



Vaughn Vernon

Domain-Driven Design kompakt

→ Aus dem Englischen übersetzt
von Carola Lilienthal und Henning Schwentner

dpunkt.verlag



Inhalt

Cover

Titel

Impressum

Vorbemerkung der Übersetzer

Vorwort

Danksagung

Inhaltsverzeichnis

1 DDD für mich

Wird DDD wehtun?

Gutes, schlechtes und effektives Design

Strategisches Design

Taktisches Design

Lernprozess und Wissensvertiefung

Legen wir los!

2 Strategisches Design mit Bounded Contexts und der Ubiquitous Language

Domain Experts und Geschäftstreiber

Fallstudie

Grundlegendes strategisches Design ist notwendig

Infragestellen und Vereinheitlichen

Eine Ubiquitous Language entwickeln

 Szenarios umsetzen

 Und auf lange Sicht?

Architektur

Zusammenfassung

3 Strategisches Design mit Subdomains

Was ist eine Subdomain?

Arten von Subdomains

Mit Komplexität umgehen

Zusammenfassung

4 Strategisches Design mit Context Mapping

Arten von Mapping

Partnership

Shared Kernel

Customer-Supplier

Conformist

Anticorruption Layer

Open Host Service

Published Language

Separate Ways

Big Ball of Mud

Context Mapping richtig nutzen

RPC mit SOAP

RESTful HTTP

Messaging

Context Mapping am Beispiel

Zusammenfassung

5 Taktisches Design mit Aggregates

Verwendung

Daumenregeln für Aggregates

Regel 1: Schütze fachliche Invarianten innerhalb von Aggregate-Grenzen

Regel 2: Entwirf kleine Aggregates

Regel 3: Referenziere andere Aggregates nur über ihre Identität

Regel 4: Aktualisiere andere Aggregates unter Verwendung von Eventual Consistency

Aggregates modellieren

Abstraktionen mit Bedacht wählen

Aggregates richtig dimensionieren

Testbare Einheiten

Zusammenfassung

6 Taktisches Design mit Domain Events

Domain Events entwerfen, implementieren und verwenden

Event Sourcing

Zusammenfassung

7 Beschleunigungs- und Managementtechniken

Event Storming

Andere Techniken

DDD in einem agilen Projekt managen

Das Wichtigste zuerst

SWOT-Analyse einsetzen

Modellierungs-Spikes und Modellierungsschuld

Tasks identifizieren und Aufwand schätzen

Modellieren mit Timebox

Wie man implementiert

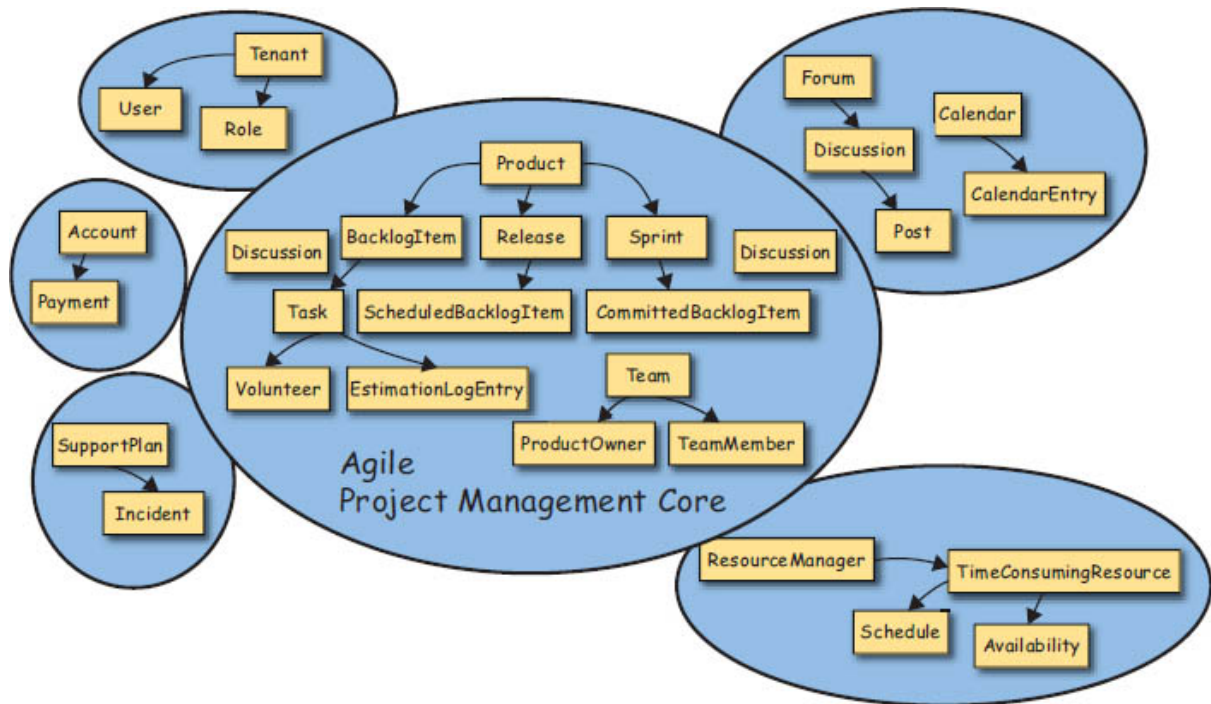
Mit Domain Experts zusammenarbeiten

Zusammenfassung

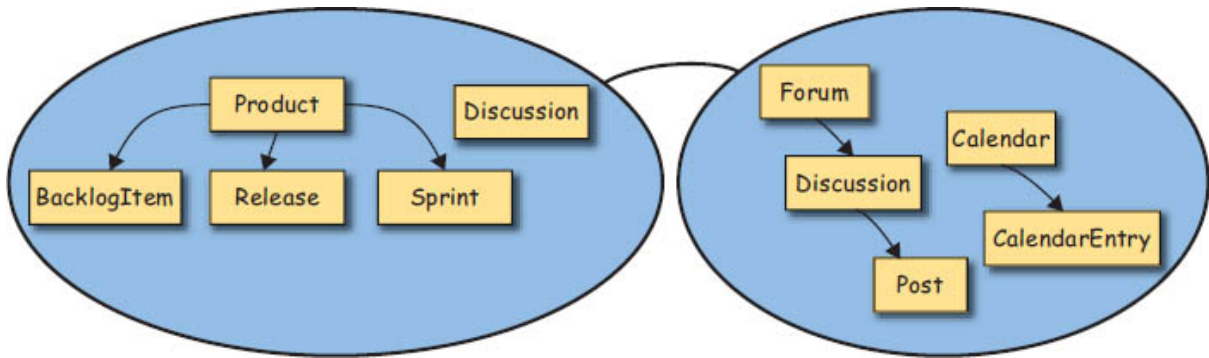
A Literatur

Index

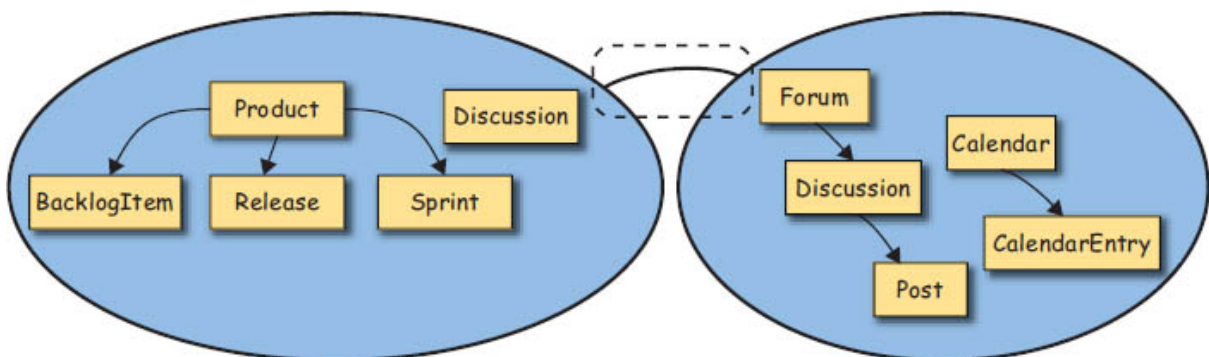
4 Strategisches Design mit Context Mapping



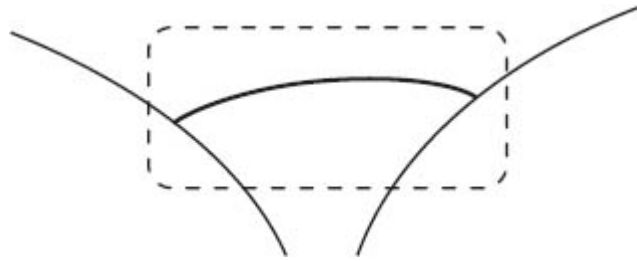
In den vorhergehenden Kapiteln haben Sie gelernt, dass in jedem DDD-Projekt zusätzlich zur *Core Domain* (dt.: *Kerndomäne, zentrale Teildomäne*) mehrere andere *Bounded Contexts* (dt.: *begrenzte Kontexte*) existieren. Alle Konzepte, die nicht in den Kontext *Agile Project Management Core* – die *Core Domain* – gehören, wurden in verschiedene andere *Bounded Contexts* verschoben.



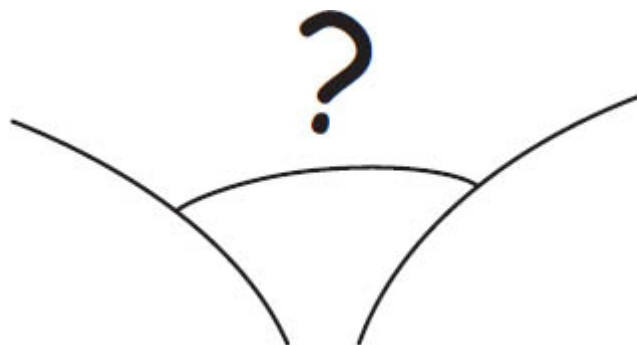
Sie haben auch gelernt, dass die *Core Domain* Agile Project Management Core mit anderen *Bounded Contexts* integriert werden muss. Diese Integration wird in *DDD Context Mapping* (dt.: *Abbilden von Kontexten*) genannt. Sie können in der vorangehenden *Context Map* (dt.: *Kontextlandkarte*) sehen, dass *Discussion* in beiden *Bounded Contexts* existiert. Wir erinnern uns daran, dass der Kontext *Collaboration* die Quelle der *Discussion* ist und der Kontext *Agile Project Management Core* der Verwender der *Discussion*.



Das *Context Mapping* ist in der Abbildung mit dem gestrichelten Kasten hervorgehoben. (Der gestrichelte Kasten ist nicht Teil des *Context Mapping*, sondern wird nur verwendet, um den Strich hervorzuheben.) Tatsächlich repräsentiert der Strich das *Context Mapping*. Mit anderen Worten: Der Strich zeigt an, dass die beiden *Bounded Contexts* in irgendeiner Weise aufeinander abgebildet werden. Es wird eine Dynamik zwischen den beiden Teams und eine Integration zwischen den beiden *Bounded Contexts* geben.



Nehmen wir einmal an, dass in zwei unterschiedlichen *Bounded Contexts* auch zwei *Ubiquitous Languages* existieren, dann repräsentiert der Strich die Übersetzung, die zwischen den beiden Sprachen erfolgt. Zur Illustration stellen Sie sich zwei Teams vor, die über Landes- und Sprachgrenzen hinweg zusammenarbeiten müssen. Entweder brauchen die Teams einen Übersetzer oder eines oder beide Teams müssten eine Menge über die Sprache des anderen Teams lernen. Einen Übersetzer zu finden, wäre weniger Arbeit für beide Teams, aber es könnte auf verschiedene Arten teuer werden. Stellen Sie sich z. B. die Extrazeit vor, die das eine Team braucht, um mit dem Übersetzer zu sprechen, und die der Dolmetscher dann benötigt, um die Aussagen dem anderen Team zu überbringen. Am Anfang funktioniert das möglicherweise ganz gut, aber mit der Zeit wird es mühsam. Trotzdem bevorzugen die Teams vielleicht diese Lösung, statt eine fremde Sprache zu lernen und fortwährend zwischen zwei Sprachen hin und her zu springen. Im Moment betrachten wir nur zwei Teams. Was, wenn noch mehr Teams involviert sind? Weitere Probleme entstehen, wenn man eine *Ubiquitous Language* in eine andere übersetzt oder versucht, sie an eine weitere *Ubiquitous Language* anzupassen.



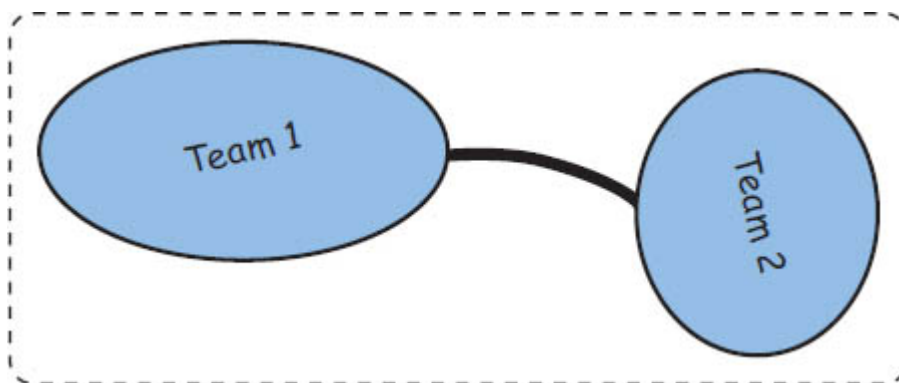
Spannend wird nun, wenn wir über *Context Mapping* sprechen, für welche Art von Beziehung und Integration zwischen Teams der Strich zwischen zwei *Bounded Contexts* steht. Sinnvoll ist es, Grenzen und Verträge zwischen den *Bounded Contexts* zu definieren. So kann man kontrollierte Anpassungen über die Zeit unterstützen. Es gibt verschiedene Arten von *Context Mappings*, sowohl auf Team- als auch auf technischer Ebene, die von dem Strich repräsentiert

werden können. Diese verschiedenen Arten des Mapping treten nicht nur einzeln auf, sondern können auch gemischt eingesetzt werden.

Arten von Mapping

Welche Beziehungen und Integrationen können von der *Context-Mapping*-Linie repräsentiert werden? Im Folgenden werden sie Ihnen vorgestellt.

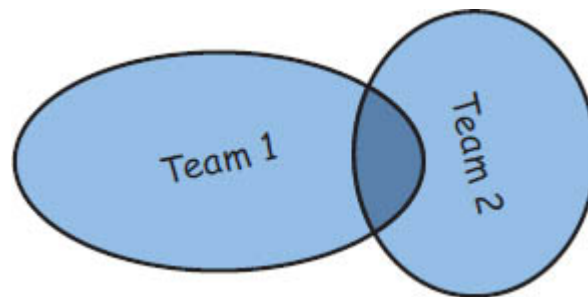
Partnership



Partnership (dt.: *Partnerschaft*) besteht zwischen zwei Teams, die jeweils für einen *Bounded Context* verantwortlich sind. Sie sollten eine *Partnership*-Beziehung zwischen zwei Teams einführen, um zwei Teams mit voneinander abhängigen Zielen zu koordinieren. In einem solchen Fall können die beiden Teams nur zusammen Erfolg (oder Misserfolg) haben. Weil sie so eng zusammenarbeiten, werden sie sich regelmäßig treffen, um Pläne und abhängige Arbeit zu synchronisieren. Die beiden Teams werden Continuous Integration nutzen müssen, um ihre Softwarestände harmonisch zu halten. Die Synchronisation wird durch die dicke Abbildungslinie zwischen den beiden Teams dargestellt. Die Dicke der Linie zeigt das notwendige Niveau von Engagement: Es ist ziemlich hoch.

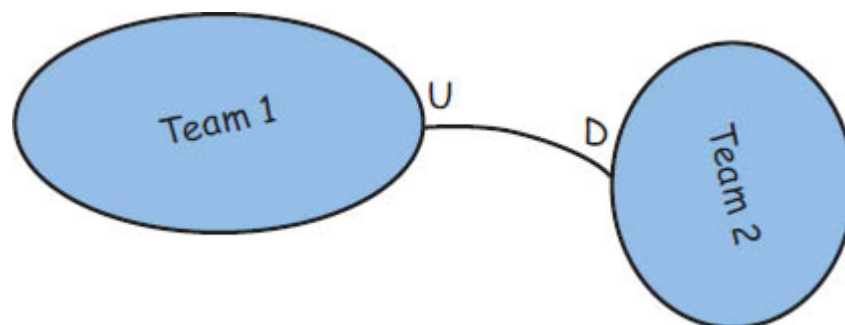
Es kann eine Herausforderung sein, *Partnership* über eine lange Zeit zu pflegen, deshalb tun Teams, die *Partnership* eingehen, gut daran, von vornherein ein Ende für die Beziehung festzusetzen. *Partnership* sollte nur so lange bestehen, wie sie einen Vorteil bietet. Sobald der Vorteil kleiner wird als die Verluste, die durch weniger eng zusammenhängende Teams entstehen würden, sollte *Partnership* in eine andere Art von Beziehung geändert werden.

Shared Kernel



Ein *Shared Kernel* (dt.: *geteilter Kern*), hier als Schnittmenge von zwei *Bounded Contexts* dargestellt, beschreibt die Beziehung zwischen zwei (oder mehr) Teams, die sich ein kleines, aber gemeinsames Modell teilen. Die Teams müssen sich darüber einigen, welche Modellelemente sie teilen wollen. Es ist möglich, dass nur eines der Teams die Pflege von Code, Build und Test für das, was geteilt wird, übernimmt. Ein *Shared Kernel* ist anfangs oft schwer zu erzeugen und schwierig zu pflegen, weil man eine offene Kommunikation zwischen den Teams und eine dauerhafte Einigkeit darüber erreichen muss, was zum geteilten Modell gehört. Trotzdem ist es möglich, einen *Shared Kernel* erfolgreich einzusetzen, wenn alle Beteiligten überzeugt sind, dass der Kern besser ist, als *Separate Ways* (dt.: *getrennte Wege*, siehe unten) zu gehen.

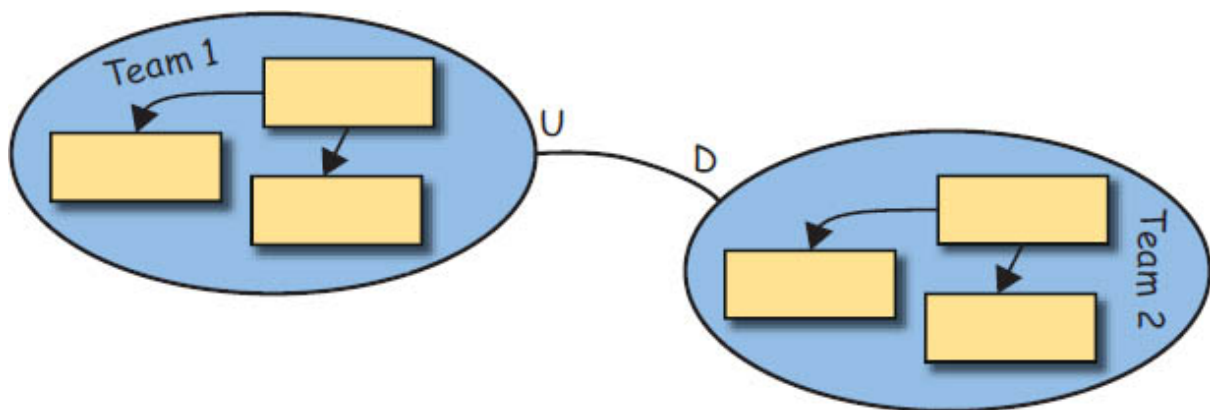
Customer-Supplier



Das Muster *Customer-Supplier* (dt.: *Kunde-Lieferant*) beschreibt eine Beziehung zwischen zwei *Bounded Contexts* und den dazugehörigen Teams, wobei der *Supplier* (dt.: *Lieferant*) vorgeschaltet (engl.: *upstream*, deshalb mit U in der Abbildung gekennzeichnet) und der *Customer* (dt.: *Kunde*) nachgeschaltet (engl.: *downstream*, D in der Abbildung) ist. Diese Beziehung hängt vom *Supplier* ab, weil er zur Verfügung stellen muss, was der *Customer* braucht. Es liegt beim *Customer*, mit dem *Supplier* zu planen, wie die unterschiedlichen Erwartungen erfüllt werden können, aber am Ende bestimmt der *Supplier*, was der *Customer*

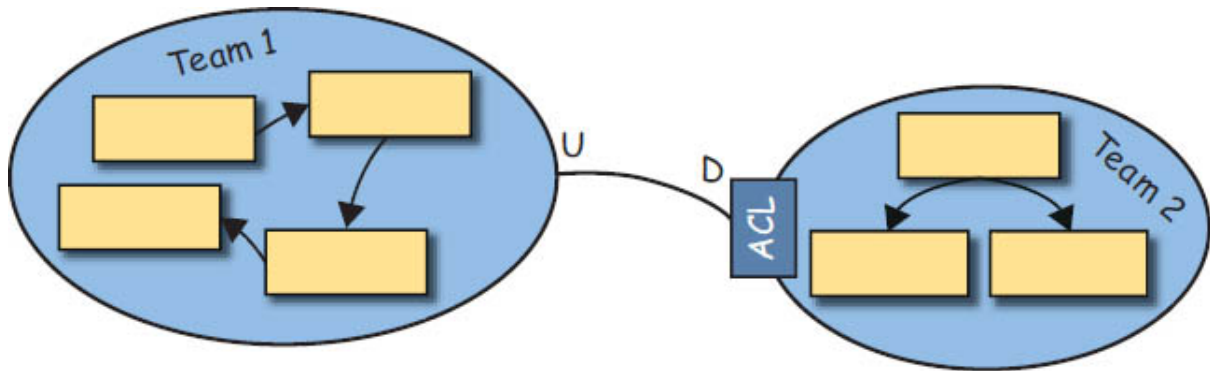
bekommen wird und wann. Dies ist eine sehr typische und praktische Beziehung zwischen Teams, sogar innerhalb derselben Organisation, solange die Unternehmenskultur dem *Supplier* nicht erlaubt, völlig eigenständig und ignorant gegenüber den Bedürfnissen des *Customer* zu sein.

Conformist



Eine *Conformist*-Beziehung (dt.: *Konformist*, *Mitläufer*) besteht, wenn es ein Upstream- und ein Downstream-Team gibt, und das Upstream-Team keine Motivation hat, die Anforderungen des Downstream-Teams zu unterstützen. Aus verschiedenen Gründen kann das Downstream-Team sich den Aufwand, die *Ubiquitous Language* des Upstream-Modells passend zu seinen Bedürfnissen zu übersetzen, nicht leisten und passt sich deshalb (engl.: *conforms to*) dem Upstream-Modell an. Viele Teams werden insbesondere dann zu *Conformists*, wenn sie sich in ein sehr großes, komplexes Modell integrieren müssen, das bereits gut etabliert ist. Beispiel: Wenn Sie Amazon-Partner werden wollen, werden Sie sich dem Amazon-Modell anpassen müssen, um sich zu integrieren.

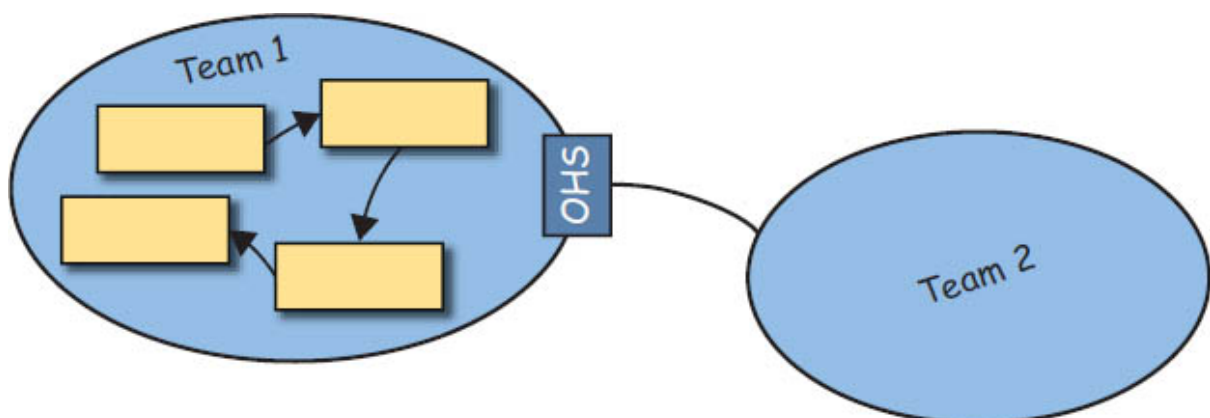
Anticorruption Layer



Ein *Anticorruption Layer* (in der Abbildung ACL, dt.: *Antikorrupsionsschicht*, *Antiverfälschungsschicht*) ist die defensivste Art einer *Context-Mapping*-Beziehung. Hier erzeugt das Downstream-Team eine Übersetzungsschicht zwischen seiner *Ubiquitous Language* (Modell) und der *Ubiquitous Language* (Modell) des Upstream-Teams. Diese Schicht isoliert das Downstream-Modell vom Upstream-Modell und übersetzt zwischen den beiden. Aus diesem Grund ist die Verwendung eines *Anticorruption Layer* auch ein Integrationsansatz.

Wann immer es möglich ist, sollte man versuchen, einen *Anticorruption Layer* zwischen dem Downstream-Modell und einem integrierenden Upstream-Modell zu erzeugen. Dadurch bekommt man die Möglichkeit, auf der eigenen Seite der Integration Modellkonzepte zu bauen, die exakt die eigenen Anforderungen befriedigen und von fremden Konzepten isoliert sind. Der Preis für den *Anticorruption Layer* kann je nach Umfang des *Anticorruption Layer* unterschiedlich hoch sein. Genau wie der Preis für einen Dolmetscher, der zwischen zwei Teams mit unterschiedlicher Sprache übersetzt.

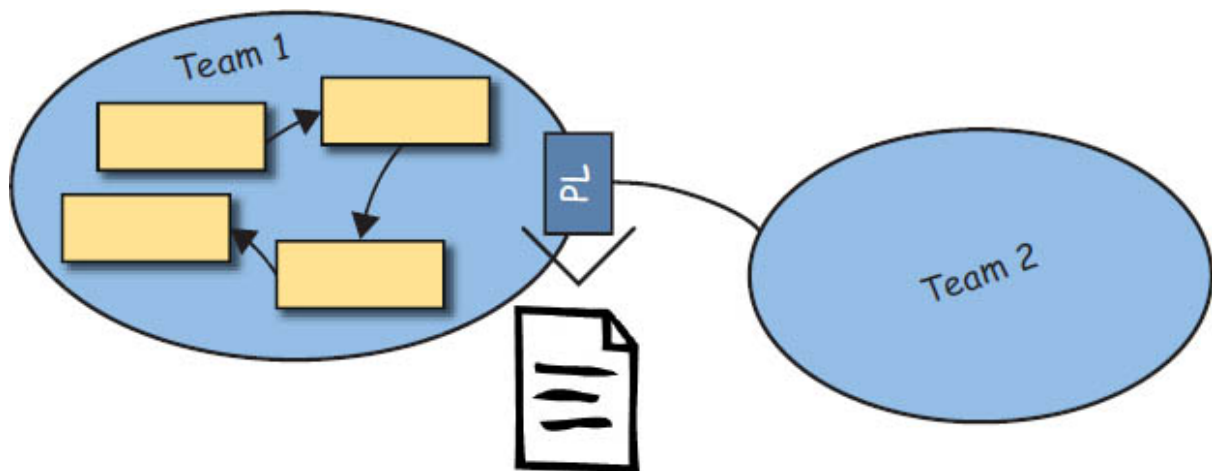
Open Host Service



Ein *Open Host Service* (in der Abbildung OHS, dt.: *offen angebotener Dienst*) definiert ein Protokoll oder eine Schnittstelle, das oder die Zugriff zu einem

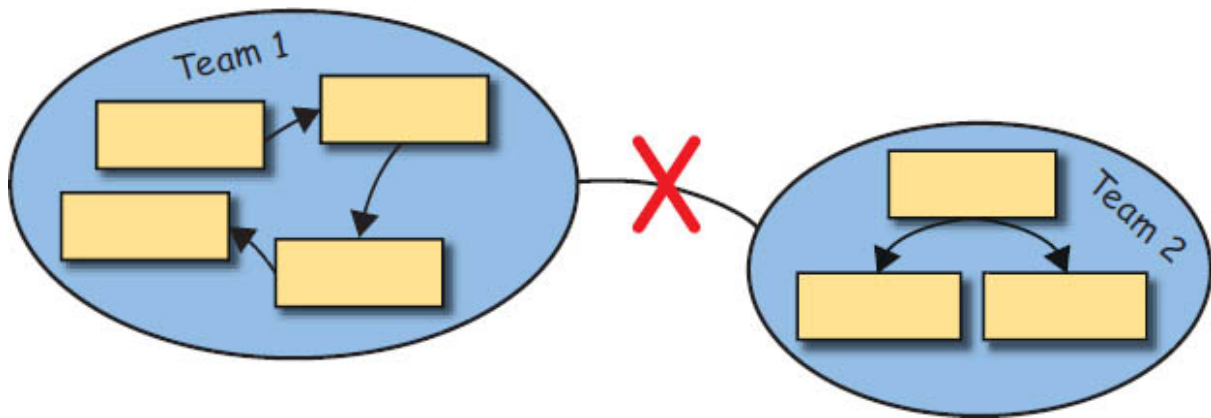
Bounded Context in Form einer Menge von Services bietet. Das Protokoll ist »offen« (open) in dem Sinne, dass jeder, der den *Bounded Context* benutzen muss, relativ leicht mit ihm interagieren kann. Die vom API bereitgestellten Services sollten wohldokumentiert sein und es sollte Spaß machen, sie zu verwenden. Selbst wenn Sie Team 2 in dieser Abbildung wären und keine Zeit hätten, um einen *Anticorruption Layer* auf Ihrer Seite zu bauen, ist es viel angenehmer, ein *Conformist* dieses Modell zu sein als ein *Conformist* von vielen Altsystemen (engl.: legacy systems), denen man sonst so begegnen kann. Man könnte sagen, dass die Sprache des *Open Host Service* viel leichter zu konsumieren ist als die von anderen Arten von Systemen.

Published Language



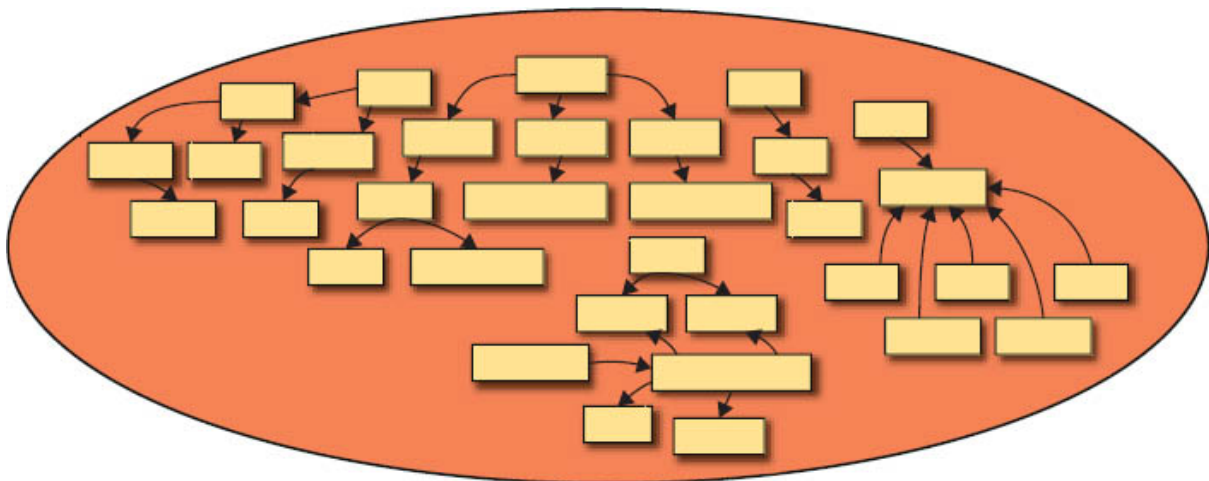
Eine *Published Language* (in der Abbildung PL, dt.: *veröffentlichte Sprache*) ist eine wohldokumentierte Sprache zum Informationsaustausch, die eine einfache Verwendung und Übersetzung von einer beliebigen Anzahl von *Bounded Contexts* ermöglicht. Konsumenten, die sowohl lesen als auch schreiben, können sich darauf verlassen, dass ihre Integration korrekt ist, wenn sie in die gemeinsame Sprache übersetzen. Solch eine *Published Language* kann in XML Schema, JSON Schema oder einem platzsparenden Format wie Protobuf oder Avro definiert werden. Oft bietet ein *Open Host Service* eine *Published Language* als Schnittstelle an. Durch diese Kombination wird die Übersetzung zwischen zwei *Ubiquitous Languages* sehr einfach.

Separate Ways

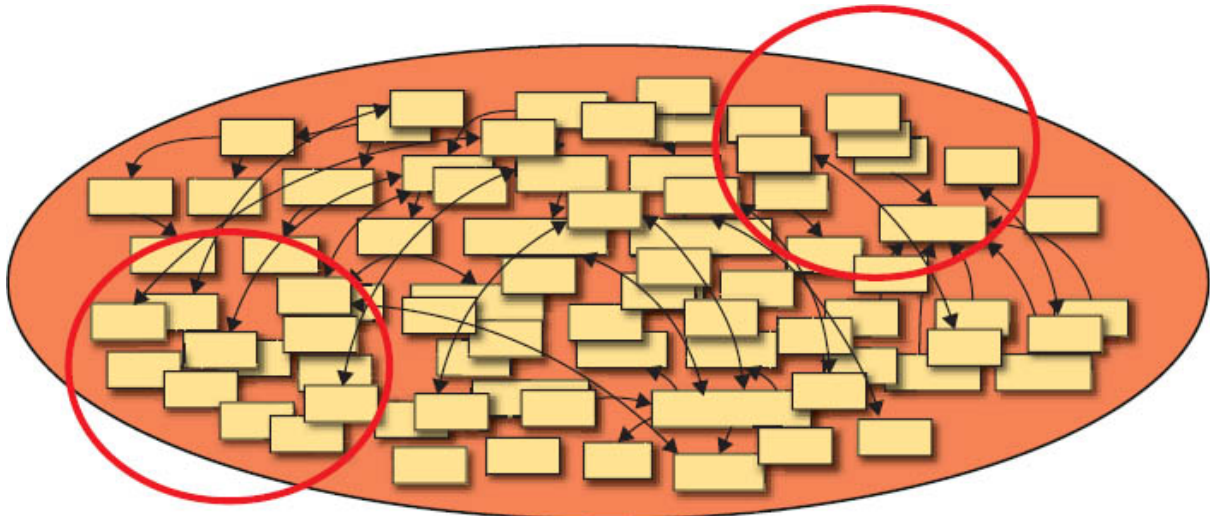


Separate Ways (dt.: *getrennte Wege*) beschreibt eine Situation, in der sich kein signifikanter Nutzen darin erkennen lässt, dass Sie die Integration mit einem oder mehreren *Bounded Contexts* und den dort vorhandenen, verschiedenen *Ubiquitous Languages* vornehmen. Vielleicht wird die von Ihnen gesuchte Funktionalität von keiner *Ubiquitous Language* komplett angeboten. In diesem Fall erzeugen Sie sich Ihre eigene spezialisierte Lösung in einem eigenen *Bounded Context* und unterlassen das Integrieren für diesen Spezialfall.

Big Ball of Mud

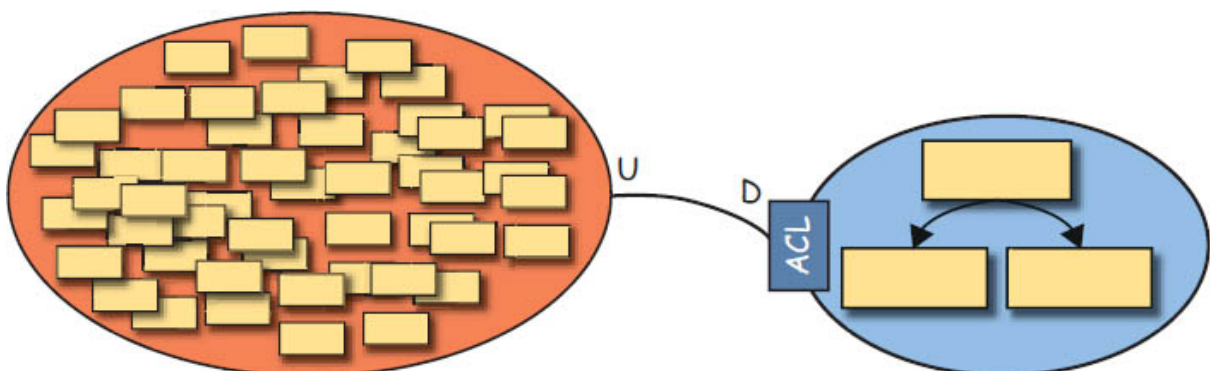


In den vorhergehenden Kapiteln haben Sie schon vieles über den *Big Ball of Mud* (dt.: *große Matschkugel, großes verworrenes Knäuel*) gelernt. Nichtsdestotrotz will ich hier noch einmal die schwerwiegenden Probleme beschreiben, die Sie erleben werden, wenn Sie in einem *Big Ball of Mud* arbeiten oder mit ihm interagieren müssen. Einen eigenen *Big Ball of Mud* zu erzeugen, sollten Sie meiden wie die Pest.



Falls das nicht Warnung genug ist, hier eine Aufzählung der Probleme, die mit der Zeit auftreten werden, wenn Sie einen *Big Ball of Mud* erzeugt haben:

1. Eine zunehmende Anzahl von *Aggregates* (dt.: *Aggregate*) verunreinigen sich gegenseitig wegen ungewollter und unzulässiger Beziehungen und Abhängigkeiten.
2. Eine Änderung an einem Teil des *Big Ball of Mud* schlägt Wellen über das ganze Modell, was zu Problemen der Art »Zieht man am einen Ende, dann zappelt es am anderen« führt.
3. Nur über Generationen weitergegebenes Geheimwissen und einzelne Heldentaten – alle Sprachen auf einmal sprechen – retten das System vor dem endgültigen Kollaps.

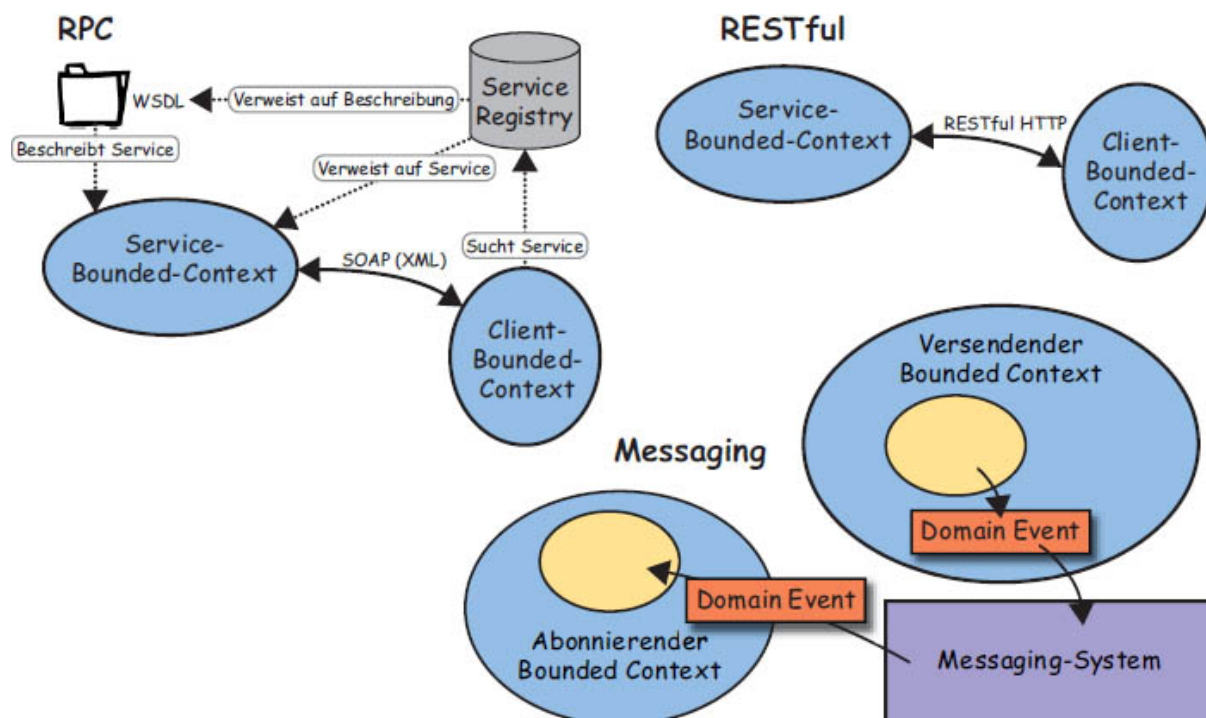


Das Problem ist, dass es da draußen in der freien Wildbahn der Softwaresysteme schon viele *Big Balls of Mud* gibt und dass ihre Zahl zweifellos jeden Monat steigt. Selbst wenn Sie es schaffen, mithilfe von DDD-Techniken die Erzeugung eines *Big Ball of Mud* zu verhindern, kann es immer noch sein, dass Sie mit einem oder mehreren zusammenarbeiten müssen. Wenn Sie diese Aufgabe vor

sich sehen, dann versuchen Sie, einen *Anticorruption Layer* gegen jedes dieser *Big-Ball-of-Mud*-Altsysteme zu erzeugen. Ihr Ziel sollte dabei sein, Ihr eigenes Modell vor dem unverständlichen Morast zu schützen, der Ihr Modell andernfalls verschmutzen würde. Was immer Sie auch tun, *übernehmen Sie nicht diese Sprache!*

Context Mapping richtig nutzen

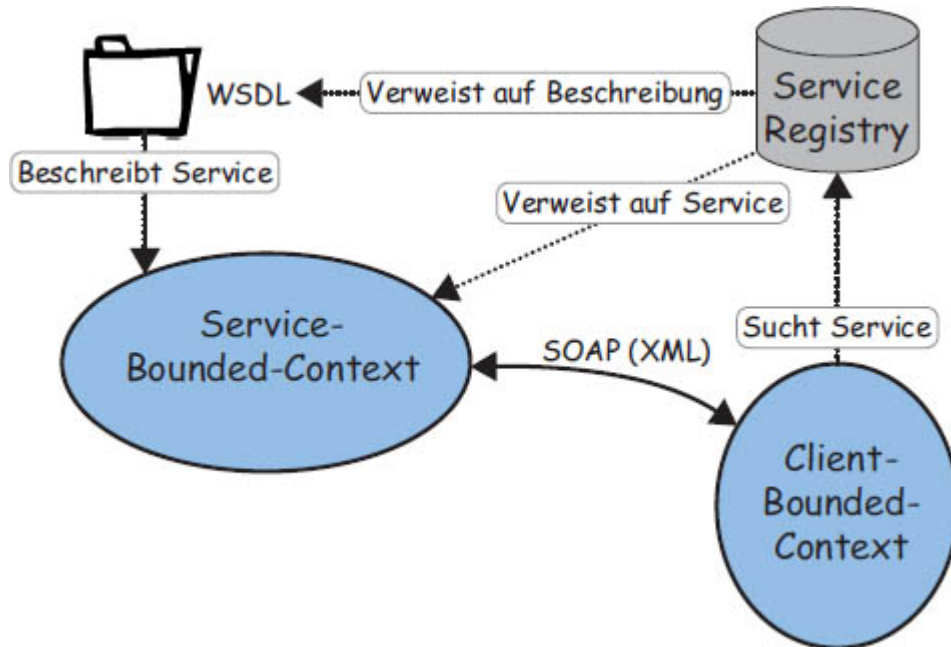
Wahrscheinlich fragen Sie sich, welche Art von Schnittstelle angeboten werden sollte, um Ihnen die Integration mit einem gegebenen *Bounded Context* zu ermöglichen. Das hängt davon ab, was das Team anbietet, dem der *Bounded Context* gehört. Es könnte RPC über SOAP oder eine RESTful API mit Ressourcen oder eine Nachrichtenschnittstelle mit einem Messaging-System, das Queues oder Publish-Subscribe verwendet, sein. Im schlechtesten Fall sind Sie dazu gezwungen, Datenbank- oder Dateisystemintegration zu verwenden, aber lassen Sie uns hoffen, dass das nicht passiert. Datenbankintegration sollte auf jeden Fall vermieden werden! Wenn Sie zu dieser Art der Integration gezwungen werden, sollten Sie wirklich sicherstellen, Ihr Modell durch einen *Anticorruption Layer* zu isolieren.



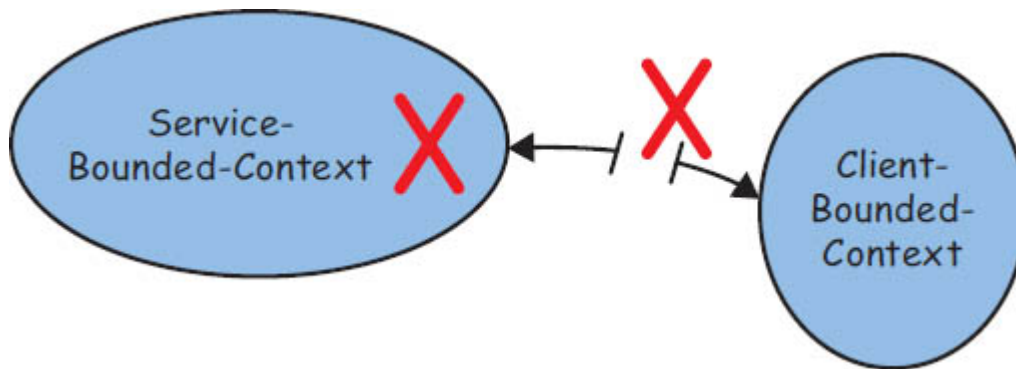
Lassen Sie uns einen Blick auf drei vertrauenswürdige Integrationstypen werfen. Wir werden beim weniger robusten Ansatz anfangen und uns zu den robustesten

Ansätzen vorarbeiten. Zunächst betrachten wir RPC, gefolgt von RESTful HTTP und Messaging.

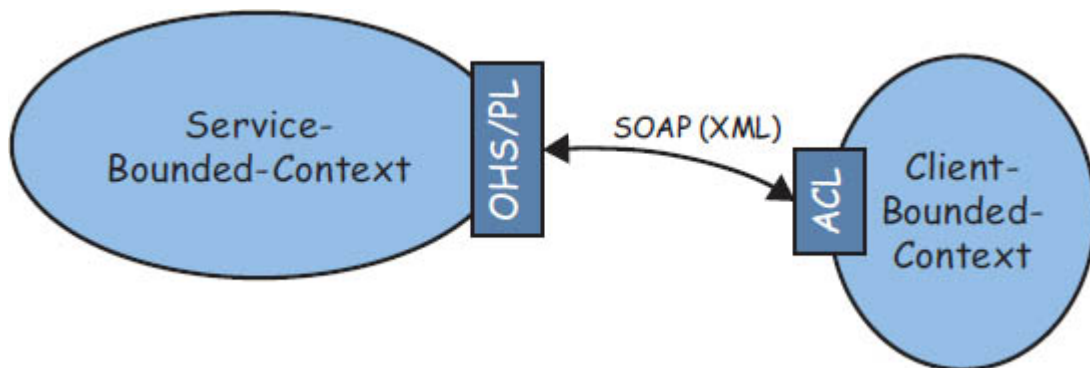
RPC mit SOAP



Remote Procedure Call, kurz RPC, kann auf verschiedene Arten umgesetzt werden. Häufig wird RPC über das Simple Object Access Protocol, kurz SOAP, genutzt. Die Idee hinter RPC mit SOAP ist es, das Aufrufen von Diensten eines anderen Systems so aussehen zu lassen wie ein einfacher lokaler Prozedur- oder Methodenaufruf. Trotzdem muss die SOAP-Anfrage über das Netz transportiert werden, das entfernte System erreichen, die Anfrage erfolgreich ausgeführt und das Ergebnis über das Netz zurückgeliefert werden. Das beinhaltet das Risiko von Netzausfall oder zumindest von größerer Latenzzeit, als man sie bei der ersten Implementierung möglicherweise vorhergesehen hat. Des Weiteren impliziert RPC über SOAP auch eine enge Kopplung zwischen einem *Client-Bounded-Context* und dem *Service-Bounded-Context*, der die Dienstleistung anbietet.

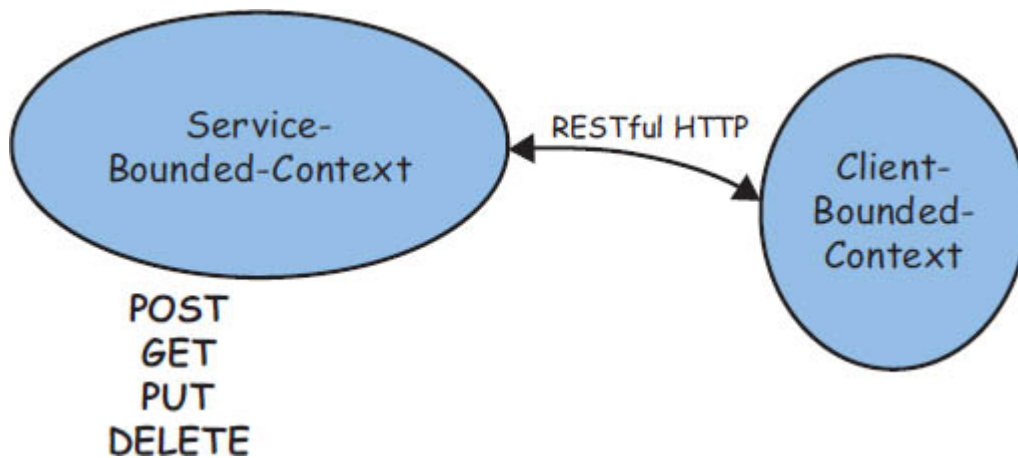


Das Hauptproblem mit RPC, sei es nun mit SOAP oder einem anderen Ansatz, ist der Mangel an Robustheit. Wenn es ein Problem mit dem Netz oder ein Problem mit dem System, das die SOAP API anbietet, gibt, wird der vermeintlich einfache Prozeduraufruf fehlschlagen und lediglich einen Fehler als Ergebnis liefern. Lassen Sie sich nicht vom scheinbar einfachen Verwenden von RPC zum Narren halten.

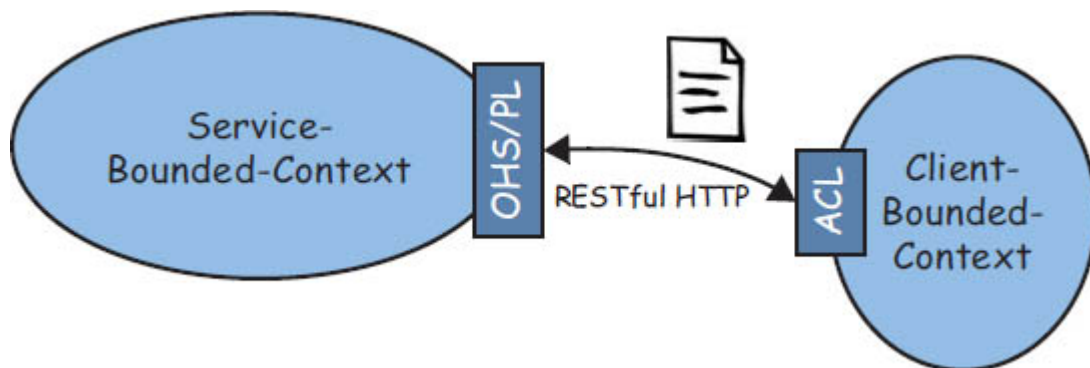


Wenn RPC funktioniert – und meistens funktioniert es – kann es ein sehr nützlicher Weg zur Integration sein. Wenn Sie das Design des dienstleistenden *Bounded Context* beeinflussen können, versuchen Sie eine sauber entworfene API zu erreichen, die einen *Open Host Service* mit einer *Published Language* anbietet. In jedem Fall kann Ihr *Bounded Context* mit einem *Anticorruption Layer* versehen werden, um Ihr Modell von ungewollten äußeren Einflüssen zu isolieren.

RESTful HTTP

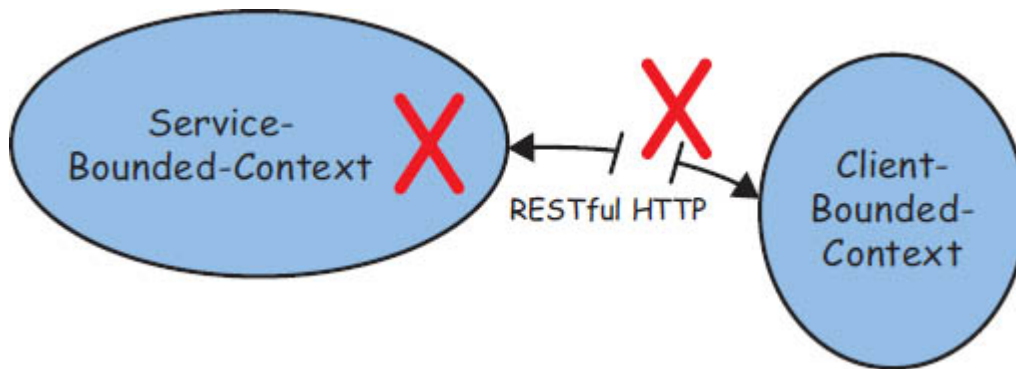


Die Integration mit RESTful HTTP lenkt die Aufmerksamkeit auf die Ressourcen, die zwischen *Bounded Contexts* ausgetauscht werden, sowie auf die vier Primäroperationen: POST, GET, PUT, DELETE. Viele finden, dass der REST-Ansatz zur Integration gut funktioniert, weil er ihnen hilft, gute APIs für verteilte Systeme zu definieren. Es ist schwierig dagegen zu argumentieren, wenn man den Erfolg des Internets und des Web betrachtet.

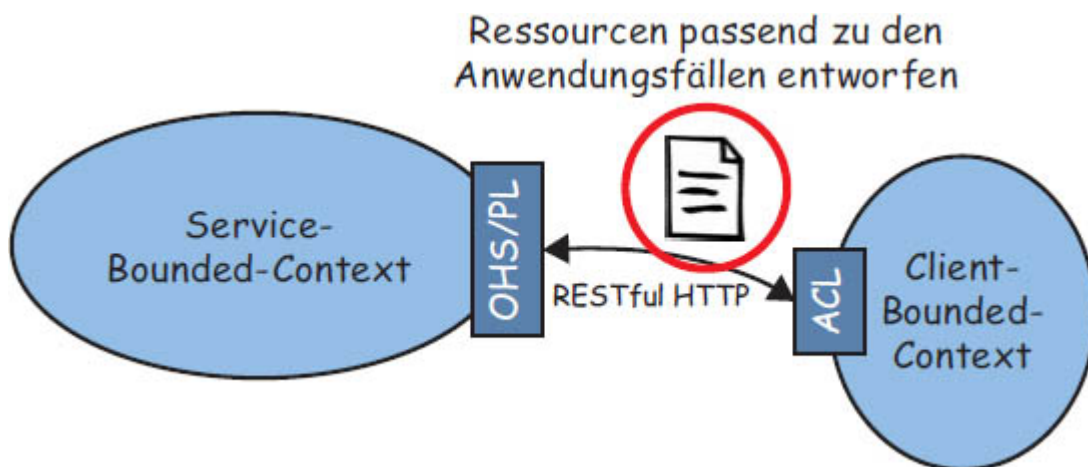


Bei der Verwendung von RESTful HTTP wird eine bestimmte Denkweise vorausgesetzt. Wir werden hier nicht ins Detail gehen, aber Sie sollten sich die Grundlagen von REST aneignen, bevor Sie versuchen, es einzusetzen. Das Buch *REST in Practice* [Webber et al. 2010] ist ein guter Ausgangspunkt.

Ein *Service-Bounded-Context*, der eine REST-Schnittstelle anbietet, sollte einen *Open Host Service* und eine *Published Language* zur Verfügung stellen. Ressourcen verdienen als *Published Language* definiert zu werden. Zusammen mit Ihren REST-URIs werden sie einen natürlichen *Open Host Service* bilden.

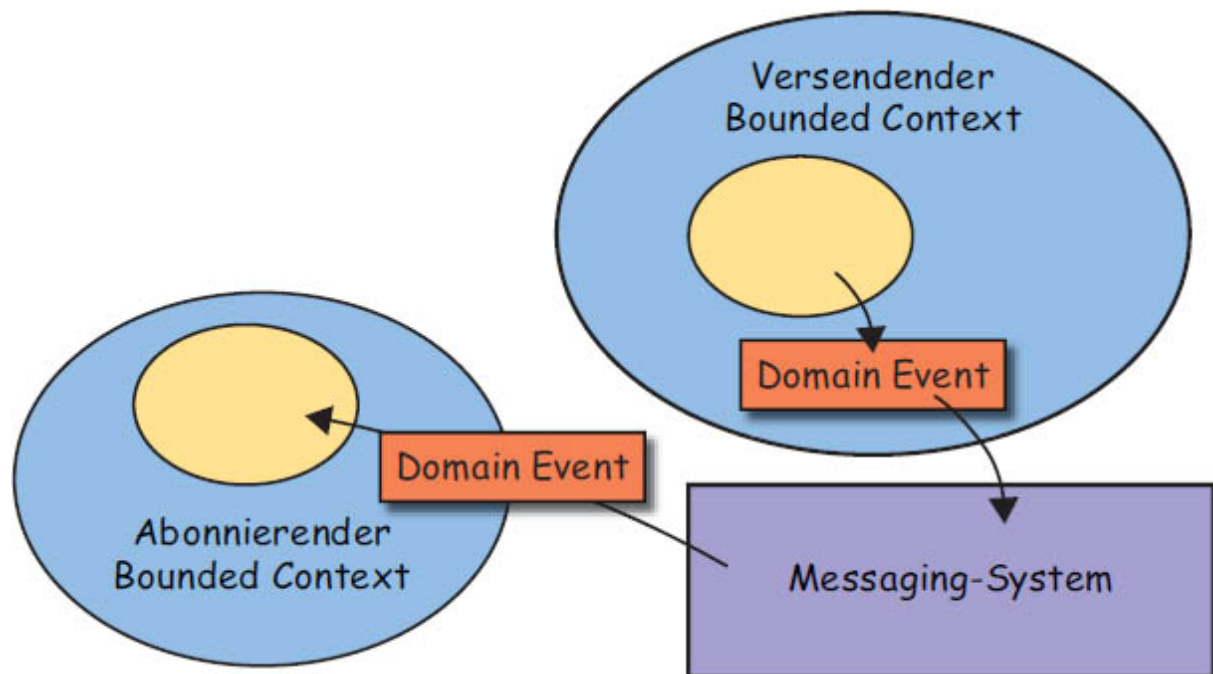


RESTful HTTP tendiert dazu, aus den gleichen Gründen fehlschlagen wie RPC – durch Ausfälle von Netz oder Servern oder eine unvorhergesehene große Latenzzeit. Nichtsdestotrotz: RESTful HTTP baut auf der Stärke des Internets auf. An der Erfolgsgeschichte des Web lässt sich kaum ein Makel finden, wenn es um Zuverlässigkeit, Skalierbarkeit und Gesamterfolg geht.



Ein häufiger Fehler bei der Verwendung von REST besteht darin, die Ressourcen so zu entwerfen, dass sie direkt die *Aggregates* des Domänenmodells widerspiegeln. Das zwingt jeden Client in eine *Conformist*-Beziehung, weil die Ressource jedes Mal angepasst wird, wenn sich das Modell ändert. Aus diesem Grund sollte man eine solche Konstruktion möglichst vermeiden. Stattdessen sollten die Ressourcen synthetisch nach den Anwendungsfällen des Client entworfen werden. Mit »synthetisch« meine ich, dass die angebotenen Ressourcen für den Client so ausgestaltet sind, wie er sie braucht, und nicht so, wie das tatsächliche Domänenmodell des *Aggregate* aussieht. Manchmal wird das Modell zufälligerweise genauso aussehen, wie es der Client benötigt. Der Entwurf der REST API sollte auf jeden Fall von den Bedürfnissen des Client getrieben werden und nicht von dem *Aggregate* des Domänenmodells. Aber der Entwurf der Ressourcen wird von dem getrieben, was der Client braucht, und nicht von der gerade aktuellen Zusammensetzung des Modells.

Messaging



Wenn man über asynchrones Messaging integriert, kann viel dadurch gewonnen werden, dass ein *Client-Bounded-Context* die *Domain Events* (dt.: *Domänenereignisse, fachliche Ereignisse*) abonniert, die von Ihrem eigenen oder einem anderen *Bounded Context* versendet werden. Messaging zu verwenden, ist einer der robustesten Formen der Integration, weil man die temporäre Kopplung loswird, die bei der blockierenden Integration mit RPC oder REST entsteht. Die Entwickler tendieren dazu, robustere Systeme zu bauen, weil diese die Latenzzeit des Nachrichtenaustausches bei der Programmierung einbeziehen und aufgrund des Programmiermodells niemals direkte Ergebnisse erwarten.

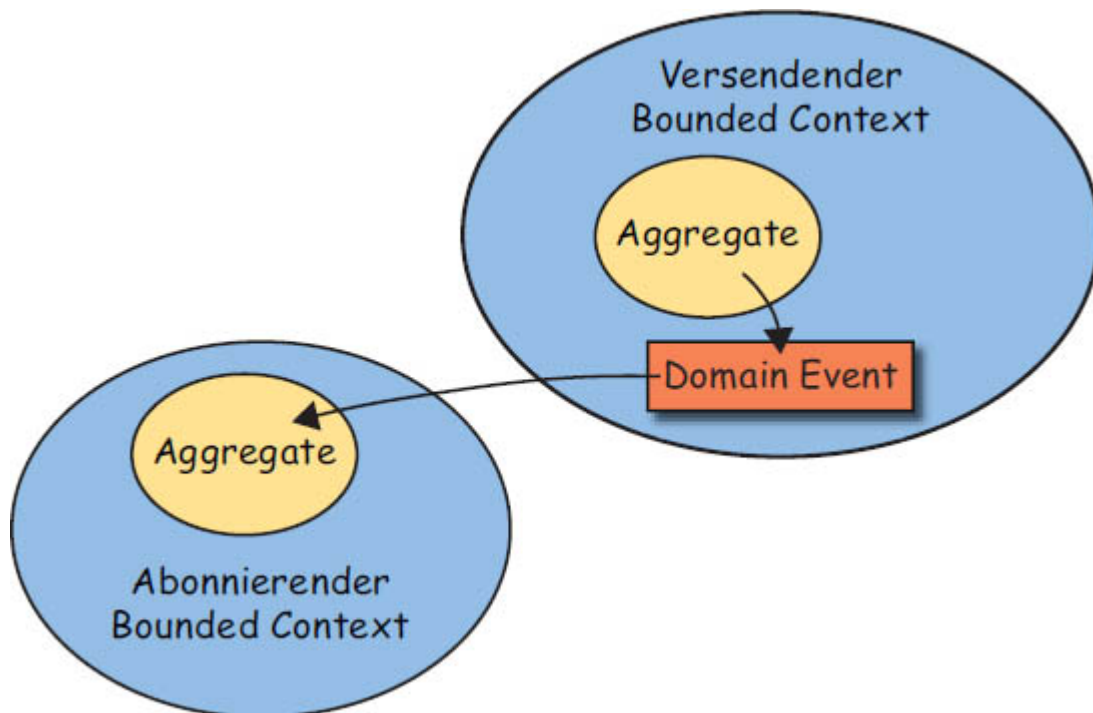
Asynchron mit REST arbeiten

Wenn man über REST-basiertes Polling eine sequenziell wachsende Menge von Ressourcen abfragt, kann man asynchrones Messaging simulieren. In einem Hintergrundprozess fragt ein Client regelmäßig eine Atom-Feed-Ressource des Servers ab, der eine wachsende Anzahl von *Domain Events* anbietet. Asynchrone Aufrufe zwischen Client und Server lassen sich auf diese Weise stabil umsetzen, während der Server ständig neue Events produziert. Wenn der Server aus irgendeinem Grund ausfällt, werden die Clients einfach in ihren normalen Intervallen versuchen, ihn erneut zu erreichen, bis die Feed-Ressource wieder zur Verfügung steht.

Dieser Ansatz wird im Detail in *Implementing Domain-Driven Design* [Vernon 2013] vorgestellt.

Massenkarambolagen bei der Integration vermeiden

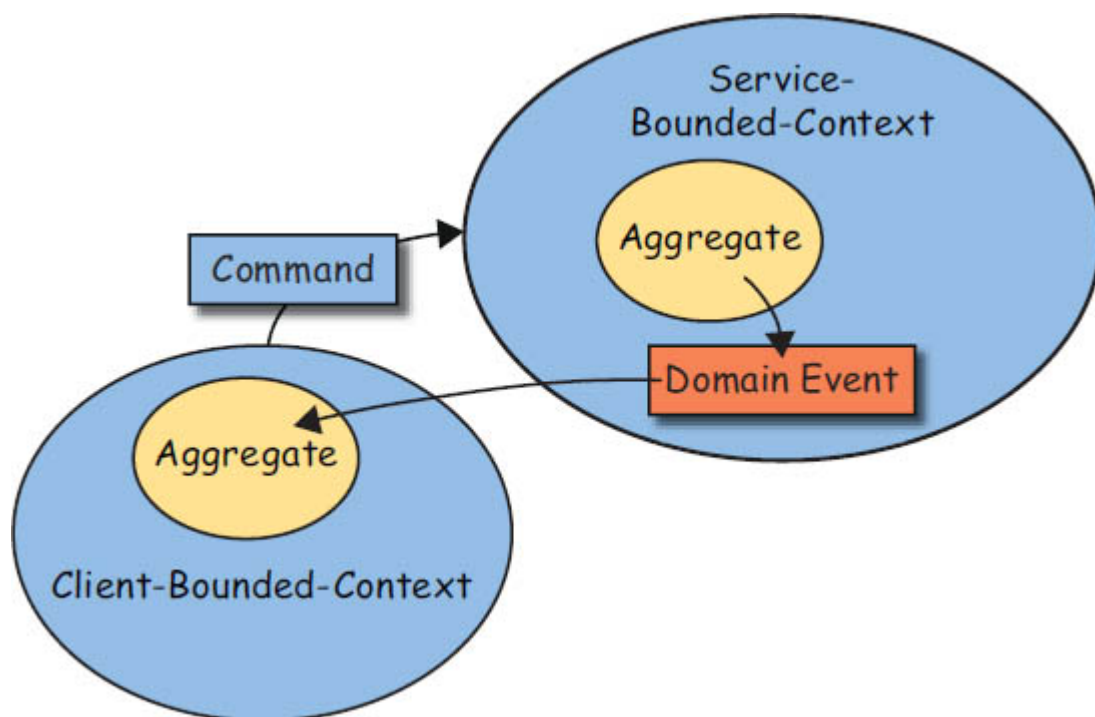
Wenn ein *Client-Bounded-Context* (C1) mit einem *Server-Bounded-Context* (S1) integriert wird, sollte C1 keine synchronen, blockierenden Anfragen an S1 als direkte Reaktion auf eine an ihn selbst gestellte Anfrage schicken. Das heißt, wenn ein anderer Client (C0) eine blockierende Anfrage an C1 stellt, dann erlauben Sie C1 nicht, eine blockierende Anfrage an S1 zu schicken. Sonst haben Sie ein hohes Risiko, eine Massenkarambolage von C0, C1 und S1 zu verursachen. Das kann durch Verwenden von asynchronem Messaging verhindert werden.



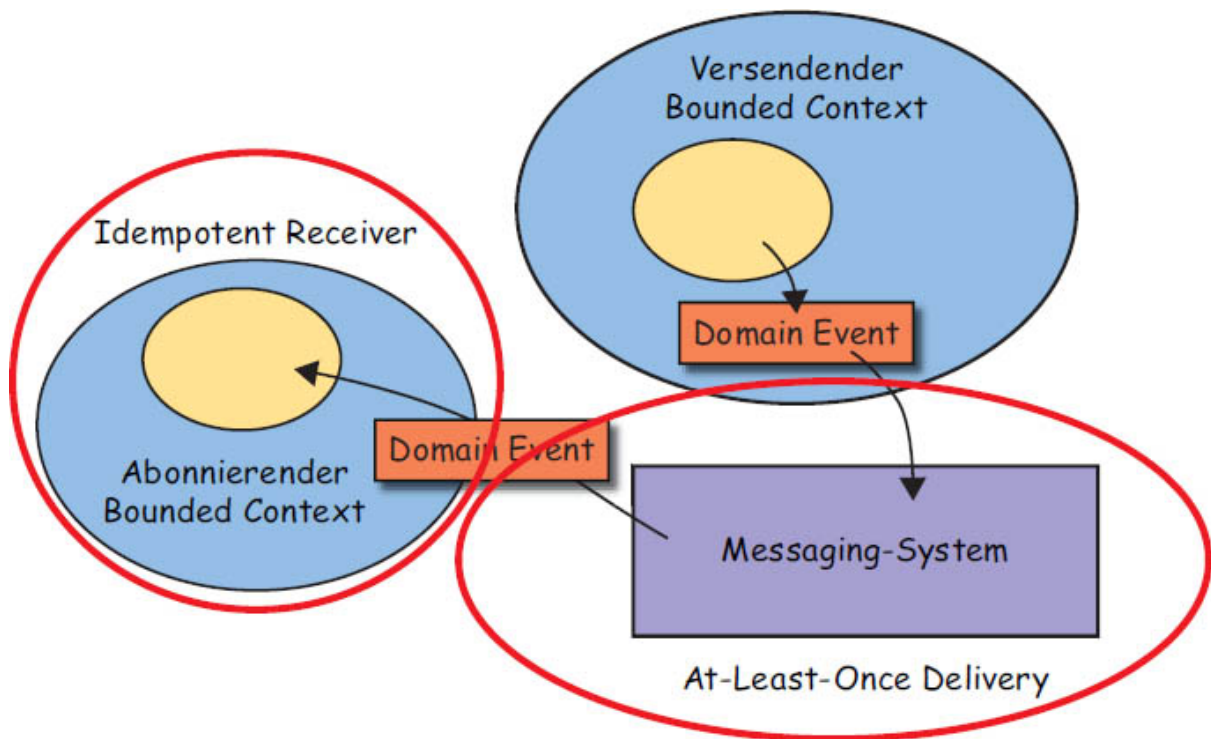
Typischerweise veröffentlicht ein *Aggregate* in einem *Bounded Context* ein *Domain Event*, das dann von einer beliebigen Anzahl von Interessenten verarbeitet werden kann. Wenn ein *Bounded Context*, der sich für diesen *Domain Event* registriert hat, das entsprechende *Domain Event* empfängt, dann wird abhängig vom Typ und Wert des *Domain Event* eine bestimmte Aktion ausgeführt. Diese Aktion im lauschenden *Bounded Context* wird normalerweise ein neues *Aggregate* erzeugen oder ein bestehendes *Aggregate* verändern.

Ist ein Bounded Context, der auf Domain Events lauscht, ein Conformist?

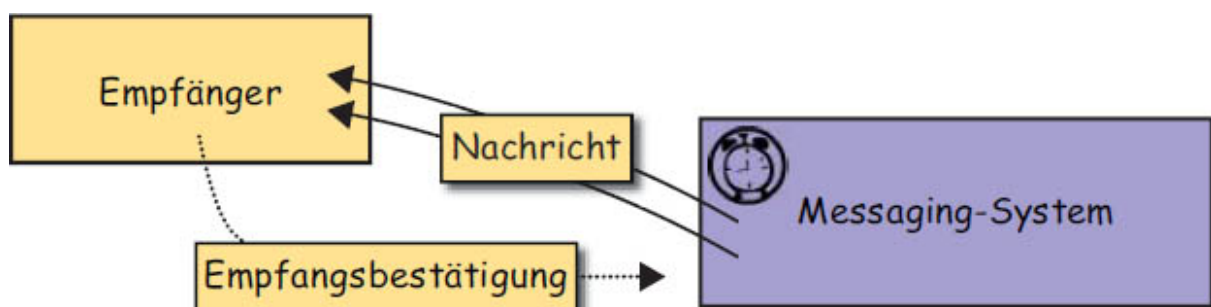
Sie fragen sich vielleicht, wie *Domain Events* eines *Bounded Context* von einem anderen verwendet werden können, ohne dass der zweite *Bounded Context* in eine *Conformist*-Beziehung gezwungen wird. Wie in *Implementing Domain-Driven Design* [Vernon 2013] und dort insbesondere in Kapitel 13, »Integrating Bounded Contexts«, empfohlen, sollten die Konsumenten nicht die Typen (d. h. Klassen) der Events des Event-Senders verwenden. Stattdessen sollten sie nur vom Schema der Events abhängen, also deren *Published Language*. Wenn die Events in JSON oder vielleicht einem platzsparenden Objektformat veröffentlicht werden, dann soll der Konsument die Events parsen, um an ihre Datenattribute heranzukommen.



Natürlich gehen diese Überlegungen davon aus, dass ein *Client-Bounded-Context* immer etwas mit den an ihn gesendeten Events aus einem anderen *Bounded Context* anfangen kann. Es gibt aber auch andere Fälle: Ein *Client-Bounded-Context* könnte z. B. gezwungen sein, proaktiv eine *Command Message* an einen *Service-Bounded-Context* zu verschicken, um eine bestimmte Aktion auszulösen. In solchen Fällen wird der *Client-Bounded-Context* trotzdem jedes Resultat als veröffentlichtes *Domain Event* empfangen.

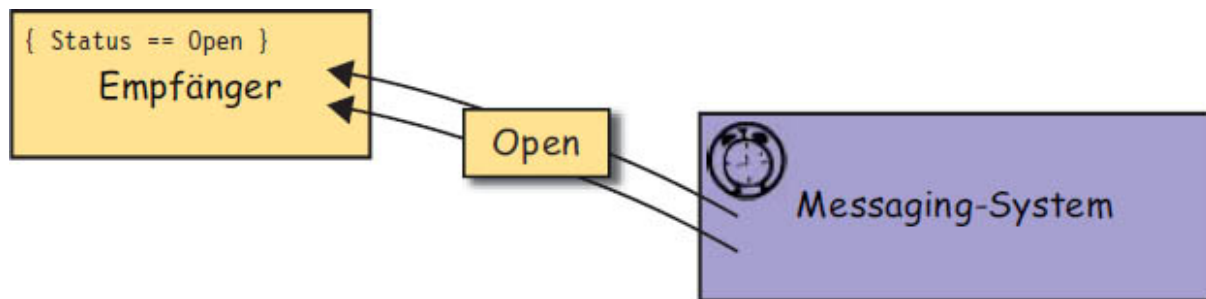


In allen Fällen, in denen Messaging für die Integration verwendet wird, hängt die Qualität der Gesamtlösung stark von der Qualität des gewählten Messaging-Verfahrens ab. Das Messaging-System sollte *At-Least-Once Delivery* [Vernon 2015] (in Deutsch etwa *Mindestens-einmal-Auslieferung*) unterstützen, um sicherzustellen, dass alle Nachrichten (zumindest irgendwann) empfangen werden. Das bedeutet auch, dass der *Client-Bounded-Context* als *Idempotent Receiver* [Vernon 2015] (dt.: *idempotenter Empfänger*) implementiert werden muss.



At-Least-Once Delivery [Vernon 2015] ist ein Messaging-Muster, bei dem das Messaging-System die Nachrichten regelmäßig immer wieder ausliefert. Diese wiederholte Auslieferung wird in folgenden Fällen angestoßen: Nachrichtenverlust, langsame Reaktion, ausgeschaltete oder ausgefallene Empfänger oder Empfänger, die den Empfang nicht bestätigen. Durch dieses Design des Messaging-Systems ist es möglich, dass die Nachricht mehrfach zugestellt wird, auch wenn der Sender sie nur einmal abgesendet hat. Das muss

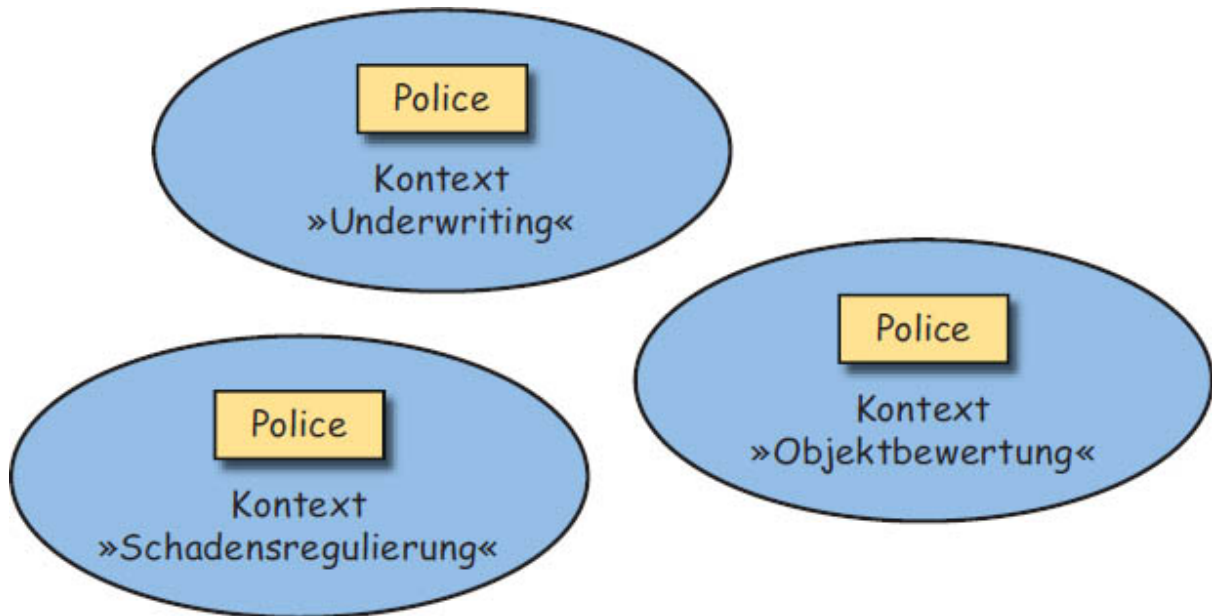
kein Problem sein, wenn der Empfänger so entworfen ist, dass er damit umgehen kann.



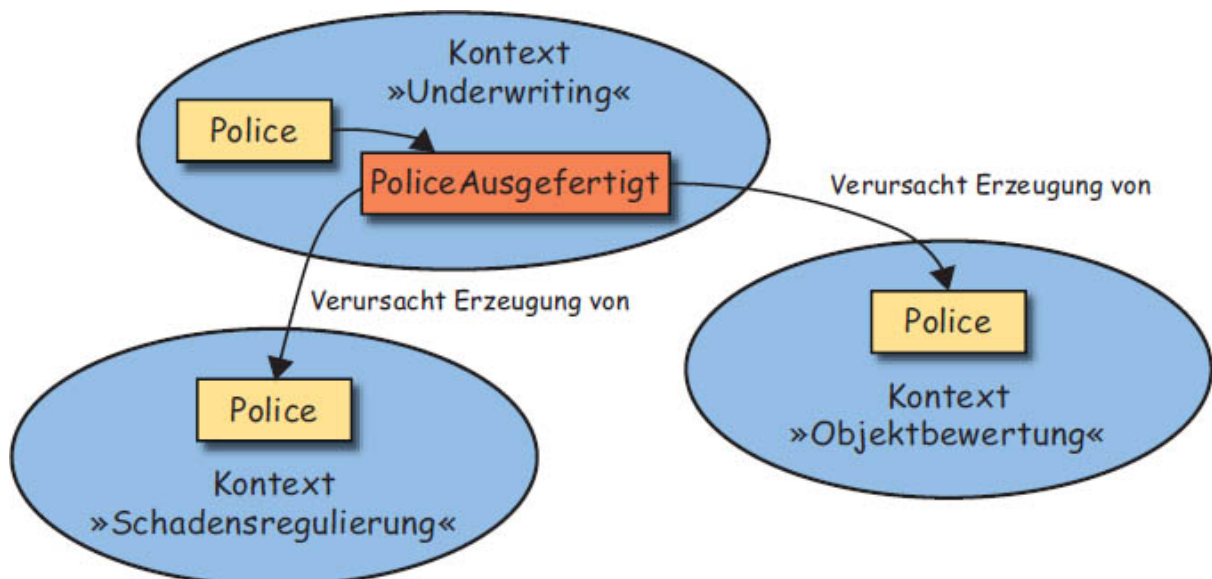
Immer wenn eine Nachricht mehr als einmal ausgeliefert werden könnte, sollte der Empfänger so konzipiert sein, dass er mit dieser Situation richtig umgehen kann. Das Muster *Idempotent Receiver* [Vernon 2015] beschreibt, wie der Empfänger einer Anfrage eine Operation so ausführt, dass er auch bei mehrfachem Ausführen derselben Operation das gleiche Ergebnis erzeugt. Dadurch wird sichergestellt, dass der Empfänger auch beim mehrfachen Empfang der gleichen Nachricht zuverlässig arbeitet. Dies kann bedeuten, dass der Empfänger einen der folgenden Mechanismen der Deduplizierung verwendet: Er ignoriert die wiederholte Nachricht oder er führt die Operation mit den exakt gleichen Ergebnissen wie bei der vorherigen Nachricht noch einmal aus.

Aufgrund der Tatsache, dass Messaging-Systeme immer asynchrone *Request-Response*-Kommunikation [Vernon 2015] einführen, ist eine gewisse Latenzzeit sowohl üblich als auch zu erwarten. Anfragen an einen Server sollten (fast) nie blockieren, solange die Dienstleistung noch nicht erfüllt ist. Entwirft man unter Berücksichtigung von Messaging, so plant man immer eine gewisse Latenzzeit ein. Dadurch wird Ihre gesamte Lösung von Anfang an viel robuster.

Context Mapping am Beispiel

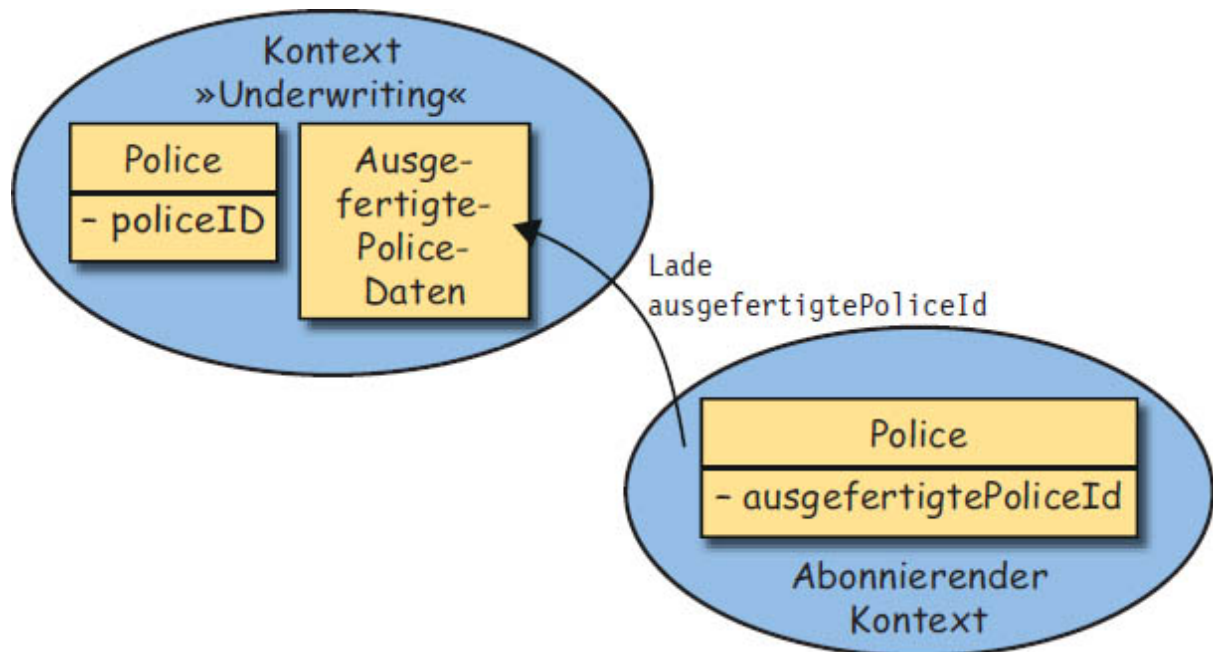


Wenn wir das Beispiel aus Kapitel 2, »Strategisches Design mit Bounded Contexts und der Ubiquitous Language«, betrachten, stellt sich die Frage, wo der zentrale Versicherungspolice-Typ `Police` hingehört. Denken Sie daran, dass es drei verschiedene `Police`-Typen in drei verschiedenen *Bounded Contexts* gibt. Also, wo ist der richtige Ort für die »echte Police« innerhalb des Versicherungsunternehmens? Es ist möglich, dass sie in die Underwriting-Abteilung gehört, da sie dort entsteht. Nehmen wir für dieses Beispiel daher an, sie gehöre zur Underwriting-Abteilung. Wie erfahren dann die anderen *Bounded Contexts* von ihrer Existenz?



Wenn eine Komponente des Typs `Police` im Underwriting-Kontext ausgefertigt wird, kann sie ein *Domain Event* mit dem Namen `PoliceAusgefertigt` veröffentlichen. Durch ein Messaging-System bereitgestellt, kann jeder andere

Bounded Context auf das *Domain Event* reagieren; wobei das Erzeugen einer entsprechenden *Police*-Komponente in einem *Bounded Context*, der das Event abonniert hat, eine mögliche Reaktion sein könnte.



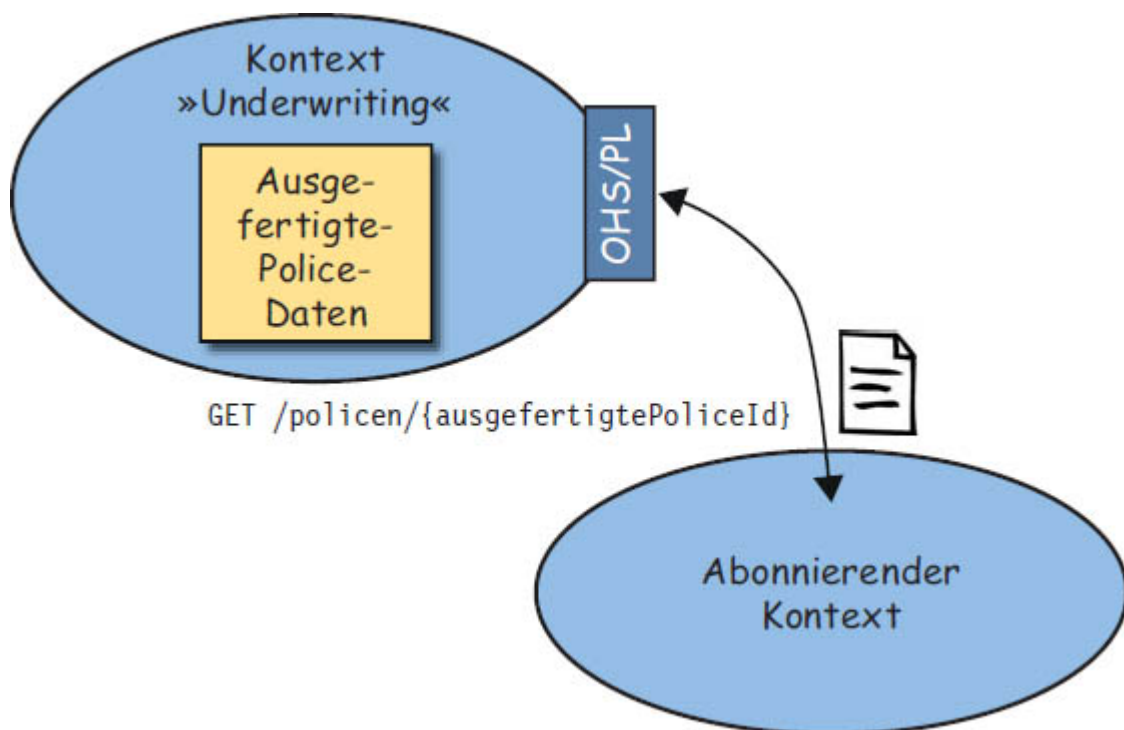
Das *Domain Event* *PoliceAusgefertigt* enthält die Identität der zentralen *Police*. Hier ist es *policeId*. Alle Komponenten, die in einem *Client-Bounded-Context* erstellt wurden, behalten diese Identität, um die *Police* zum ursprünglichen *Underwriting-Kontext* zurückverfolgen zu können. In diesem Beispiel wird die Identität als *ausgefertigtePoliceId* gespeichert. Wenn mehr *Police*-Daten erforderlich sind, als das *Domain Event* *PoliceAusgefertigt* bereitstellt, kann der abonnierende *Bounded Context* immer wieder den *Underwriting-Kontext* nach weiteren Informationen abfragen. Hier verwendet der abonnierende *Bounded Context* die *ausgefertigtePoliceId*, um eine Anfrage im *Underwriting-Kontext* durchzuführen.

Abwägung zwischen Anreichern und Nachladen

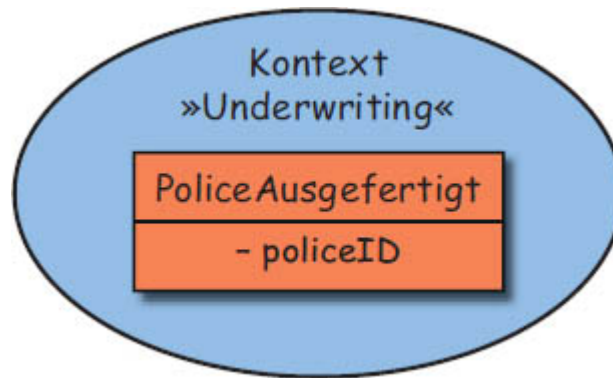
Manchmal ist es ein Vorteil, *Domain Events* vermehrt mit Daten anzureichern, um die Bedürfnisse aller Konsumenten zu befriedigen. Manchmal ist es von Vorteil, die *Domain Events* schmal zu halten, und den Konsumenten zu erlauben, Anfragen zu stellen und im Bedarfsfall Daten nachzuladen. Die erste Alternative, Anreichern, bietet größere Autonomie für die abhängigen Konsumenten. Wenn Autonomie die treibende Anforderung ist, sollten Sie das Anreichern in Betracht ziehen.

Andererseits ist es schwierig, jedes kleine Stück Daten vorherzusagen, das irgendein Konsument irgendwann einmal von einem bestimmten *Domain Event* benötigen wird. Wahrscheinlich ist das Anreichern zu umfangreich, wenn Sie alle Daten anbieten. In puncto Sicherheit kann es zum Beispiel eine schlechte Entscheidung sein, *Domain Events* im großen Stil anzureichern. In diesem Fall kann es besser sein, schmale *Domain Events* und ein reichhaltiges Query-Modell mit Sicherheitsmechanismen für Konsumenten zu entwerfen.

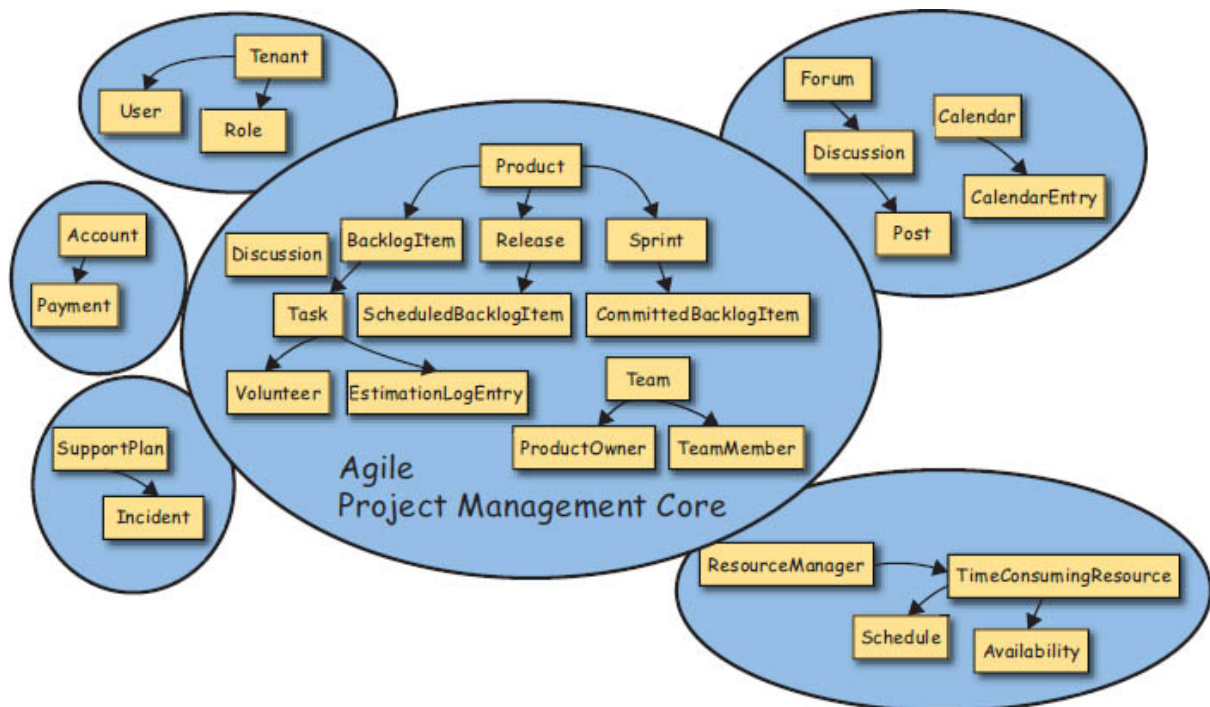
Manchmal werden die Gegebenheiten nach einer Mischung aus beiden Ansätzen verlangen.



Wie könnte das Nachladen vom Underwriting-Kontext funktionieren? Man könnte einen RESTful *Open Host Service* samt *Published Language* entwerfen. Ein einfaches HTTP-GET mit der ausgefertigtePoliceId würde dann die AusgefertigtePoliceDaten liefern.



Sie fragen sich wahrscheinlich, welche Daten das *Domain Event* `PoliceAusgefertigt` im Detail enthält. In Kapitel 6, »Taktisches Design mit Domain Events«, erfahren Sie die Details des Designs von *Domain Events*.



Sind Sie neugierig, was aus dem Beispielkontext *Agile Project Management Core* geworden ist? Dadurch dass wir in die Domäne Versicherungswirtschaft abgeschweift sind, konnten Sie DDD anhand mehrerer Beispiele betrachten. Das wird Ihnen helfen, DDD noch besser zu begreifen. Keine Sorge, im nächsten Kapitel werden wir zum Kontext *Agile Project Management Core* zurückkehren.

Zusammenfassung

In diesem Kapitel haben Sie Folgendes kennengelernt:

- Wie die verschiedenen Arten von *Context-Mapping*-Beziehungen, wie *Partnership*, *Customer-Supplier* und *Anticorruption Layer*, aussehen.
- Wie man *Context-Mapping*-Integration mit RPC, RESTful HTTP und Messaging verwendet.
- Wie *Domain Events* mit Messaging funktionieren.
- Eine Grundlage, auf der Sie Ihre *Context-Mapping*-Erfahrung ausbauen können.

Schauen Sie für eine tiefer gehende Betrachtung von *Context Maps* in Kapitel 3 von *Implementing Domain-Driven Design* [Vernon 2013] nach.

Index

A

Abstraktionen

- richtig wählen 90

Aggregate 73

- aktualisieren 82

- Daumenregeln 78

- im Event Storming 117

- kleine und große 80

- modellieren 85

- referenzieren anderer über ihre Identität 81

- richtig dimensionieren 92

- Root 75

Aktorenmodell 41, 76

Akzeptanztests 14, 37, 131

Altsysteme 45

Anemic Domain Model 85

- öffentliche Setter 89

Anticorruption Layer 54

- als Schutz gegen einen Big Ball of Mud 58

Anwendungsfälle 40

Application Service 40, 76

Architektur 40

At-Least-Once Delivery 66

Aufwand schätzen 126

B

BDD *siehe Behaviour-Driven Development*

Behaviour-Driven Development (BDD) 37

Big Ball of Mud 16

Probleme 57

Bounded Context 7, 11

Abbildung auf Subdomains 47

beim Event Storming finden 119

enthaltene Elemente 40

Integration von 50

Name 20

über Messaging integrieren 63

Zuordnung zu einem Team 14

C

Causal Consistency 97

Cloud Computing 41

Command

als Ursache für ein Domain Event 101

im Event Storming 115

Command Query Responsibility Segregation (CQRS) 41

beim Einsatz von Event Sourcing 108

Conformist 54

Context Map 8, 12

Context Mapping 8, 50

am Beispiel 68

Arten 52

Integrationstechnologien 58

Continuous Integration 52

Core Domain 12–13, 44, 49

in der Wartung 39

mit Event Storming modellieren 111

CQRS *siehe Command Query Responsibility Segregation*

Customer-Supplier 53

D

Datenbank

Aggregate in einer Transaktion schreiben 76

Zuordnung zu einem Team 14

Deduplizierung 67

Design 3

Domain Events 8

als Ausdruck eines vergangenen Ereignisses 100

als Geschäftsprozess 114

als Statusrepräsentation eines Aggregate 104

entwerfen, implementieren und verwenden 99

reichhaltige versus schlanke 69

richtige Reihenfolge 104

über Messaging 63

Domain Experts 7, 25

am Lernprozess beteiligen 110

Verständnis von Quellcode 39

versus Product Owner 26

zusammenarbeiten mit 132

Domänenmodell

Dokumente 35

mehrere ineinander verschlungene 46

Modellierungsschuld im 125

platzieren von Fachlogik in 85

technologiefrei entwickeln 41

validieren 38

downstream 53

E

Entity 8, 74

Event Sourcing 41, 76, 106

Verhältnis zu Event Storming 118

Event Store 107

Event Storming 110

Prozess 116

zum schnellen Lernen 34

Event Stream 106

Eventual Consistency 37, 83

F

Fachgebiet 18

fachliche Invarianten

schützen von 79

funktionale Programmierung 86

G

Generic Subdomain 45

Given/When/Then-Ansatz 38

I

Idempotent Receiver 66

Impact Mapping 121

Infragestellen 28

J

JSON Schema 56

K

Kanban 122

Komplexität

fachliche versus technische 27

umgehen mit 45

L

Lösungsraum 12

M

Messaging

Integration über 63

Microservices 41

Modellierungsschuld 125

Module 47

Monolith 16

N

Nachrichtenverlust 66

No Estimates 121

O

Open Host Service 55

mit REST 61

mit RPC 60

Outsourcing 45

P

Partnership 52

Persistenz-Mechanismen 82

Polling 63

Ports-and-Adapters-Architektur 40

Probleme, die durch DDD vermieden werden können 4

Problemraum 12, 44

Protobuf 56

Published Language 56

Q

Quellcode-Repository 14, 39

R

Reactive 41

Remote Procedure Call (RPC) 59

Representational State Transfer (REST) 41

RESTful HTTP 61

REST *siehe Representational State Transfer*

Root Entity *siehe Aggregate Root*

RPC *siehe Remote Procedure Call*

S

Scrum 122

Master 26

Product Owner 26, 103

Separate Ways 56

Serviceorientierte Architektur (SOA) 41

Shared Kernel 53

Simple Object Access Protocol (SOAP) 59

Single Responsibility Principle (SRP) 81

SOA *siehe Serviceorientierte Architektur*

SOAP *siehe Simple Object Access Protocol*

Specification by Example 37, 121

Sprache

Deutsch oder Englisch? 18

mehrere in einem großen Modell 15

unterschiedliche 19

SRP *siehe Single Responsibility Principle*

Strategisches Design 7, 24

mit Context Mapping 49

Subdomains 44

Arten 44

Supporting Subdomain 45

SWOT-Analyse 123

Szenarios 34, 130

T

Taktisches Design 8

mit Aggregates 73

mit Domain Events 97

Testen

modellieren für Testbarkeit 94

Vorteile durch Verwenden von Bounded Contexts 24

Timebox

Modellieren mit 129

Transaktionen 40, 76

U

Ubiquitous Language 7, 12

Domain Events als Teil der 99

entwickeln 34

mit Event Storming entwickeln 111

Übersetzen von 51

von Scrum 28

UML 87, 110

Unit Tests 14, 94

Unternehmensstrategie 13

upstream 53

User Story Mapping 121

V

Value Object 8, 75

Vereinheitlichen 28

Versicherung

Domain Events 68

Fachgebiete 18

Police 18

Views 120

W

Wartung 39

Wissenserwerb 6, 122

in der Wartung 39

X

XML Schema 56