Typs der Wurzel dieses Aggregates vermitteln kann. Setze den Zugriff über eine bekannte globale Schnittstelle um. Stelle Methoden zum Hinzufügen und Entfernen von Objekten bereit, welche das tatsächliche Einfügen oder Entfernen von Daten in die Datenbank kapseln. Biete Methoden zur Auswahl von Ob-

jekten nach Kriterien an, die für Domänenexperten von Bedeutung sind. Liefere vollständig instanziierte Objekte oder Sammlungen von Objekten zurück, deren Attributwerte den Kriterien entsprechen, wodurch die eigentliche Speicher- und Abfragetechnologie gekapselt wird, oder gib Proxies zurück, welche die Illusion von vollständig instanziierten Aggregates vermitteln, aber die Daten verzögert nachladen. Stelle Repositories nur für Aggregate-Typen bereit, die tatsächlich direkten Zugriff benötigen. Halte die Anwendungslogik auf das Modell fokussiert und delegiere die gesamte Speicherung der Objekte und den Zugriff auf die Objekte an die Repositories.

Factories

Dt.: Fabriken

Wenn die Erstellung eines ganzen, intern konsistenten Aggregates oder eines großen Value Objects kompliziert wird oder zu viel von der internen Struktur preisgibt, bieten Factories eine Kapselung.

Das Erzeugen eines Objekts kann ein wichtiger Vorgang an sich sein, aber komplexe Operationen für das Zusammenbauen eines Objekts passen nicht in die Verantwortung der erstellten Objekte selbst. Werden diese Verantwortlichkeiten vermischt, kann dies zu schwer verständlichen Entwürfen führen. Muss der Client das Objekt direkt erzeugen, wird die Kapselung des erzeugten Value Objects oder Aggregates gebrochen und der Client zu stark an die Implementierung des erstellten Objekts gekoppelt.

Daher:

Verlagere die Verantwortung für das Erstellen von Instanzen komplexer Objekte und Aggregates auf ein separates Objekt, das selbst keine Verantwortung im Domänenmodell hat, aber dennoch Teil des Domänenentwurfs ist. Stelle eine Schnittstelle zur Verfügung, die den komplexen Zusammenbau kapselt und die es erlaubt, dass der Client keine Referenzen auf die konkreten Klassen der zu instanziierenden Objekte haben muss. Erstelle ein ganzes Aggregate als ein Stück und erzwinge seine Invarianten. Erstelle ein komplexes Value Object am Stück, womöglich durch Verwendung eines Builders für den Zusammenbau der einzelnen Elemente.

III. Flexibles Design

Ein Projekt, welches im Laufe der Entwicklung Fahrt aufnehmen kann - anstatt durch sein eigenes Erbe belastet zu werden - erfordert ein Design, mit dem man gerne arbeitet und das zum Wandel einlädt. Ein *flexibles Design*.

Flexibles Design ist die Ergänzung zur tiefgehenden Modellierung.

Entwickler spielen zwei Rollen, von denen jede durch das Design unterstützt werden muss. Die gleiche Person könnte durchaus beide

Rollen spielen - sogar innerhalb von Minuten hin und her wechseln - aber die Beziehung zum Code ist dennoch unterschiedlich. Eine Rolle ist der Entwickler eines Clients, der die Domänenobjekte in den Anwendungscode oder einen anderen Domänenschichtcode einbindet und dabei die Möglichkeiten des Designs nutzt. Ein flexibles Design offenbart ein tiefes, zugrunde liegendes Modell, das sein Potenzial deutlich macht. Die Client-Entwicklerin kann flexibel einen minimalen Satz lose gekoppelter Konzepte verwenden, um eine Reihe von Szenarien in der Domäne auszudrücken. Elemente des Designs passen auf natürliche Weise zusammen, mit einem Ergebnis, das vorhersehbar, klar charakterisiert und robust ist.

Ebenso wichtig ist, dass das Design den Entwickler, der daran arbeitet, dabei unterstützen muss, es zu ändern. Um offen für Veränderungen zu sein, muss ein Design leicht verständlich sein und das gleiche zugrunde liegende Modell offenbaren, auf das der Client-Entwickler zurückgreift. Es muss den Konturen eines tiefen Modells der Domäne folgen, so dass die meisten Änderungen das Design an flexiblen Punkten biegen. Die Auswirkungen des Codes müssen transparent sein, so dass die Folgen einer Änderung leicht voraussehbar sind.

- Verhalten offensichtlich machen
- Reduktion der Änderungskosten
- Erstellung von Software, mit der Entwickler*innen gerne arbeiten

Intention-Revealing Interfaces

Dt.: Ausdrucksstarke Schnittstellen

Wenn ein Entwickler die Implementierung einer Komponente berücksichtigen muss, um sie zu nutzen, geht der Wert der Kapselung verloren. Wenn jemand anderes als der ursprüngliche Entwickler den Zweck eines Objekts oder einer Operation aufgrund seiner Implementierung ableiten muss, kann dieser neue Entwickler einen Zweck ableiten, den die Operation oder Klasse nur durch Zufall erfüllt. Wenn das nicht die Absicht war, kann der Code im Moment funktionieren, aber die konzeptionelle Grundlage des Designs wird beschädigt sein, und die beiden Entwickler werden gegeneinander arbeiten.

Daher:

Benenne Klassen und Operationen, um ihre Wirkung und ihren Zweck zu beschreiben, ohne die Mittel zu nennen, mit denen sie das tun, was sie versprechen. Dies befreit den Client-Entwickler von der Notwendigkeit, das Innenleben zu verstehen. Diese Namen sollten der Ubiquitous Language entsprechen, damit Teammitglieder schnell ihre Bedeutung ableiten können. Schreibe einen Test für ein Verhalten, bevor du es implementierst, um dein Denken in den Client-Entwicklermodus zu zwingen.

Side-Effect-Free Functions

Dt.: Seiteneffektfreie Funktionen

Interaktionen mehrerer Regeln oder Kompositionen von Berechnungen werden extrem schwer vorherzusagen. Der Entwickler, der eine Operation aufruft, muss ihre Implementierung und die Implementierung aller Delegationen verstehen, um das Ergebnis voraussehen zu können. Der Nutzen jeder Abstraktion von Schnittstellen ist begrenzt, wenn Entwickler gezwungen sind, diesen Schleier zu lüften. Ohne sicher vorhersehbare Abstraktionen müssen Entwickler die kombinatorische Explosion begrenzt halten, was eine obere Grenzen für die Menge an Verhalten darstellt, die umgesetzt werden kann.

Daher:

Platziere so viel von der Programmlogik wie möglich in Funktionen, also Operationen, die Ergebnisse ohne beobachtbare Nebenwirkungen liefern. Separiere Kommandos (Methoden, die zu Änderungen des beobachtbaren Zustands führen) strikt in sehr einfache Operationen, die keine Domäneninformationen zurückgeben. Kontrolliere Seiteneffekte zusätzlich, indem komplexe Logik in Value Objects verschoben wird, wenn sich in der Fachlichkeit ein entsprechendes Konzept zeigt.

Alle Operationen eines Value Objects sollten seiteneffektfreie Funktionen sein.

Assertions

Dt.: Zusicherungen

Wenn die Nebenwirkungen von Operationen nur implizit durch ihre Umsetzung definiert sind, werden Entwürfe mit viel Delegation zu einem Gewirr aus Ursache und Wirkung. Der einzige Weg, ein Programm zu verstehen, besteht darin, die Ausführung durch verzweigte Pfade hindurch nachzuverfolgen. Der Wert der Kapselung geht verloren. Die Notwendigkeit, die konkrete Ausführung nachverfolgen zu müssen, zerstört die Abstraktion.

Daher:

Gib die Nachbedingungen von Operationen und die Invarianten von Klassen und Aggregaten an. Wenn Zusicherungen nicht direkt in der Programmiersprache kodiert werden können, schreibe automatisierte Unit Tests für sie. Schreibe sie in Dokumentationen oder in Diagramme, falls dies zum Stil des Entwicklungsprozesses des Projekts passt.

Suche nach Modellen mit kohärenten Konzepten, die einen Entwickler unterstützen, die beabsichtigten Zusicherungen zu erkennen, dadurch die Lernkurve zu beschleunigen und das Risiko von widersprüchlichem Code zu reduzieren.

Zusicherungen definieren Schnittstellenverträge und Modifikatoren von Entities.

Zusicherungen definieren Invarianten auf Aggregates.

Standalone Classes

Dt.: Eigenständige Klassen

Selbst innerhalb eines Moduls nimmt die Schwierigkeit der Interpretation eines Entwurfs mit zunehmenden Abhängigkeiten stark zu. Dies führt zu geistiger Überlastung und begrenzt die Komplexität des Entwurfs, mit der ein Entwickler umgehen kann. Implizite Konzepte tragen zu dieser Belastung noch mehr bei als explizite Referenzen.

Eine geringe Kopplung ist für den Entwurf von Objekten von grundlegender Bedeutung. Wenn du kannst, gehe den ganzen Weg: eliminiere alle anderen Konzepte aus dem Bild. Dann ist eine Klasse völlig in sich geschlossen und kann allein studiert und verstanden werden. Jede dieser in sich geschlossenen Klassen erleichtert das Verständnis eines Moduls erheblich.

Closure of Operations

Dt.: Geschlossene Funktionen

Die meisten interessanten Objekte tun am Ende Dinge, die nicht nur von primitiven Datentypen charakterisiert werden können.

Daher:

Wenn es passt, definiere eine Funktion, deren Rückgabetyp derselbe ist wie der Typ ihres/ihrer Argumente(s). Wenn der Implementierer einen Zustand hat, der bei der Berechnung verwendet wird, dann ist der Implementierer faktisch ein Argument der Funktion, so dass das/die Argument(e) und der Rückgabewert vom gleichen Typ wie der Implementierer sein sollten. Eine solche Funktion ist unter der Menge der Instanzen dieses Typs geschlossen. Eine geschlossene Funktion bietet eine höherwertige Schnittstelle, ohne Abhängigkeit auf andere Konzepte.

Dieses Muster wird meistens auf die Operationen eines Value Objects angewendet. Da der Lebenszyklus einer Entity eine Bedeutung in der Domäne hat, kann man nicht einfach ein neue Entity erzeugen, um eine Anfrage zu beantworten. Es gibt Vorgänge, die unter einem Entitätstyp abgeschlossen sind. Du kannst ein Mitarbeiterobjekt nach seinem Vorgesetzten fragen und ein anderes Mitarbeiterobjekt zurückerhalten. Aber im Allgemeinen sind Entities keine Konzepte, die häufig das Ergebnis einer Berechnung sind. In den meisten Fällen ist dies also eine Eigenschaft, die in Value Objects zu suchen ist.

Manchmal erreicht man dieses Muster nur halb: das Argument passt zum Implementierer, aber der Rückgabetyp ist unterschiedlich, oder der Rückgabetyp passt zum Empfänger, aber das Argument ist unterschiedlich. Diese Funktionen sind nicht geschlossen, aber sie bieten einen Teil des Vorteils der Geschlossenheit, indem sie einfacher verständlich sind.

Declarative Design

Dt.: Deklarativer Entwurf

Es kann keine wirklichen Garantien in prozeduraler Software geben. Um nur eine Möglichkeit für das Umgehen von Assertions zu nennen: Code könnte zusätzliche Seiteneffekte haben, die nicht ausdrücklich ausgeschlossen wurden. Egal wie modellgetrieben unser Entwurf ist, wir schreiben immer noch Prozeduren, um die Wirkung von konzeptionellen Interaktionen zu erzeugen. Und wir verbringen einen Großteil unserer Zeit damit, Boilerplate Code zu schreiben, der keine wirkliche Bedeutung oder kein Verhalten hat. Intention Revealing Interfaces und die anderen Muster in diesem Kapitel helfen, aber sie können konventionellen objektorientierten Programmen niemals formale Strenge verleihen.

Dies sind einige der Motivationen für deklarativen Entwurf. Dieser Begriff bedeutet verschiedenen Menschen unterschiedliche Dinge, aber normalerweise beschreibt er einen Weg, ein Programm oder einen Teil eines Programms als eine Art ausführbare Spezifikation zu schreiben. Eine genaue Beschreibung der Eigenschaften steuert effektiv die Software. Mit verschiedenen Herangehensweisen könnte dies durch Reflection oder zur Kompilierzeit durch Codegenerierung (automatische Erzeugung von konventionellem Code auf der Grundlage der Deklaration) geschehen. Dieser Ansatz ermöglicht es einem anderen Entwickler, die Deklaration zum Nennwert zu nehmen. Es ist eine absolute Garantie.

Viele deklarative Ansätze können korrumpiert werden, wenn die Entwickler sie absichtlich oder unabsichtlich umgehen. Dies ist wahrscheinlich, wenn das System schwer zu bedienen oder zu restriktiv ist. Jeder muss die Regeln des Frameworks befolgen, um die Vorteile eines deklarativen Programms zu nutzen.

Ein deklarativer Entwurfstil

Sobald dein Entwurf über Intention Revealing Interfaces, Side Effect Free Functions und Assertions verfügt, begibst du Dich in deklaratives Gebiet. Viele der Vorteile des deklarativen Entwurfs werden erzielt, sobald du kombinierbare Elemente hast, die ihre Bedeutung kommunizieren, und die charakteristische oder offensichtliche Effekte oder gar keine beobachtbaren Effekte haben.

Ein flexibles Design kann es dem Client-Code ermöglichen, einen deklarativen Entwurfstil zu verwenden. Zur Veranschaulichung werden im nächsten Abschnitt einige der Muster in diesem Kapitel zusammengefasst, um die Spezifikation flexibler und deklarativer zu gestalten.

Drawing on Established Formalisms

Dt.: Arbeiten auf Basis etablierter Formalismen

Ein enges konzeptionelles Framework von Grund auf zu schaffen, ist etwas, was man nicht jeden Tag tun kann. Manchmal entdeckt und verfeinert man eines davon im Laufe des Lebens eines Projekts. Aber oft kannst du konzeptionelle Systeme verwenden und anpassen, die in deiner oder einer anderen fachlichen Domänen seit langem etabliert sind. Einige davon wurden möglicherweise sogar über Jahrhunderte hinweg verfeinert und kondensiert. Viele Geschäftsanwendungen betreffen z.B. das Rechnungswesen. Das Rechnungswesen definiert ein gut entwickeltes Set von Entities und Regeln, die eine einfache Anpassung an ein tiefgehendes Modell und ein flexibles Design ermöglichen.

Es gibt viele solcher formalisierten konzeptionellen Rahmen, aber mein persönlicher Favorit ist die Mathematik. Es ist überraschend, wie nützlich es sein kann, einen bestimmten Trick aus der Arithmetik zu verwenden. Viele Domänen beinhalten Mathematik irgendwo. Such danach, grab sie aus. Spezialisierte Mathematik ist sauber, kombinierbar mit klaren Regeln, und Leute finden sie leicht zu verstehen.

Ein praktisches Beispiel, “Shares Math”, wurde in Kapitel 8 des Buches “Domain-Driven Design” diskutiert.

Conceptual Contours

Dt.: Konzeptionelle Konturen

Manchmal schneiden Leute die Funktionalität in kleine Teile, um eine flexible Kombination zu ermöglichen. Manchmal favorisieren sie einen eher grobgranularen Schnitt, um die Komplexität zu kapseln. Manchmal streben sie nach einer konsistenten Granularität, wodurch alle Klassen und Operationen in ähnlicher Größenordnung umgesetzt werden. Das sind übertriebene Vereinfachungen, die als allgemeine Regeln nicht gut funktionieren. Aber sie sind durch grundlegende Probleme motiviert.

Wenn Elemente eines Modells oder Entwurfs in ein monolithisches Konstrukt eingebettet sind, wird deren Funktionalität dupliziert. Die externe Schnittstelle sagt nicht alles, was einem Verwender wichtig sein könnte. Ihre Bedeutung ist schwer zu verstehen, da verschiedene Konzepte miteinander vermischt werden.

Umgekehrt kann die Zerlegung von Klassen und Methoden den Client sinnlos verkomplizieren, indem sie Clients dazu zwingt, zu verstehen, wie kleine Teile zusammenpassen. Schlimmer noch, ein Konzept kann völlig verloren gehen. Die Hälfte eines Uranatoms ist nicht Uran. Natürlich kommt es nicht nur auf die Größe der Maserung an, sondern auch, wie die Maserung verläuft.

Daher:

Zerlege die Entwurfselement (Funktionen, Schnittstellen, Klassen, Aggregate) in zusammenhängende Einheiten. Berücksichtige deine Intuition zu den wichtigen Teilbereiche in der Domäne. Beobachte die Achsen von Veränderung und Stabilität durch sukzessives Refactoring und suche nach den zugrunde liegenden konzeptionellen Konturen, die dieses Schnittmuster erklären. Richte das Modell auf die konsistenten Aspekte der Domäne aus, die es überhaupt zu einem lebensfähigen Wissensgebiet machen.

Ein flexibles Design, das auf einem tiefgehenden Modell basiert, ergibt eine einfache Reihe von Schnittstellen, die sich logisch kombinieren, um sinnvolle Aussagen in der Ubiquitous Language zu treffen, ohne die Ablenkung und den Wartungsaufwand irrelevanter Optionen.

IV. Context Mapping für Strategic Design

Bounded Context

Eine Beschreibung einer Grenze (typischerweise ein Subsystem oder die Arbeit eines bestimmten Teams), innerhalb derer ein bestimmtes Modell definiert und anwendbar ist.

Upstream-Downstream

Eine Beziehung zwischen zwei Gruppen, in der die Handlungen der "Upstream"-Gruppe den Projekterfolg der "Downstream"-Gruppe beeinflussen, während die Handlungen der Downstream-Gruppe die Upstream-Projekte nicht wesentlich beeinflussen. (Z.B. wenn zwei Städte am gleichen Fluss liegen, wirkt sich die Verschmutzung der upstream (Dt.: flussaufwärts) gelegenen Stadt in erster Linie auf die downstream gelegene Stadt (Dt.: flussabwärts) aus).

Das Upstream-Team kann unabhängig vom Schicksal des Downstream-Teams erfolgreich sein.

Mutually Dependent

Mutually Dependent (Dt.: wechselseitig abhängig) beschreibt eine Situation, in der zwei Softwareentwicklungsprojekte in getrennten Kontexten beide geliefert werden müssen, damit jedes von beiden als erfolgreich angesehen werden kann. (Wenn zwei Systeme jeweils auf Informationen oder Funktionen des anderen angewiesen sind - was wir im Allgemeinen vermeiden würden - sehen wir die Projekte,

die diese Systeme entwickeln, natürlich als wechselseitig miteinander verflochten. Es gibt aber auch wechselseitig abhängige Projekte, bei denen Systemabhängigkeiten nur in eine Richtung verlaufen. Wenn ein System ohne ein bestimmtes abhängiges System und ohne die Integration mit diesem abhängigen System wenig Wert hat - vielleicht weil dies der einzige Ort ist, an dem es verwendet wird - dann führt es zu einem Misserfolg beider Projekte, wenn das abhängige System nicht ausgeliefert wird.)

Free

Ein Softwareentwicklungskontext ist free (Dt.: frei), wenn die Ausrichtung, der Erfolg oder das Scheitern der Entwicklungsarbeit in anderen Kontexten wenig Einfluss auf die Auslieferung hat.

Context Map

Dt.: Kontextlandkarte

Um eine Strategie entwickeln zu können, benötigen wir eine realistische, groß angelegte Sicht auf die Modellentwicklung, die sich über unser gesamtes Projekt und andere, in die wir integrieren, erstreckt.

Ein einzelner Bounded Context hinterlässt einige Probleme, wenn keine globale Sichtweise vorhanden ist. Der Kontext anderer Modelle kann immer noch vage und im Fluss sein.

Leute in anderen Teams sind sich der Grenzen der Bounded Contexts nicht sehr bewusst und nehmen unwissentlich Änderungen vor, welche die Übergänge verwischen oder die Abhängigkeiten verkomplizieren. Wenn Verbindungen zwischen verschiedenen Kontexten hergestellt werden müssen, neigen diese dazu, ineinander zu verlaufen.

Selbst wenn die Grenzen klar sind, stellen Beziehungen zu anderen Kontexten Einschränkungen für die Art des Modells oder das mögliche Änderungstempo dar. Diese Einschränkungen manifestieren sich in erster Linie durch nicht-technische Kanäle, die manchmal schwer mit den von ihnen betroffenen Entwurfsentscheidungen in Verbindung zu bringen sind.

Daher:

Identifiziere jedes Modell, das beim Projekt im Spiel ist und definiere seinen Bounded Context. Dazu gehören auch die impliziten Modelle von nicht-objekt-orientierten Subsystemen. Benenne jeden Bounded Context und mache diese Namen zu einem Teil der Ubiquitous Language.

Beschreibe die Berührungspunkte zwischen den Modellen, skizziere die explizite Übersetzung für jede Art der Kommunikation, hebe alle Austausch-, Isolations- und Einflussmechanismen hervor.

Kartografiere das vorhandene Gelände. Nimm Transformationen erst später vor.

Diese Karte kann die Grundlage für eine realistische Design-Strategie sein.

Die Charakterisierung von Beziehungen wird auf den folgenden Seiten mit einer Reihe von gängigen Mustern für Beziehungen zwischen Bounded Contexts konkretisiert.

Partnership

Dt.: Partnerschaft

Wenn Teams in zwei Kontexten gemeinsam erfolgreich sein oder scheitern werden, entsteht oft eine kooperative Beziehung.

Eine schlechte Koordination voneinander abhängiger Subsysteme in getrennten Kontexten führt bei beiden Projekten dazu, dass sie nicht erfolgreich liefern können. Eine wichtige Funktionalität, die in einem System fehlt, kann dazu führen, dass das andere System nicht liefern kann. Schnittstellen, die nicht den Erwartungen der Entwickler des anderen Subsystems entsprechen, können zum Fehlschlagen der Integration führen. Eine gemeinsam vereinbarte Schnittstelle kann sich als so umständlich in der Verwendung erweisen, dass sie die Entwicklung des Client-Systems verlangsamt, oder als so schwer zu implementieren, dass sie die Entwicklung des Server-Subsystems verlangsamt. Ein Misserfolg bringt beide Projekte zum Scheitern.

Daher:

Wenn ein Fehlschlag bei der Entwicklung in einem der beiden Kontexte zu einem Misserfolg für beide führen würde, solltest du eine Partnerschaft zwischen den für die beiden Kontexte zuständigen Teams etablieren. Führe einen Prozess zur koordinierten Planung der Entwicklung und zum gemeinsamen Managements der Integration ein.

Die Teams müssen bei der Entwicklung ihrer Schnittstellen zusammenarbeiten, um die Bedürfnisse bei der Entwicklung beider Systeme zu berücksichtigen. Voneinander abhängige Funktionalitäten sollten so eingeplant werden, dass sie auf das gleiche Release hin abgeschlossen werden.

Es ist in den meisten Fällen nicht notwendig, dass Entwickler das Modell des anderen Subsystems im Detail verstehen, aber sie müssen ihre Projektplanung koordinieren. Wenn die Entwicklung in einem Kontext auf Hindernisse stößt, ist eine gemeinsame Auseinandersetzung mit

dem Thema erforderlich, um eine schnelle Entwurfslösung zu finden, die beide Kontexte nicht übermäßig negativ beeinträchtigt.

Außerdem ist ein klarer Prozess zur Steuerung der Integration erforderlich. So kann beispielsweise eine spezielle Testsuite definiert werden, die zeigt, dass die Schnittstelle den Erwartungen des Client-Systems entspricht und die im Rahmen der Continuous Integration auf dem Server-System ausgeführt werden kann.

Partnership ist ein neuer Begriff, der seit dem Buch von 2004 entstanden ist.

Shared Kernel

Dt.: Geteilter Kern

Die gemeinsame Nutzung eines Teils des Modells und des zugehörigen Codes erzeugt eine sehr enge wechselseitige Abhängigkeit, welche die Entwurfsarbeit in Schwung bringen oder sie untergraben kann.

Wenn die funktionale Integration begrenzt ist, kann der Aufwand für die Continuous Integration eines großen Kontextes für zu hoch gehalten werden. Das kann insbesondere der Fall sein, wenn das Team nicht über die Fähigkeit oder die politische Organisation verfügt, eine Continuous Integration aufrechtzuerhalten, oder wenn ein einzelnes Team einfach zu groß und zu schwerfällig wäre. So können stattdessen getrennte Bounded Contexts definiert und mehrere Teams gebildet werden.

Einmal getrennt, können unkoordinierte Teams, die an eng verwandten Anwendungen arbeiten, eine Weile schnell vorankommen, aber was sie produzieren, passt vielleicht nicht zusammen. Selbst partnerschaftliche Teams können am Ende viel für Übersetzungsschichten und Umbauten ausgeben, während sie Aufwände verdoppeln und die Vorteile einer gemeinsamen Ubiquitous Language verlieren.

Daher:

Markiere mit einer expliziten Grenze eine Teilmenge des Domänenmodells, welche die Teams gemeinsam nutzen wollen. Halte diesen Kern klein.

Füge innerhalb dieser Grenze neben dieser Teilmenge des Modells die Teilmenge des Codes oder des Datenbankdesigns hinzu, die mit diesem Teil des Modells verbunden ist. Diese explizit geteilten Artefakte haben einen besonderen Status und sollten nicht ohne Rücksprache mit dem anderen Team geändert werden.

Definiere einen Prozess für Continuous Integration, der das Kernmodell kompakt hält und stimme die Ubiquitous Language der Teams aufeinander ab. Integriere ein funktionales System häufig, wenn auch etwas seltener als das Tempo der Continuous Integration innerhalb der Teams.

Customer/Supplier Development

Dt.: Kunde/Lieferant

Wenn sich zwei Teams in einer Upstream-Downstream-Beziehung befinden, in der das Upstream-Team unabhängig vom Schicksal des Downstream-Teams erfolgreich sein kann, werden die Bedürfnisse des Downstream-Teams auf verschiedene Arten mit vielen unterschiedlichen Konsequenzen berücksichtigt.

Ein Downstream-Team kann hilflos sein, vollständig den Upstream-Prioritäten ausgeliefert. Währenddessen kann das Upstream-Team blockiert sein, weil es sich Sorgen macht, dass Downstream-Systeme gebrochen werden. Die Probleme des Downstream-Teams werden durch umständliche Verfahren zum Anfordern von Änderungen über komplexe Genehmigungsprozesse nicht geringer. Und die ungehinderte Entwicklung des Upstream-Teams wird gestoppt, wenn das Downstream-Team ein Vetorecht bei Änderungen hat.

Daher:

Etabliere eine klare Customer-Supplier-Beziehung zwischen den beiden Teams, d.h. Downstream-Prioritäten fließen in die Upstream-Planung ein. Verhandle und budgetiere Aufgaben für die Downstream-Anforderungen, so dass jeder die Verpflichtungen und den Zeitplan versteht.

Agile Teams können das Downstream-Team bei der Planung die Rolle des Kunden gegenüber dem Upstream-Team spielen lassen. Gemeinsam entwickelte automatisierte Akzeptanztests können die erwartete Schnittstelle vom Upstream validieren. Das Hinzufügen dieser Tests zur Testsuite des Upstream-Teams, die im Rahmen der Continuous Integration ausgeführt wird, befreit das Upstream-Team bei Änderungen von der Angst, unbeabsichtigte Auswirkungen auf Downstream zu haben.

Conformist

Dt.: Konformist

Wenn zwei Entwicklungsteams eine Upstream/Downstream-Beziehung haben, in der Upstream keine Motivation hat, den Bedürfnissen des Downstream-Teams zu entsprechen, ist das Downstream-Team hilflos. Altruismus mag Upstream-Entwickler motivieren, Versprechungen zu machen, aber sie werden wahrscheinlich nicht erfüllt werden. Der Glaube an diese guten Absichten führt dazu, dass das Downstream-Team auf der Grundlage von Funktionalitäten plant, die nie verfügbar sein werden. Das Downstream-Projekt wird sich verzögern, bis das Team schließlich lernt, mit dem zu leben, was ihm gegeben wird. Eine auf die Bedürfnisse des Downstream-Teams zugeschnittene Schnittstelle ist nicht in Sicht.

Daher:

Eliminiere die Komplexität der Übersetzung zwischen Bounded Contexts, indem du dich sklavisch an das Modell des Upstream-Teams hältst. Obwohl dies den Stil der Downstream-Designer einengt und wahrscheinlich nicht das ideale Modell für die Anwendung ergibt, vereinfacht Konformität die Integration enorm. Außerdem wirst du dir eine Ubiquitous Language mit dem Upstream-Team teilen. Upstream hat das Sagen, daher ist es gut, die Kommunikation für sie möglichst einfach zu gestalten. Altruismus kann ausreichen, um sie dazu zu bringen, Informationen mit dir zu teilen.

Anticorruption Layer

Dt.: Antikorruptionsschicht

Übersetzungsschichten können einfach, ja sogar elegant sein, wenn es darum geht, gut gestaltete Bounded Context kooperativer Teams zu verbinden. Aber wenn die Kontrolle oder die Kommunikation nicht ausreicht, um einen Shared Kernel, eine Partnership oder Customer/-Supplier-Beziehung zu erreichen, wird die Übersetzung komplexer. Die Übersetzungsschicht nimmt eher einen defensiven Charakter an.

Eine große Schnittstelle mit einem Upstream-System kann am Ende die Absicht des Downstream-Modells insgesamt überfordern, so dass das Modell ad hoc an das Modell des anderen Systems angeglichen wird. Die Modelle von Altsystemen sind in der Regel schwach (wenn nicht sogar Big Balls of Mud), und selbst wenn das Modell ausnahmsweise gut gestaltet ist, mag es nicht den Anforderungen des aktuellen Projekts entsprechen, so dass es unpraktisch wäre, sich an das Upstream-Modell anzupassen. Dennoch kann die Integration für das Downstream-Projekt sehr wertvoll oder sogar erforderlich sein.

Daher:

Erstelle als Downstream-Client eine Isolationsschicht, um deinem System die Funktionalität des Upstream-Systems in Form deines eigenen Domänenmodells zur Verfügung zu stellen. Diese Schicht spricht mit dem anderen System über seine bestehende Schnittstelle und erfordert wenig oder gar keine Änderungen am anderen System. Intern übersetzt die Schicht je nach Bedarf in eine oder beide Richtungen zwischen den beiden Modellen.

Open-host Service

Dt.: Offen angebotener Dienst

Typischerweise definierst du für jeden Bounded Context und jede Komponente eine Übersetzungsschicht, mit der du integrieren musst und die außerhalb des Kontextes liegt. Wenn jede Integration anders ist, vermeidet dieser Ansatz mit einer Übersetzungsschicht für jedes externe System die Korruption der Modelle mit minimalen Kosten. Wenn dein Subsystem jedoch stark nachgefragt ist, benötigst du möglicherweise einen flexibleren Ansatz.

Wenn ein Subsystem mit vielen anderen integriert werden muss, kann die Anpassung der Übersetzungsschicht für jede einzelne Integration das Team dazu bringen sich zu verzetteln. Es gibt immer mehr zu warten und immer mehr zu beachten, wenn Änderungen vorgenommen werden.

Daher:

Definiere ein Protokoll, das den Zugriff auf dein Subsystem als eine Menge von Services ermöglicht. Öffne das Protokoll, so dass alle, die sich mit dir integrieren müssen, das Protokoll nutzen können. Verbessere und erweitere das Protokoll, um neue Integrationsanforderungen zu erfüllen, es sei denn, ein einzelnes Team hat eigenartige Anforderungen. Verwende dann eine spezifische Übersetzungsschicht, um das Protokoll für diesen speziellen Fall zu erweitern, so dass das gemeinsame Protokoll einfach und kohärent bleibt.

Damit befindet sich der Anbieter des Service in der Upstream-Position. Jeder Client ist Downstream, und typischerweise sind einige von ihnen Conformist und andere bauen Anticorruption Layer. Ein Kontext mit einem Open Host Service kann zudem jede Art von Beziehung zu anderen Kontexten haben, die nicht solche Clients sind.

Published Language

Dt.: veröffentlichte Sprache

Die Übersetzung zwischen den Modellen zweier Bounded Contexts erfordert eine gemeinsame Sprache.

Die direkte Übersetzung in und aus den bestehenden Domänenmodellen ist möglicherweise keine gute Lösung. Diese Modelle können übermäßig komplex oder schlecht aufgeteilt sein. Sie sind wahrscheinlich undokumentiert. Wenn eines dieser Modelle als Datenaustauschsprache verwendet wird, wird es im Wesentlichen eingefroren und kann nicht auf neue Entwicklungsbedürfnisse reagieren.

Deshalb:

Verwende eine gut dokumentierte gemeinsame Sprache, welche die notwendigen Domäneninformationen als gemeinsames Kommunikationsmedium ausdrückt und übersetze nach Bedarf in diese Sprache bzw. aus dieser Sprache.

Viele Branchen definieren Published Languages in Form von Datenaustauschstandards. Projektteams entwickeln aber auch ihre eigenen für den Einsatz in ihrem Unternehmen.

Published Language wird oft mit Open Host Service kombiniert.

Separate Ways

Dt.: Getrennte Wege

Wir müssen rücksichtslos sein, wenn es um die Definition von Anforderungen geht. Wenn zwei Mengen von Funktionalitäten keine signifikante Beziehung zueinander haben, können sie vollständig voneinander getrennt werden.

Integration ist immer teuer, und manchmal ist ihr Nutzen gering.

Daher:

Definiere, dass ein Bounded Context gar keine Verbindung zu den anderen hat, so dass Entwickler einfache, spezialisierte Lösungen in diesem kleinen Bereich finden können.

Big Ball of Mud

Dt.: Große Matschkugel

Wenn wir bestehende Softwaresysteme untersuchen und versuchen zu verstehen, wie unterschiedliche Modelle innerhalb definierter Grenzen angewendet werden, finden wir Teile von Systemen, oft große, in denen Modelle gemischt und Grenzen inkonsistent sind.

Es ist leicht, beim Versuch festzufahren, die Kontextgrenzen von Modellen in Systemen zu beschreiben, in denen es einfach keine Grenzen gibt.

Gut definierte Kontextgrenzen entstehen erst durch intellektuelle Entscheidungen und soziale Kräfte (auch wenn die Personen, welche die Systeme schaffen, sich dieser Ursachen zu dem jeweiligen Zeitpunkt nicht immer bewusst waren). Wenn diese Faktoren fehlen oder verschwinden, vermischen sich mehrere konzeptionelle Systeme und machen Definitionen und Regeln mehrdeutig oder widersprüchlich. Die Systeme beginnen nach einer unvorhergesehenen Logik zu funktionieren, wenn Funktionalitäten hinzugefügt werden. Abhängigkeiten durchziehen die Software kreuz und quer. Ursache und Wirkung werden immer schwieriger nachvollziehbar. Schließlich erstarrt die Software zu einer großen Matschkugel.

Ein solcher Big Ball of Mud ist für einige Situationen tatsächlich recht praktisch (wie im Originalartikel von Foote und Yoder beschrieben), aber er verhindert fast vollständig die Subtilität und Präzision, die für nützliche Modelle erforderlich ist.

Deshalb:

Ziehe eine Grenze um das gesamte Durcheinander und bezeichne es als einen Big Ball of Mud. Versuche nicht, in diesem Kontext eine ausgeklügelte Modellierung anzuwenden. Sei auf der Hut vor der Tendenz, dass sich solche Systeme in andere Kontexte ausbreiten.

(siehe http://www.laputan.org/mud/mud.html[5], Brian Foote und Joseph Yoder)

Big Ball of Mud ist ein neuer Begriff, der seit dem Buch von 2004 entstanden ist.

[5] http://www.laputan.org/mud/mud.html

V. Destillation für Strategic Design

$$divD = \rho$$

$$divB = 0$$

$$curlE = -\frac{\partial B}{\partial t}$$

$$curlH = J + \frac{\partial D}{\partial t}$$

> James Clerk Maxwell, A Treatise on Electricity & Magnetism, 1873
>
> Diese vier Gleichungen drücken zusammen mit der Definition ihrer Terme und der notwendigen Mathematik das gesamte Wissen über Elektromagnetismus im 19. Jahrhundert aus.

Wie kann man sich auf das zentrale Problem konzentrieren und verhindern, dass man in einem Meer von Nebensächlichkeiten ertrinkt?

Destillation bezeichnet einen Prozess, um die Bestandteile einer Mischung zu trennen und so die Essenz in einer Form zu extrahieren, die sie wertvoller und nützlicher macht. Ein Modell ist eine Destillation von Wissen. Mit jedem Refactoring Toward Deeper Insight

abstrahieren wir entscheidende Aspekte des Domänenwissens und der Prioritäten in der Domäne. Um auf die strategische Perspektive zurückzukommen, beschäftigt sich dieses Kapitel mit Möglichkeiten, die größten Bestandteile des Modells zu unterscheiden und das Domänenmodell als Ganzes zu destillieren.

Core Domain

Dt.: Kerndomäne

In einem großen System gibt es so viele Komponenten, die alle kompliziert und absolut notwendig für den Erfolg sind, dass die Essenz des Domänenmodells, der eigentliche Unternehmenswert, unklar sein und dann vernachlässigt werden kann.

Die harte Realität ist, dass nicht alle Teile des Entwurfs gleich ausgefeilt sein werden. Es müssen Prioritäten gesetzt werden. Um das Domänenmodell zu einem Wert zu machen, muss der kritische Kern dieses Modells schlank sein und vollständig genutzt werden, um Anwendungsfunktionalitäten zu erstellen. Aber die wenigen hochqualifizierte Entwickler neigen dazu, sich auf die technische Infrastruktur oder sauber definierbare Domänenprobleme zu konzentrieren, die ohne spezielle Domänenkenntnisse verstanden werden können.

Daher:

Dampfe das Modell auf das Wesentliche ein. Definiere eine Kerndomäne und biete eine Möglichkeit, sie leicht von der großen Anzahl unterstützender Modelle und unterstützendem Code zu unterscheiden. Bringe die wertvollsten und spezialisiertesten Konzepte auf den Punkt. Mache den Kern klein.

Nutze die besten Talente für die Core Domain und rekrutiere entsprechend. Investiere den Aufwand, um ein tiefgreifendes Modell zu finden und ein flexibles Design zu entwickeln, das ausreicht, um die Vision des Systems zu erfüllen.

Rechtfertige die Investition in andere Teile, indem du betrachtest, wie sie den destillierten Kern unterstützen.

Generic Subdomains

Dt.: Allgemeine Teildomänen

Einige Teile des Modells erhöhen die Komplexität, ohne Spezialwissen zu erfassen oder zu vermitteln. Alles, was nicht relevant ist, macht die Core Domain schwieriger zu erkennen und zu verstehen. Das Modell wird zugestellt mit allgemeinen Prinzipien, die jeder kennt, oder mit Details, die zu Spezialitäten gehören, die nicht das Hauptaugenmerk sind, sondern eine unterstützende Rolle spielen. Doch wie allgemein auch immer sie sind, diese anderen Elemente sind für das Funktionieren des Systems und die vollständige Ausprägung des Modells von wesentlicher Bedeutung.

Daher:

Identifiziere zusammenhängende Subdomains, die nicht die Motivation für dein Projekt sind. Extrahiere generische Modelle dieser Subdomains und lege sie in separate Module. Hinterlasse in diesen Modellen keine Spuren von Spezialitäten deiner Domäne.

Sobald sie getrennt wurden, gib ihrer weiteren Entwicklung eine niedrigere Priorität als der Core Domain und vermeide es, Kernentwickler den Aufgaben zuzuordnen (da sie aus ihnen wenig Domänenwissen gewinnen). Erwäge auch Standardlösungen oder öffentlich verfügbare Modelle für diese Generic Subdomains.

Domain Vision Statement

Dt.: Aussage zur Domänenvision

Zu Beginn eines Projekts existiert das Modell in der Regel noch gar nicht, aber seine Entwicklung muss schon fokussiert werden. In späteren Entwicklungsstadien muss der Wert des Systems erklärt werden, ohne dass man dazu das Modell eingehend studieren muss. Auch können sich kritische Aspekte des Domänenmodells über mehrere Bounded Contexts erstrecken, aber per Definition können diese unterschiedlichen Modelle nicht so strukturiert werden, dass sie ihren gemeinsamen Fokus zeigen.

Daher:

Verfasse eine kurze Beschreibung (etwa eine Seite) der Core Domain und des Wertes, den sie bringen wird, das "Leistungsversprechen". Ignoriere Aspekte, die dieses Domänenmodell nicht von anderen unterscheiden. Zeige, wie das Domänenmodell verschiedenen Interessen dient und sie ausgleicht. Halte diese Aussage kurz. Schreibe sie frühzeitig und überarbeite sie, wenn du neue Erkenntnisse gewinnst.

Highlighted Core

Dt.: Hervorgehobener Kern

Ein Domain Vision Statement beschreibt die Core Domain grob, aber überlässt die Definition der spezifischen Elemente des Kernmodells den Ungenauigkeiten einer individuellen Interpretation. Ohne ein außergewöhnlich hohes Maß an Kommunikation im Team wird das Domain Vision Statement alleine wenig Einfluss haben.

Auch wenn die Teammitglieder im Großen und Ganzen wissen, was zur Core Domain gehört, werden verschiedene Personen nicht genau die gleichen Elemente herausgreifen, und selbst die gleiche Person wird nicht von einem Tag auf den anderen konsistente Aussagen dazu machen. Der mentale Aufwand, das Modell ständig zu filtern, um die wesentlichen Teile zu identifizieren, benötigt die volle Konzentration, die besser für das Nachdenken über den Entwurf aufgewendet werden sollte, und erfordert detaillierte Kenntnisse über das Model. Die Core Domain muss leichter erkennbar werden.

Signifikante strukturelle Änderungen am Code sind der ideale Weg, um die Core Domain zu identifizieren, aber sie sind nicht immer kurzfristig praktikabel. Tatsächlich sind solche großen Code-Änderungen nur schwer durchführbar, ohne die Sichtweise, welche dem Team genau fehlt.

Daher (als eine Form des Highlighted Core):

Verfasse ein sehr kurzes Dokument (drei bis sieben kurze Seiten), das die Core Domain und die primären Interaktionen zwischen den Kernelementen beschreibt.

und/oder (als eine weitere Form des Highlighted Core):

Markiere die Elemente der Core Domain in der primären Ablage des Modells, ohne deren Rolle besonders zu erklären. Mache es Entwicklern leicht herauszufinden, was im Kern ist, und was nicht.

Wenn das Destillationsdokument die Grundzüge der Kerndomäne beschreibt, dann dient es als pragmatischer Indikator für die Bedeutung einer Modelländerung. Wenn eine Modell- oder Code-Änderung Auswirkungen auf das Destillationsdokument hat, muss mit anderen Teammitgliedern Rücksprache gehalten werden. Wenn die Änderung vorgenommen wird, erfordert das eine sofortige Benachrichtigung aller Teammitglieder und die Veröffentlichung einer neuen Version des Dokuments. Änderungen außerhalb des Kerns oder an Details, die nicht im Destillationsdokument enthalten sind, können ohne Rücksprache oder Benachrichtigung vorgenommen werden und werden von anderen Mitgliedern im Laufe ihrer Arbeit wahrgenommen. So haben Entwickler die volle Unabhängigkeit, welche die meisten agilen Prozesse vorschlagen.

Obwohl das Domain Vision Statement und der Highlighted Core informieren und leiten, ändern sie nicht wirklich das Modell oder den Code selbst. Generic Subdomains physisch aufzuteilen entfernt einige störende Elemente. Als nächstes werden wir uns andere Möglichkeiten ansehen, das Modell und den Entwurf selbst strukturell zu ändern, um die Core Domain sichtbarer und überschaubarer zu machen.

Cohesive Mechanisms

Dt.: Zusammenhängender Mechanismus

Logik erreicht manchmal eine Komplexität, die den Entwurf aufzublähen beginnt. Das konzeptuelle "Was" wird vom mechanischen "Wie" überlagert. Eine große Anzahl von Methoden, die Algorithmen zur Lösung des Problems bereitstellen, verdecken die Methoden, die das Problem ausdrücken.

Daher:

Unterteile einen konzeptionell zusammenhängenden Mechanismus in ein separates, leichtgewichtiges Framework. Achte insbesondere auf Formalismen oder gut dokumentierte Kategorien von Algorithmen. Stelle die Funktionalitäten des Frameworks durch ein Intention-Revealing Interface zur Verfügung. Nun können sich die anderen Elemente der Domäne darauf konzentrieren, das Problem auszudrücken ("was") und die Feinheiten der Lösung ("wie") an das Framework delegieren.

Generic Subdomains herauszufaktorisiern reduziert die Unübersichtlichkeit, und Cohesive Mechanisms dienen dazu, komplexe Vorgänge zu kapseln. Dies hinterlässt ein fokussierteres Modell mit weniger störenden Ablenkungen, welche die Benutzer bei ihren Aktivitäten keinen besonderen Wert bieten. Aber du wirst kaum einen guten Platz für alles im Domänenmodell finden, was nicht zum Kern gehört. Der Segregated Core verfolgt einen direkten Ansatz zur strukturellen Abgrenzung der Core Domain.

Segregated Core

Dt.: Abgetrennter Kern

Elemente im Modell können teilweise der Core Domain dienen und teilweise eine unterstützende Rolle spielen. Kernelemente können eng mit generischen gekoppelt sein. Der konzeptionelle Zusammenhang des Kerns ist möglicherweise nicht stark genug oder nicht sichtbar. All diese Unübersichtlichkeit und verworrenen Abhängigkeiten ersticken den Kern. Designer können die wichtigsten Beziehungen nicht klar erkennen, was zu einem schlechten Entwurf führt.

Daher:

Überarbeite das Modell, um die Kernkonzepte von unterstützenden Teilen (einschließlich schlecht definierten) zu trennen und den Zusammenhang des Kerns zu stärken und gleichzeitig seine Kopplung an den restlichen Code zu reduzieren. Verschiebe alle generischen oder unterstützenden Elemente in andere Objekte und platziere sie in anderen Paketen, auch wenn dies bedeutet, das Modell so zu überarbeiten, dass stark gekoppelte Elemente getrennt werden.

Abstract Core

Dt.: Abstrakter Kern

Selbst das Modell der Core Domain weist in der Regel so viele Details auf, dass es schwierig sein kann, das Gesamtbild zu vermitteln.

Wenn es zu viele Interaktionen zwischen Subdomänen in separaten Modulen gibt, müssen entweder viele Referenzen zwischen den Modulen erstellt werden, was einen Großteil des Wertes der Aufteilung zunichte macht, oder die Interaktion muss indirekt gemacht werden, was das Modell unklarer macht.

Daher:

Identifiziere die grundlegendsten differenzierenden Konzepte im Modell und überführe sie in eigenständige Klassen, abstrakte Klassen oder Schnittstellen. Entwerfe dieses abstrakte Modell so, dass es den größten Teil der Interaktion zwischen wichtigen Komponenten ausdrückt. Lege dieses abstrakte Gesamtmodell in ein eigenes Modul, während die spezialisierten, detaillierten Implementierungsklassen in ihren eigenen Modulen verbleiben, die durch die jeweilige Subdomain definiert sind.

VI. Large-scale Structure für Strategic Design

Dt.: Struktur im Großen für Strategic Design

Fehlt in einem großen System ein übergreifendes Prinzip, mit dem Elemente bezüglich ihrer Rollen anhand von für den gesamten Entwurf gültigen Mustern interpretiert werden können, dann werden Entwickler den Wald vor lauter Bäumen nicht sehen. Wir müssen in der Lage sein, die Rolle eines einzelnen Teils im Ganzen zu verstehen, ohne uns mit den Details des Ganzen zu befassen.

Eine "Large-scale Structure" ist eine Sprache, mit der man das System in groben Zügen diskutieren und verstehen kann. Einige übergeordnete Konzepte oder Regeln oder beides zusammen legen ein Muster für den Entwurf des gesamten Systems fest. Dieses Gestaltungsprinzip kann sowohl die Entwurfsarbeit als auch das Verständnis unterstützen. Es hilft dabei, unabhängige Arbeiten zu koordinieren, denn es gibt nun ein gemeinsames Konzept für das Gesamtbild, nämlich wie die Rollen der verschiedenen Teile das Ganze gestalten.

Daher:

Entwickle ein Muster, welches aus Regeln oder Rollen und Beziehungen besteht, welches das gesamte System erfasst und welches es ermöglicht, ein grundlegendes Verständnis der Lage jedes Teils im Ganzen zu erhalten, auch ohne detaillierte Kenntnis der Verantwortung dieses Teils zu besitzen.

Evolving Order

Dt.: Sich entwickelnde Ordnung

Ein Entwurf ohne Regeln bringt Systeme hervor, die niemand als Ganzes verstehen kann und die sehr schwer zu warten sind. Aber Architekturen können ein Projekt durch frühzeitige Annahmen über den Entwurf zu sehr einschränken und den Entwicklern/Designern der einzelnen Teile der Anwendung zu viel Macht entziehen. Entwickler werden die Anwendung rasch stark vereinfachen, um sie an die geforderte Struktur anzupassen, oder sie werden die Architektur untergraben und haben dann überhaupt keine Struktur, was die Probleme einer unkoordinierten Entwicklung wieder aufwirft.

Daher:

Sorge dafür, dass sich die konzeptionelle Large-scale Structure zusammen mit der Anwendung weiterentwickelt und sich dabei unterwegs möglicherweise in eine ganz andere Struktur verwandelt. Behindere nicht die detaillierten Entwurfs- und Modellentscheidungen, die mit detailliertem Wissen getroffen werden müssen.

Eine Large-scale Structure sollte angewendet werden, wenn eine Struktur gefunden werden kann, welche das System wesentlich klarer macht, ohne dabei unnatürliche Einschränkungen bei der Entwicklung des Modells zu erzwingen. Da eine schlecht passende Struktur schlechter ist als überhaupt keine, ist es am besten, nicht auf Vollständigkeit zu achten, sondern eine minimale Menge zu finden, welche die aufgetretenen Probleme löst. Weniger ist mehr.

Was folgt, ist eine Reihe von vier speziellen Mustern für die Large-scale Structure, die bei einigen Projekten entstehen und für diese Art von Muster repräsentativ sind.

System Metaphor

Dt.: System-Metapher

Metaphorisches Denken ist in der Softwareentwicklung weit verbreitet, insbesondere bei Modellen. Aber die Praktik der "Metapher" aus Extreme Programming hat mittlerweile die besondere Bedeutung bekommen, nämlich eine Metapher zu nutzen, um Ordnung in die Entwicklung eines ganzen Systems zu bringen.

Softwareentwürfe sind in der Regel sehr abstrakt und schwer zu verstehen. Sowohl Entwickler als auch Benutzer benötigen konkrete Möglichkeiten, das System zu verstehen und eine gemeinsame Sichtweise auf das Gesamtsystem miteinander zu teilen.

Daher:

Wenn eine konkrete Analogie zum System entsteht, welche die Phantasie der Teammitglieder einfängt und das Denken in eine nützliche Richtung zu lenken scheint, adaptiere sie als Large-scale Structure. Organisiere den Entwurf um diese Metapher herum und nimm sie in die Ubiquitous Language auf. Die System Metaphor sollte sowohl die Kommunikation über das System erleichtern als auch die Entwicklung des Systems leiten. Dies erhöht die Konsistenz in verschiedenen Teilen des Systems, möglicherweise sogar über verschiedene Bounded Contexts hinweg. Da aber alle Metaphern nicht exakt sind, überprüfe die Metapher ständig auf Übertreibungen oder Ungeeignetheit und sei dazu bereit, sie fallen zu lassen, wenn sie dir im Weg steht.

Responsibility Layers

Dt.: Schichten nach Zuständigkeiten

Bei objekt-orientierten Entwürfen werden den einzelnen Objekten kleine Mengen zusammenhängender Zuständigkeiten zugewiesen. Der Entwurf anhand von Zuständigkeiten betrifft aber auch größere Einheiten.

Wenn jedes einzelne Objekt einzeln zugewiesene Verantwortlichkeiten hat, dann gibt es keine Richtlinien, keine Einheitlichkeit und keine Möglichkeiten, größere Teile der Domäne zusammen zu bearbeiten. Um einem großen Modell Zusammenhalt zu verleihen, ist es sinnvoll, der Zuweisung von Verantwortlichkeiten eine gewisse Struktur aufzuerlegen.

Daher:

Betrachte die konzeptionellen Abhängigkeiten im Modell und die unterschiedlichen Änderungsgeschwindigkeiten und -quellen der verschiedenen Teile der Domäne. Wenn du natürliche Schichten in der Domäne identifizierst, dann entwerfe sie als grobe und abstrakte Verantwortlichkeiten. Diese Verantwortlichkeiten sollten eine Geschichte über den Zweck und den Entwurf deines Systems auf grober Ebene erzählen. Überarbeite das Modell so, dass die Verantwortlichkeiten der einzelnen Domänenobjekte, Aggregates und Module genau in die Verantwortung einer Schicht passen.

Knowledge Level

Dt.: Wissensebene

Eine Gruppe von Objekten, die beschreiben, wie sich eine andere Gruppe von Objekten verhalten soll.

In einer Anwendung, in der die Rollen und Beziehungen zwischen den Entities in verschiedenen Situationen unterschiedlich sind, kann die Komplexität explodieren. Weder ganz allgemeine noch sehr individuelle Modelle erfüllen die Bedürfnisse der Anwender. Am Ende haben Objekte Referenzen auf andere Typen, um eine Vielzahl von Fällen abzudecken, oder sie haben Attribute, die in verschiedenen Situationen unterschiedlich verwendet werden. Klassen, welche gleiche Daten und das gleiche Verhalten haben, können mehrfach vorkommen, nur um unterschiedliche Regeln bei der Zusammensetzung der Objekte zu berücksichtigen.

Daher:

Erstelle eine eigene Gruppe von Objekten, welche die Struktur und das Verhalten des Basismodells beschreiben und einschränken können. Halte diese Belange als zwei "Ebenen" getrennt: die eine ist sehr konkret, die andere spiegelt Regeln und Wissen wider, die ein Benutzer oder Administrator anpassen kann.

(siehe *Fowler, M. 1997. Analysis Patterns: Reusable Object Models, Addison-Wesley*)

Pluggable Component Framework

Dt.: Zusammensteckbares Komponenten-Framework

Ein sehr ausgereiftes Modell, das tief und destilliert ist, bietet viele Möglichkeiten. Ein Pluggable Component Framework kommt in der Regel erst ins Spiel, wenn bereits einige Anwendungen in der gleichen Domäne implementiert wurden.

Wenn eine Vielzahl von Anwendungen zusammenarbeiten müssen, die alle auf den gleichen Abstraktionen basieren, aber unabhängig voneinander entworfen worden sind, schränken Übersetzungen zwischen mehreren Bounded Contexts die Integration ein. Ein Shared Kernel kann nur mit Teams realisiert werden, die eng zusammenarbeiten. Doppelte Arbeit und Fragmentierung erhöhen die Kosten für Entwicklung und Installation, und die Interoperabilität wird sehr schwierig.

Daher:

Destilliere einen abstrakten Kern von Schnittstellen und Interaktionen heraus und entwickle ein Framework, das es ermöglicht, verschiedene Implementierungen dieser Schnittstellen auszutauschen. Ebenso solltest du jeder Anwendung erlauben, diese Komponenten zu verwenden, solange sie ausschließlich über die Schnittstellen des abstrakten Kerns arbeitet.