



N Hallo

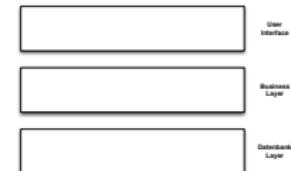
A Hi, na wie geht's?

N Uh, nicht so prima. Jetzt entwickle ich doch schon seit Jahren Software, und eigentlich hätte ich erwartet, dass es irgendwann mal besser klappt, aber irgendwie lande ich bei jedem Projekt wieder bei den gleichen Problemen. Erst fängt man ganz gut an, aber mit der Zeit wird alles immer mühsamer... Guck mal zum Beispiel unser aktuelles Projekt:

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Unsere aktuelle Architektur

Unsere aktuelle Architektur

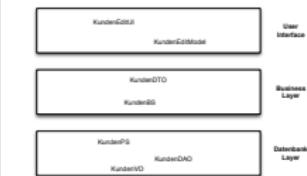


N So sieht's gerade bei uns aus. Ganz normal, oder? UI-Layer, Business-Layer und Datenbank-Layer. Aber irgendwie stellt sich heraus, dass unsere gesamte Logik total über die Schichten verschmiert ist. Guck mal hier:

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Unsere aktuelle Architektur

Unsere aktuelle Architektur

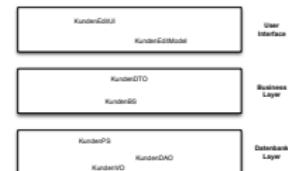


- N Schwierig, Tests zu schreiben / Technik auch / Auftraggeber versteht nur Bahnhof / wirft alle Abkürzungen durcheinander / alles ist aufgesplittert / Missverständnisse selbst bei Entwicklern / Jede Menge Implis und Managers
- A Ja, das kenn ich gut. Wir waren auch mal in so einer Abkürzungs-Hölle.
- N Waren? Das heißt, Ihr seid da rausgekommen? Und es gibt noch Hoffnung?
- A Naja, geschenkt bekommen haben wir das natürlich auch nicht. Das war schon ganz schön Arbeit.
- N Egal! Wie habt Ihr das gemacht, sag schon!

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Unsere aktuelle Architektur

Unsere aktuelle Architektur



- A Naja, ... wir machen jetzt Domain-Driven Design.
- N Hm, ich glaub davon hab ich schon gehört... DDD ... noch ne Abkürzung - nee, das hilft doch nicht weiter.
- A Na, wart's mal ab, ich erzähl Dir einfach mal ein bisschen davon, ok?
- N Na also gut... schlimmer kann's ja nicht mehr werden.

└ Domain-Driven Design

- ▶ Domänen-Experten und Entwickler gemeinsam
- ▶ Fokus der Entwicklung auf die Fachlichkeit
- ▶ Bausteine und Werkzeuge für gute Anwendungen

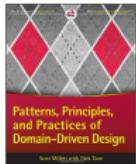
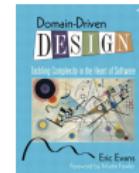
A stellt die Punkte vor

N Das klingt ja sehr interessant! Aber ist das jetzt nicht wieder so ein neuer Hype?

A Nein - Teile schon Jahrzehnte alt

A DAS Buch ist über 10 Jahre alt

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen



└ Literatur

A (Kommentar zu Vernon)

N OK. Aber bist Du wirklich sicher, dass das auch für uns passt?

A DDD ist besonders geeignet, wenn Fachlichkeit Kern der Anwendung / Abgegrenzte Domäne

N Das ist bei uns der Fall. Dann erzähl doch einfach mal weiter!

└ Baustein: Entität

- hat eine Identität (beschreibt das „wer“)
- hat einen Lebenszyklus
- Modellierung fokussiert darauf
- eindeutigen Identifikator festlegen

- A Identität gilt oft auch außerhalb der gerade betrachteten Software
- N Ja, das kenne ich, wir modellieren Kunden, die haben natürlich auch im echten Leben einen Namen und ein Geburtsdatum
- A Innerhalb des Lebenszyklus können sich Werte der Entität ändern
- N Ja, das macht immer am meisten Stress, wenn die Kunden umziehen oder heiraten. Nicht so einfach, damit umzugehen...
- A Genau, deswegen ist es wichtig, einen eindeutigen Identifikator festzulegen. Es könnten ja auch mal zwei Kunden denselben Namen haben.

└ Baustein: Entität

- hat eine Identität (beschreibt das „wer“)
- hat einen Lebenszyklus
- Modellierung fokussiert darauf
- eindeutigen Identifikator festlegen

- N Das habe ich auf der Datenbank ja auch immer, dass ich einen Schlüssel definieren muss.
- A Stimmt - die IDs von der Datenbank sind oft ein sinnvoller Kandidat für eine Identität im Objekt-Modell.
- N Okay, jetzt nenne ich also alle meine derzeitigen Datenbankobjekte Entitäten, oder? Dann verstehe ich noch nicht, was jetzt das Großartige und Neue sein soll...
- A Nur Geduld! Ich vermute nämlich, dass nicht alle Deiner derzeitigen Datenbankobjekte wirklich Entitäten sind. Es gibt ja auch noch Value Objects.

└ Baustein: Value Object

- Wert
- hat keine Identität (beschreibt „was“, nicht „wer“)
- fachlicher Wrapper um technische Datentypen
- bildet eine konzeptionelle Einheit
- kann oft als Immutable implementiert werden
 - dann ist Sharing möglich

A Value Objects wendet man da an, wo man Werte repräsentieren will. Und wo die Identität keine Rolle spielt, also wo ich ein Value Objekt durch ein anderes identisches ersetzen kann.

N Also Du meinst Strings und ints und sowas?

A Im Prinzip schon. Aber die haben keine fachliche Semantik: Was bedeutet dieser Double, welche Werte sind hier gültig?

N Oh ja, das kenne ich. Wir haben eine Methode mit 10 Parametern, alles Doubles. Der häufigste Bug an der Stelle ist, dass irgendjemand mal wieder die Werte in der falschen Reihenfolge übergeben hat...

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Baustein: Value Object

- Wert
- hat keine Identität (beschreibt „was“, nicht „wer“)
- fachlicher Wrapper um technische Datentypen
- bildet eine konzeptionelle Einheit
- kann oft als immutable implementiert werden
 - dann ist Sharing möglich

- A Und deswegen ist es sinnvoll, fachliche Wrapper-Klassen zu erstellen, die die Semantik ausdrücken. Simple Beispiele sind Geldbetrag oder Prozentsatz. Interessant dabei ist, dass man Value Objects oft als immutable implementieren kann, d. h. dass man das Objekt nach der Erzeugung nicht mehr verändern kann.
- N Hm, aber was ist, wenn sich jetzt was ändert, zum Beispiel die Gesamtsumme in meinem Warenkorb?

└ Baustein: Value Object

- Wert
- hat keine Identität (beschreibt „was“, nicht „wer“)
- fachlicher Wrapper um technische Datentypen
- bildet eine konzeptionelle Einheit
- kann oft als Immutable implementiert werden
 - dann ist Sharing möglich

A Da kann ich einfach einen neuen Geldbetrag mit der neuen Gesamtsumme bauen und den alten dadurch ersetzen. Und die Implementierung meines Value Objects wird dadurch viel einfacher. Man kann sogar häufig vorkommende Value Objects wiederverwenden - das ist ungefährlich, denn sie können ja nicht verändert werden.

N Ah, klingt interessant! Wir haben ganz oft einen Preis von 99 Cent, da könnte ich dann immer dasselbe Value Object verwenden, oder?

A Ja, zum Beispiel.

N Mensch, danke für die vielen Infos! Jetzt bin ich mal gespannt, wie wir das Ganze bei uns im Team umsetzen können. Tschüs, ich muss los!

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

Einige Wochen später...



└ Einige Wochen später...

- A Hallo, schön Dich wiederzusehen! Wie läuft's denn so mit Domain-Driven Design?
- N Oh, sehr gut! Wir haben inzwischen einige unserer Probleme in den Griff bekommen und auch coole neue Sachen entdeckt.
- A Das ist ja schön zu hören! Magst Du mir was davon erzählen?

└ Neues über Value Objects

- ▶ aus Entitäten auslagern
- ▶ können auch komplexer aufgebaut sein
- ▶ entwickeln sich zu „Code-Magneten“

N wir hatten ziemlich viele riesige Entitäten, z. B. hat unser Kunde alle Angaben über ihn direkt als Attribute gehabt: Alle Namens- und Adressbestandteile, Telefonnummer, Kreditkartennummer und was noch alles. Davon haben wir den Großteil herausgezogen und in spezialisierten Value Objects untergebracht. Value Objects müssen nämlich nicht immer nur simple Wrapper sein, sie können auch aus mehreren Werten zusammengesetzt sein. Der Kunde kennt jetzt nur noch diese Value Objects, zum Beispiel seine Adresse.

└ Neues über Value Objects

- ▶ aus Entitäten auslagern
- ▶ können auch komplexer aufgebaut sein
- ▶ entwickeln sich zu „Code-Magneten“

A Eine Adresse ist ein gutes Beispiel für ein komplexes Value Object in der Domäne „Online-Store“. Denk nur dran, dass es in anderen Domänen ganz anders sein kann. In einer Grundstücksverwaltung oder in einer Briefzustellungssoftware wird man Adressen vermutlich eher als Entitäten modellieren.

N Hm...

A Aber ich hab Dich unterbrochen. Was hast Du denn sonst noch so rausgefunden?

└ Neues über Value Objects

- ▶ aus Entitäten ausgliern
- ▶ können auch komplexer aufgebaut sein
- ▶ entwickeln sich zu „Code-Magneten“

N Das coolste ist: Die Value Objects ziehen den Code förmlich an! Alles, was mit einem Value Object zu tun hat, implementieren wir dort als Methode: z. B. Validierung oder das Abfragen von bestimmten Eigenschaften. Früher war dieser Code im ganzen System verstreut, und meistens hat man ihn nicht wiedergefunden, wenn man ihn gebraucht hat, und hat alles nochmal implementiert, und nochmal... Jetzt gibt es einen klaren Platz für diesen Code, das ist super!

A Mir scheint, Ihr seid auf einem richtig guten Weg!

N Aber es knirscht auch noch ganz schön an vielen Stellen.

└ Neues über Value Objects

- ▶ aus Entitäten ausgliern
- ▶ können auch komplexer aufgebaut sein
- ▶ entwickeln sich zu „Code-Magneten“

A Wo denn zum Beispiel?

N Naja, wir Entwickler kommen noch nicht so gut klar mit den Leuten vom Fachbereich. Sie sprechen eine ganz andere Sprache als wir, zum Beispiel sagen sie „Artikel“, und das heißt bei uns im Code „Bestellposition“. Einer von uns hat das gut im Griff und spielt immer den Übersetzer. Aber wenn der nicht da ist, läuft echt gar nix. Manchmal benutzen aber auch alle denselben Begriff, meinen nur etwas komplett Unterschiedliches. Dass hinterher nix zusammenpasst, ist da ja fast schon garantiert.

A Dass eine gemeinsame Sprache sehr wichtig ist, das ist auch ein Schwerpunkt von DDD.

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Die allgegenwärtige und gemeinsame Sprache

- A Du hast ja selbst schon festgestellt, wie wichtig und schwierig die Sprache ist. Alle müssen dieselbe Sprache sprechen - vom Auftraggeber über den Fachbereich bis zum Entwickler. Idealerweise solltest Du dem Auftraggeber den Namen irgendeiner Klasse nennen können, und er kann sich etwas darunter vorstellen.
- N Hm, also das klappt bei uns noch nicht so gut, bei unserem KundeVO kann sich unser Auftraggeber echt nichts vorstellen, komisch eigentlich.
- A Naja, ein VO gibt es in seiner Fachlichkeit nun mal nicht. Hast Du vielleicht einen besseren Namen für die Klasse?

- Ubiquitous Language
- Fachbegriffe überall verwenden, auch im Code!
- Glossar zur Begriffsklärung
- muss reichhaltig genug sein für sämtliche Kommunikation
- beschreibt nicht nur die Einheiten im System, sondern auch Aufgaben und Funktionalitäten
- muss widerspruchsfrei und eindeutig sein
- Bounded Context, um grosse Domänen zu strukturieren

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Die allgegenwärtige und gemeinsame Sprache

- Ubiquitous Language
- Fachbegriffe überall verwenden, auch im Code!
- Glossar zur Begriffsklärung
- muss reichhaltig genug sein für sämtliche Kommunikation
- beschreibt nicht nur die Einheiten im System, sondern auch Aufgaben und Funktionalitäten
- muss widerspruchsfrei und eindeutig sein
- Bounded Context, um grosse Domänen zu strukturieren

N Naja, eigentlich ist es ja ein Kunde - also wie wäre es damit? Ging das? Einfach nur „Kunde“?

A Klar, das wäre ziemlich gut.

N Aber das ist ja sicher noch nicht alles.

A Das Wichtigste hierbei ist sicherlich, konsequent zu sein. Die gemeinsame Sprache muss alles umfassen und muss in allen Gesprächen benutzt werden - egal ob da nur Entwickler oder nur Fachler oder beide miteinander reden. Ein Glossar hilft ungemein, um die Begriffe zu schärfen, damit es möglichst wenige Missverständnisse gibt. Und natürlich darf jeder Begriff nur für genau eins verwendet werden - wenn's mehrdeutig wird, muss man gemeinsam überlegen, wie man das auflösen kann.

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Die allgegenwärtige und gemeinsame Sprache

A Apropos mehrdeutig: In umfangreichen Domänen kann es schwierig werden, eine einzige gemeinsame Sprache und ein einziges Domänenmodell zu finden. Z.B. in der Versicherungsdomäne mit Subdomänen wie Sach-, Lebens- oder Nicht-Lebensversicherungen kann sich die Frage stellen, ob Begriffe wie Kunde, Vertrag, Versicherungsgegenstand etc. einheitlich verstanden werden oder nicht. Um mit verschiedenen Sprachen und Modellen in grossen Domänen auch organisatorisch umgehen zu können, gibt es bei DDD das Konzept der Bounded Contexts, mit denen quasi Töpfe für verschiedene Bereiche einer Domäne definiert werden können.

- Ubiquitous Language
- Fachbegriffe überall verwenden, auch im Code!
- Glossar zur Begriffsklärung
- muss reichhaltig genug sein für sämtliche Kommunikation
- beschreibt nicht nur die Einheiten im System, sondern auch Aufgaben und Funktionalitäten
- muss widerspruchsfrei und eindeutig sein
- Bounded Context, um grosse Domänen zu strukturieren

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Die allgegenwärtige und gemeinsame Sprache

N Und wenn ich mal merke, dass ich einen ungeschickten Begriff eingeführt habe, und den ändern möchte? Dann muss ich vermutlich auch in meinem Code alles umbenennen, oder?

A Ja genau. Ich sehe, so langsam findest Du Zugang zu dem Thema :-) Hatte ich eigentlich schonmal den Begriff Aggregate erwähnt?

N Nicht, dass ich wüsste. Erzähl doch mal!

- Ubiquitous Language
- Fachbegriffe überall verwenden, auch im Code!
- Glossar zur Begriffsklärung
- muss reichhaltig genug sein für sämtliche Kommunikation
- beschreibt nicht nur die Einheiten im System, sondern auch Aufgaben und Funktionalitäten
- muss widerspruchsfrei und eindeutig sein
- Bounded Context, um grosse Domänen zu strukturieren

└ Baustein: Aggregate

- besteht aus Value Objects und Entities
- Aggregate Root als Einstiegspunkt
- gemeinsame Invarianten und Konsistenzbedingungen
- Zugriff auf Elemente über die Root

A Ein Aggregate ist flapsig gesagt eine höherwertige Entity. Es besteht aus einem in sich abgeschlossenen Teilgraphen des Domänenmodells. Ein Aggregate hat immer einen Einstiegspunkt: Die Root-Entity.
(NOTIZ: Optional Skizze auf Flipchart.)

N Uh, das finde ich noch recht abstrakt. Lass mich mal probieren, ob ich das irgendwie anwenden kann...
Wenn bei uns die Kunden was bestellen, dann haben wir bisher die Bestellung als Ganzes und gleichzeitig die einzelnen Bestellpositionen. Bisher sind das jeweils einzelne Entitäten. Ahne ich das richtig, dass wir die Bestellung als Ganzes als Aggregate behandeln sollen.

└ Baustein: Aggregate

- besteht aus Value Objects und Entities
- Aggregate Root als Einstiegspunkt
- gemeinsame Invarianten und Konsistenzbedingungen
- Zugriff auf Elemente über die Root

- A Genau. Natürlich ist es erstmal richtig, dass auch Bestellpositionen als Entities existieren. So richtig lebensfähig und sinnvoll verwendbar sind sie aber nur im Kontext einer Bestellung. So richtig interessant wird es, wenn dann herauskommt, dass für das Aggregate als Ganzes bestimmte Bedingungen gelten, die auf Ebene der einzelnen Elemente davon gar nicht überprüfbar sind.
- N Meinst Du sowas wie, dass mindestens ein Artikel im Warenkorb sein muss oder nach dem Klick auf den Bestell-Knopf die Positionen nicht mehr hinzugefügt oder gelöscht werden können.

└ Baustein: Aggregate

- besteht aus Value Objects und Entities
- Aggregate Root als Einstiegspunkt
- gemeinsame Invarianten und Konsistenzbedingungen
- Zugriff auf Elemente über die Root

- A Ja, um solche Invarianten geht es. Die kann ich auf Ebene einzelner Bestellpositionen nicht gewährleisten, außer wenn ich ganz eklige zyklische Abhängigkeiten zur Bestellung aufbaue. Daher müssen die Manipulationen immer über die Root-Entity erfolgen. In Reinform gibt die Root-Entity nur Kopien der Elemente nach außen. Die können dann zum Lesen verwendet werden. Dadurch dass es Kopien sind, ist das Schreiben nicht sinnvoll und auch nicht möglich.
- N Eine Frage habe ich aber jetzt schon noch. Ich versuche jetzt ja immer, meine Logik schön auf Values und Entities bzw. Aggregates zu verteilen.

└ Baustein: Aggregate

- besteht aus Value Objects und Entities
- Aggregate Root als Einstiegspunkt
- gemeinsame Invarianten und Konsistenzbedingungen
- Zugriff auf Elemente über die Root

N Oft ist das auch ganz natürlich, wo eine Methode hingehört. Aber dann gibt es immer wieder Code, der einfach nirgends so richtig passen will. Der landet bei uns dann meistens in static-Methoden von irgendwelchen Entitäten.

A Ich ahne zwar schon, was Du meinen könntest, aber kannst Du mal ein Beispiel nennen?

N Mhhm.... Eine Sache war eine static Methode `authenticate()` auf der Klasse User. Die muss überprüfen, ob Username und Passwort da sind und auch ob sie richtig sind. Den ersten Teil hätte ich ja schon als Instanzmethode sehen können. Der Abgleich mit den hinterlegten Werten hat aber in einer Instanzenvariablen nichts zu suchen, oder siehst Du das anders?

└ Domain Services

- Hinweis auf fehlenden Service: Code in static Methoden
- zustandslos
- oft als Einstieg in die Domäne

- A Du scheinst auf Code gestoßen zu sein, der in einen Domain Service gehört.
- N Service? Oh nein - muss ich jetzt eine WSDL schreiben? Diese ganze XML-Hölle wollte ich hier eigentlich nicht haben!
- A Keine Sorge, hier geht es nicht um SOAP-Services oder um Remote-Aufrufe.
- N Dann bin ich ja beruhigt. Was ist das sonst?

└ Domain Services

- Hinweis auf fehlenden Service: Code in static Methoden
- zustandslos
- oft als Einstieg in die Domäne

- A Ein Domain Service ist ein Stück Logik, das im Normalfall zustandslos ist. Das Zustandslose hat Dich ja auch dazu gebracht, den Code als static Methode zu implementieren. Der Domain Service gibt solch einem herrenlosen Stück Code ein zu Hause. Es kann von außen angestoßen werden und triggert dann Aktionen in der Domain.
- N Das klingt interessant! Ich glaube, das werde ich gleich mal ausprobieren. Danke!
- A OK, viel Erfolg!



N Hallo Arif!

A Hallo, wie geht's?

N Sehr gut, wir haben Deine Domain Services mal ausprobiert und dabei viel Interessantes gelernt.

A Klingt gut, erzähl doch mal!

└ Domain Services Revisited

- ▶ zu viel Code in Services
 - prozedurale Programmierung
 - Blümchens Modell
- ▶ zu viel Code in Entitäten
 - Überfrachtung der Entitäten mit Verhalten
 - Verlust konzeptioneller Klarheit
 - Abhängigkeiten an der falschen Stelle

N Als Du mir letztes Mal die Services vorgestellt hast, dachte ich, das ist ja einfach: Jede Methode, bei der ich nicht weiß, wo sie hinkommt, packe ich einfach erstmal in einen Service. Aber das ist auch nicht die Lösung.

A Da sind wir einer Meinung.

N Dann lande ich nämlich bei Code, der nicht mehr viel mit Objektorientierung zu tun hat - alle Funktionalitäten sind ausgelagert in Services, und meine Objekte sind nur noch Datencontainer. Aber wenn ich zu wenig auslagere, sind irgendwann meine Objekte vollgestopft mit Methoden, dann sieht man auch nichts mehr. Das ist eine ziemliche Gratwanderung, auf die man sich da begibt.

└ Domain Services Revisited

- ▶ zu viel Code in Services
 - prozedurale Programmierung
 - Blaues Modell
- ▶ zu viel Code in Entitäten
 - Überfrachtung der Entitäten mit Verhalten
 - Verlust konzeptioneller Klarheit
 - Abhängigkeiten an der falschen Stelle

- A Das habt Ihr gut beobachtet. Daher hatte ich die Services auch nicht gleich am Anfang erwähnt. Sie sind eher das Mittel für den Notfall. Nur wenn Du keine Entität oder kein Value Object findest, wo die Logik hingehört, solltest Du über einen Service nachdenken.
- N Genau! Ich will meine Daten und mein Verhalten möglichst eng beieinander haben. Das haben wir inzwischen ganz gut hinbekommen. Aber bei einem anderen Design-Problem beißen wir uns noch die Zähne aus.
- A Worum geht's denn? Vielleicht kann ich ja helfen.

└ Repositories

- ▶ fachliche Schnittstelle zu Daten
- ▶ gaukeln In-Memory-Collection vor
- ▶ keine Infrastruktur-Abhängigkeiten in Domain Code

N Wir hatten bisher vor allem die Kern-Fachlichkeit im Blick. Mit Value Objects, Entities und Services haben wir das recht gut abbilden können. Jetzt wollten wir aber mal eine Datenbank anbinden. Ich habe da mal die Bücher gewälzt und bin auf Repositories gestoßen. Wir sind ganz naiv drangegangen und haben uns eine Schnittstelle gebaut, bei der wir einfach nur fachlich beschreiben, welche Daten wir lesen, löschen oder speichern wollen. Von der Ansteuerung ist das dann in etwa so, als ob es eine Collection wäre, die im Speicher liegt. Nur dass ich fachliche Zugriffsmethoden habe. Technische Klassen werden in der Schnittstelle nicht referenziert, also nichts aus Persistence-Frameworks oder ähnlichem.

└ Repositories

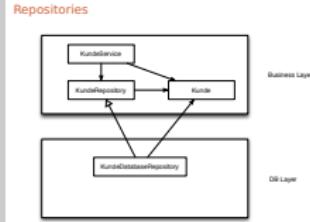
- ▶ fachliche Schnittstelle zu Daten
- ▶ gaukeln In-Memory-Collection vor
- ▶ keine Infrastruktur-Abhängigkeiten in Domain Code

A Das klingt doch schon mal sehr vernünftig. Verstehe ich das richtig, dass Ihr eine Trennung von Schnittstelle und Implementierung habt?

N Ja genau. Schau mal hier:

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

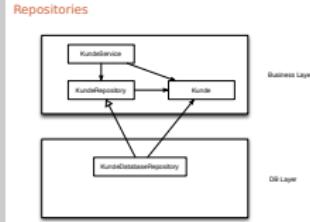
└ Repositories



N Wir haben die Schnittstelle, also in unserem Fall das KundenRepository. Irgendwo muss dann natürlich auch mal wirklich eine Datenbank angesprochen werden. Das passiert dann in der Klasse KundenDatabaseRepository. Dort werden dann Queries gebaut oder der OR-Mapper angeworfen. In unseren Domain-Services, wo wir die Repositories verwenden, sind wir dann aber absolut Datenbank-los unterwegs. Zuerst haben wir pro Entity-Klasse ein Repository gebaut.

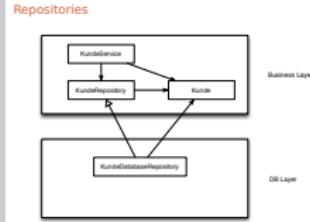
A Oh - das klingt nicht so clever!

└ Repositories



- N Dann haben wir aber bald gemerkt (und dummerweise erst danach im Buch gelesen), dass man das nur pro Aggregate macht. Das Repository geht dann von der Wurzel des Aggregate aus und lädt das dann in einem konsistenten Zustand.
- A Klingt, als ob Ihr das gut im Griff habt. Und wo ist jetzt Euer Problem, von dem Du geredet hast?
- N Wie strukturieren wir das am Besten? Wir haben hier unseren Domain Service. Der programmiert sauber gegen die Schnittstelle des Repositories. sobald ich aber dann doch die wirkliche Implementierung für die

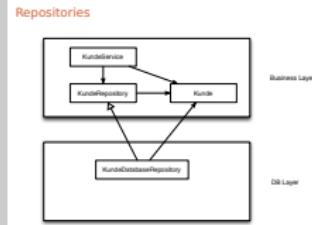
└ Repositories



- N DB verwenden will, habe ich doch die Abhängigkeit aus der Domain zur Datenbank, also zu technischen Dingen. Das will ich so doch nicht haben, oder?
- A Das ist wirklich nicht so ideal. Natürlich bist Du schonmal deutlich besser unterwegs als davor! Du hast alle Fachlichkeit in einer Schicht angesiedelt. Aber diese Abhängigkeit fühlt sich irgendwie falsch an.
- N Letzten Endes habe ich aber keine andere Option, wenn ich mich an die Schichtenarchitektur halten will. Alle Zugriffe müssen ja von oben nach unten gehen. Von unten nach oben ist verboten.

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

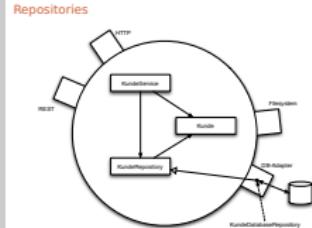
└ Repositories



A Stimmt, das klassische 3-Tier-Schichten-Modell mit UI - Business - Persistence führt zu diesem Phänomen. Es gibt aber auch noch eine andere Art, wie man das ganze betrachten kann. Die finde ich inzwischen einleuchtender. Halte Dich mal gut fest, beim ersten Betrachten kann es einem da schon mal kurz schwindlig werden.

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

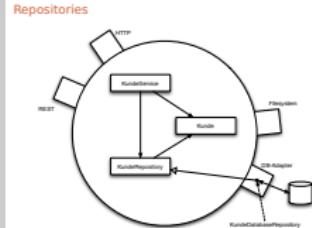
└ Repositories



- N Uuuups - wie muss ich das jetzt verstehen?
- A Wenn wir die Architektur so darstellen, dass stellen wir die Domäne ins Zentrum.
- N Das passt ja auch zu unserer ganzen Vorgehensweise bisher. Die Fachlichkeit ist das Kernstück der Software.
- A Statt von oben und unten reden wir eher von innen und außen. Innen ist also die Domäne. Außen ist der Rest der Welt, mit dem wir in Verbindung stehen. Wir können über verschiedene Kanäle Nachrichten nach außen loswerden oder auch von außen angestoßen werden. Wir definieren eine fachliche Schnittstelle innen. Die Anbindung nach außen passiert dann über einen Adapter. Die Implementierung des Repositories ist dann so ein Adapter.

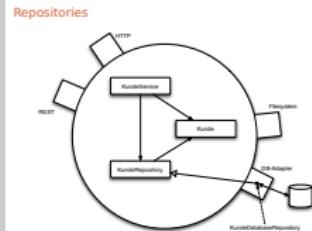
Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Repositories



- A Dort wird die Schnittstelle z.B. auf eine SQL-Datenbank adaptiert. Und wer kein SQL kann, nimmt halt eine NO-SQL-DB.
- N Und wo kriege ich einen solchen Adapter dann her? Auf Deinem Bild hier sieht es ja so aus, als ob es keine Zugriffe aus der Domäne auf die Außenwelt gibt.
- A Gut erkannt! Dort wo ich ein Repository brauche - also typischerweise in einem Domain Service, wünsche ich mir einfach ein Repository - also erwarte das z.B. als Konstruktor-Argument. Das Wiring mit der Datenbank-Implementierung erfolgt dann durch das Zusammenbauen der Anwendung. Dependency Injection ist das Stichwort.
- N Ach so - Du meinst Spring oder CDI!

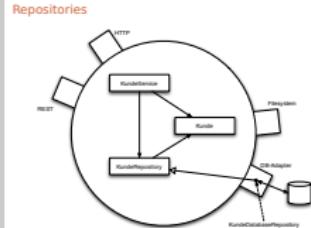
└ Repositories



- A Nicht unbedingt. Klar - ich kann mir das Zusammenbauen von einem Container oder Framework abnehmen lassen. Oft ist das auch eine gute Idee. In einfachen Fällen kann ich aber auch einfach in der main-Methoden die richtigen Implementierungen instanziieren und dann miteinander verheiraten.
- N Moment mal - ich glaub, jetzt sehe ich wo das hinführt. Das hilft mir auch beim Testen. Auch hier baue ich mir ja einen Teil der Anwendung zusammen. Da habe ich die Wahl, ob ich ein echtes also DB-gebundenes Repository verwende, oder einfach nur einen Mock oder Stub, der die Daten in einer Collection hält. Das ist wirklich elegant. Ich habe gesehen, hier gibt es noch andere Adapter. Wozu sind die da?

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

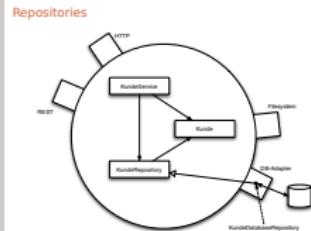
└ Repositories



A Bisher haben wir ja vor allem über Repositories geredet, also, wo die Daten herkommen oder hingehen. Die anderen Adapter beschreiben vor allem die Wege in die Domäne. Also, wie werden fachliche Prozesse getriggert. Wie sieht die Anbindung aus. Auch hier sind ja die unterschiedlichsten Wege denkbar: GUI, REST-/SOAP-Schnittstelle, ... Rein technisch sehen die entsprechenden Ansteuerungsstrecken ja oft extrem unterschiedlich aus. Letzten Endes ist die fachliche Schnittstelle, bei der sie landen ja immer die gleiche. Ob ich jetzt einen "BestellenKnopf drücke oder per Rest-Call das Signal dazu gebe, letzten Endes soll ja das gleiche passieren.

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Repositories



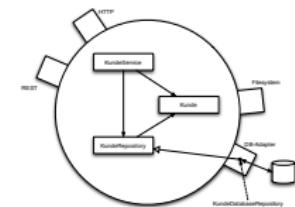
N Und auch das ist ja für das Testen wieder geschickt. Ich kann mir einen Testtreiber schreiben, der einfach auf der Domänenschicht aufsetzt. Das ist ja auch nur ein weiterer Adapter.

A Ganz genau. Wenn ich mein System so strukturiere, dann stelle ich die Domäne ins Zentrum. Die Interaktion mit außen geschieht über Ports und Adapters. Das zwingt mich dazu, dass ich mir über meine Schnittstelle sehr bewusst bin. Aber das ist ja immer eine gute Idee. In einem ersten Schritt setze ich dann vielleicht einen Rest-Adapter drauf. Wenn dann später die Anforderungen sich ändern und ich eine SOAP-Schnittstelle brauche, weil ein Kunde das nur so bei sich integrieren kann, dann brauche ich nur einen weiteren Adapter. Der ist dann nochmal zu testen.

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Repositories

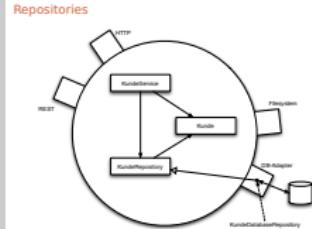
Repositories



- N Und hat diese Art zu strukturieren auch einen Namen. Schichten-Architektur kennt ja jeder...
- A Es gibt nicht einen einzigen Namen. Üblich sind "Ports and Adapters" oder "Hexagonal Architecture" (wobei das ein seltsamer Name ist, der nur wegen der Visualisierung des inneren Kreises als Sechseck gewählt wurde. Nicht ganz so üblich aber sehr sprechend ist Onion Architecture".

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

└ Repositories



- A Innendrin ändert sich aber erstmal gar nichts.
- N Nochmal zu meiner Frage mit den Repositories und der Strukturierung...
- A Stimmt, da waren wir ja gestartet. In der Domäne definierst Du Deine gewünschte Schnittstelle. Die Implementierung kann dann in separaten Packages oder sogar in einem eigenen Modul erfolgen. Natürlich muss die Implementierung ihre Schnittstelle kennen. In die andere Richtung ist aber keine Abhängigkeit notwendig.
- N Okay - das macht Sinn. Da muss ich jetzt mal ein paar Sachen umbauen gehen. Danke auf jeden Fall fürs Erklären.



- N Dass ist ja schön, dass wir uns mal wieder sehen!
- A Freut mich auch. Erzähl doch mal, wie es bei Euch jetzt inzwischen so läuft? Kommt Ihr gut voran?
- N Oh ja - wir sind sehr zufrieden. Natürlich klappt nicht alles perfekt. Aber gerade gestern hatte ich mal wieder so ein Aha-Erlebnis. Wir hatten beim Sprint-Review bemerkt, dass wir bei der Rabattberechnung einen Randfall nicht richtig umgesetzt hatten. Ich habe direkt danach kurz in den Code geschaut. Und mein Fachler saß noch da und hat auf einmal angefangen, mit mir im Code zu lesen. Der Hammer war, dass er den Fehler vor mir gefunden hat. Dadurch, dass er die Begriffe im Code kannte, haben ihn die ganzen eckigen und geschweiften Klammern anscheinend nicht gestört.



- A Das klingt ja echt, als ob alles in eine gute Richtung geht.
- N Das denke ich auch. Wir merken jetzt einfach, dass es sich ausgezahlt hat, in ein passendes Objekt-Modell zu investieren. Das steht jetzt einfach bombensicher.
- A Ähhmmm. Da wäre ich mir mal nicht zu sicher.
- N Wieso - alle Funktionalität und Logik ist dort, wo sie hingehört.
- A Es kann theoretisch schon sein, dass das jetzt so bleibt, aber das ist nicht so wahrscheinlich. Das kann verschiedene Gründe haben.
- N Und welche könnten das sein?

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

- A Meistens hängt es damit zusammen, dass man im Laufe der Zeit lernt.
- N Hm, jetzt wo Du das sagst... Man muss ja irgendwie anfangen, sonst bleibt man ewig in der Analyse stecken und man bekommt nie ein laufendes System.
- A Gleichzeitig ist es halt so, dass man zu Beginn am wenigsten Informationen und Erfahrung hat. Daher besteht schon die Gefahr, dass man Entscheidungen trifft, die nicht auf Dauer tragen. Im einfachsten Fall kann das sein, dass man einen Begriff wählt, der letztendlich doch nicht der ideale war.
- N Da haben wir ja schon drüber geredet. Da heißt es dann umbenennen.

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

- N Bei uns im Team machen wir das inzwischen so: Wenn bei uns der Verdacht aufkommt, dass wir einen Begriff ersetzen sollten, dann nehmen wir den erstmal auf eine Liste auf und beobachten das ein wenig für uns. Die eigentliche Ersetzung ziehen wir dann mal möglichst schnell durch, wenn es ein wenig ruhiger ist. Dann haben wir nicht ganz so viel zu mergen.
- A Gute Idee, das so anzugehen. Das müssen wir auch mal ausprobieren. Oft geht es bei den Änderungen aber auch über Benennungen hinaus.

└ Gründe für Anpassungen am Modell

N Zum Beispiel?

A Oft merkt man, dass die Art der Beziehungen zwischen zwei Klassen nicht geschickt gewählt wurde. In manchen Teams ist der Reflex, dass fast alle Assoziationen bi-direktional umgesetzt werden, damit man ja von jedem Objekt aus den gesamten Graph durchlaufen kann. Das führt dann zu unnötigen Zyklen. Meistens ist die Richtung, in die ich laufe, von der Fachlogik recht klar vorgegeben. Wenn man dann merkt, dass eine Richtung nicht oder nicht mehr nötig ist, dann sollte man die dringend rausmachen. Denn das vereinfacht das Modell, was eigentlich immer gut ist.

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

- N Aber manchmal geht's auch in die andere Richtung:
Im letzten Sprint sind wir mit einer uni-direkionalen Beziehung gestartet und haben dann später gemerkt, dass das nicht reichte.
- A Wichtig ist immer, dass die Software soft bleibt. Das Modell soll möglichst geschmeidig bleiben, damit wir weiter modellieren können. Und nichts ist für die Änderbarkeit besser, als wenn man häufig Änderungen vornimmt.
- N Wahrscheinlich ist das dann auch die Argumentation, mit der ich das verkauft kriege, wenn ich schon wieder am Refactoren bin. Jeder in der Firma sollte ja ein Interesse daran haben, dass die Software nicht irgendwann total starr und unflexibel wird.

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

A Das stimmt. Wo Du gerade das Stichwort Refactoring verwendest: Manchmal genügen die klassischen Refactorings - wie die von Martin Fowler - hier nicht.

N Wie meinst Du das?

A Die meisten der klassischen Refactorings setzen eher darauf, dass man kleinere Schritte macht. So etwas wie das Umbenennen, von dem wir gerade schon sprachen. Es kommt aber auch immer wieder vor, dass man merkt, dass ein komplettes Konzept geändert werden muss.

N Woran kann ich das merken?

A Ein klassisches Symptom ist, dass ständig Begriffe verwendet werden, die sich nicht im Glossar wiederfinden

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

A und irgendwie so zwischen den Stühlen sind. Wenn das immer wieder auftaucht, dann will ich das auch im Code wiederfinden. Sobald ich im Kopf ein Mapping vornehmen oder jemanden fragen muss, wo etwas implementiert ist, bin ich wahrscheinlich schon in Schwierigkeiten. Wir haben da mal ein Bestellsystem implementiert, bei dem immer über den Status der Bestellung geredet wurde. Im Code war die entsprechende Logik dann aber verteilt auf den Kunden, die Bestellung, und die Lagerverwaltung. Die haben da alle irgendwie mitgespielt. Als wir dann den Zustand als Konzept eingeführt haben, wurde das alles sehr viel klarer.

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

- A Ein typischer Bereich, wo Konzepte erst so nach und nach auftauchen sind Validierungen und Constraints.
- N Ich hatte gedacht, dass wir das immer gleich direkt an den Objekten machen. Bei den Value Objects im Normalfall direkt bei der Erzeugung. Oft ist der Code zuerst mitten in einer Methode. Dann ziehen wir das aber immer in eine private Methode, die einen Namen bekommt, der die Fachlichkeit widerspiegelt. Damit fahren wir auch ganz gut bisher.

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

- A Das ist oft auch genau das richtige. Es kann aber auch sein, dass die Art der Validierung je nach Kontext unterschiedlich sein kann. Wenn dann das Objekt unterschiedlichste Fälle unterscheiden muss, führt das bald zur Überfrachtung mit Logik. Typischerweise ändern sich die Validierungen auch unabhängig vom Rest eines Objekts. Da kann dann eine Validierungs-Regel als Strategiekasse der bessere Weg sein, das zu modellieren.
- N Klingt sinnvoll. Je nach Kontext kann ich die entsprechende Strategie verwenden. Die einzelnen Strategien kann ich dann unabhängig ändern und auch isoliert testen, während das Kernobjekt nicht zu sehr aufgebläht wird.

└ Gründe für Anpassungen am Modell

- Grund Nummer 1: Lernen
- bessere Begriffe
- Beziehungen werden klarer
- neue Konzepte/Abstraktionen tauchen auf

N Das gefällt mir jetzt wirklich. Wenn ich dran denke, wie unaufgeräumt wir gestartet waren. Logik überall verteilt. Abhängigkeiten und Aufrufketten quer durch alle Schichten hindurch. Namen, bei denen mir die Haare jetzt zu Berge stehen. Da sind wir jetzt einen großen Schritt weitergekommen.

A Oh ja. Die Fachlichkeit steht jetzt im Zentrum und kann vernünftig getestet werden.. Neue Anforderungen können sinnvoll eingebaut werden. Und ihr spreicht im Team eine einheitliche Sprache.

N Vielen Dank Dir! Ich geh dann mal weiter programmieren!

A Tschüss, viel Erfolg!

Mit Domain-driven Design (DDD) nützliche und flexible Software bauen

Und sie programmierten glücklich
bis an ihr Lebensende...

So weit zu der Geschichte unserer beiden Helden - kehren wir zurück in die Wirklichkeit und schauen wir uns mal an, was wir von ihnen gelernt haben.