Ausarbeitung

-

Eric Evans

Domain-Driven Design

Tackling Complexity in the Heart of Software

Philipp Cordes

Kontakt: http://www.pcordes.de

Freie Universität Berlin

Fachbereich Mathematik und Informatik

Arnimallee 14, 14195 Berlin

12. Februar 2009

Inhaltsverzeichnis

1 Motivation und Übersicht **2**

2 Ubiquitous Language **4**

3 Ein Modell aus Code **5**
3.1 Entitäten 5
3.2 Value Objects 5
3.3 Services 6
3.4 Aggregate 6
3.5 Factory 7
3.6 Repositories 8
3.7 Module 9

4 Geschmeidiges Design **10**
4.1 Beispiel: Fluent Interfaces 11

5 Struktur und Kontext **12**
5.1 Verschiedene Domänen 12

6 Fazit/Einordnung **13**

1 Motivation und Übersicht

Es kann schon einige Zeit vergehen bevor wir erfahren, dass die Benutzung einer Mikrowelle eine bestimmte Reihenfolge verlangt in der die Tasten für die Bedienung zu drücken sind. In solch einem Fall würde die Anordnung oder das Design der Funktionstasten nicht explizit auf diesen wichtigen Aspekt eingehen. Ähnlich verhält es sich mit Touchpads die das Konzept Multi-Touch verwenden. Um die möglichen Gesten zu erlernen, benötigen wir entweder das Handbuch oder wir greifen auf Erfahrung zurück. Dahingegen zeigen herkömmliche Touchpads schon durch das Vorhandensein von mehreren Tasten, dass es verschiedene Funktionen gibt die wir mit diesen verwenden können. Diese machen explizit was in Multi-Touchfähigen Touchpads implizit steckt.

Mit *Domain-Driven Design* will Eric Evans explizit machen was sonst implizit im Programmcode verteilt ist. Häufig kommt es in Softwareprojekten vor, dass das was am Ende der Entwicklung entstanden ist nicht die Anforderungen des Kunden, des Benutzers entspricht. Evans sieht vor allen Dingen das unterschiedliche Verständnis von domänenspezifischen Konzepte als einen Grund für diese divergierenden Vorstellung der beiden Gruppen Benutzer und Anwendungsentwickler. Um also ein einheitliches Verständnis zwischen diesen Gruppen zu schaffen, will Evans ein Modell der Domäne etablieren. Eine vereinfachte und vor allen Dingen zweckorientierte Formulierung der relevanten Konzepte. Dabei handelt es sich hierbei nicht um ein abstraktes Modell im Sinne der *Modellgetriebenen Softwareenticklung* sondern um ein Modell, dass direkt durch Programmcode abgebildet ist. In Abbildung 1 sehen wir die Einordnung

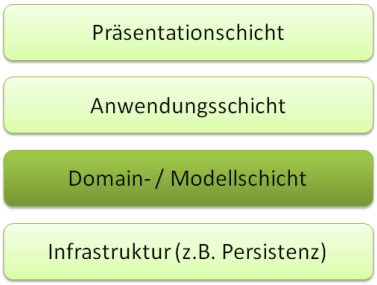

Abbildung 1: Einführung einer separaten Schicht für ein Domänen-Modell.

dieser neuen Domänen-Schicht in die Standardarchitektur eines Softwaresystems. Ziel ist es, dass sich die Anwendungsentwickler in ihrem Programmcode der Terminologie der Domäne bedienen.

Diese Ausarbeitung wird vor allen Dingen die einfachen strukturellen Konzepte erläutern, die dafür benötigt werden eine solche Domänen-Schicht zu realisieren. In [Eva04] geht jedoch Evans ebenfalls sehr genau auf den Entwicklungsprozess und die verschiedene Formen von Domänen ein.

2 Ubiquitous Language

Die Domänenschicht soll nach Evans alle wichtigen Konzepte der Domäne sammeln die sonst auf alle drei Schichten Präsentations-, Anwendungs- und Infrastrutkurschicht verteilt sind. Was alles in dieser Schicht abgebildet wird, ergibt sich aus dem Fachgebiet, den Aktivitäten und Interessen der Benutzer (siehe S. 2 in [Eva04]). Das Verständnis über die Domäne soll so fest in dieser Schicht verankert werden, dass eine Änderung Programmcode als Änderung des Modells aufzufassen ist.

Da bereits in dem jeweiligen Fachgebiet eine Terminologie, eine Sprache existiert die mit Oberbegriffe und Bedingungen komplexe Konzepte beschreibt, sieht Evans gerade hier die wichtigen Konzepte einer Domäne. Die *Ubiquitous Language* , die allgegenwärtige Sprache, ist nach Evans eines der wichtigsten Schlüsselkonzepte die für die Etablierung eines einheitlichen Domänen-Modells sorgen sollen. Es verwundert also nicht, dass Evans viel Kontakt zum Kunden für den Prozess der Modellierung vorsieht.

Die *Ubiquitous Language* ist das zentrale Schlüsselkonzept von *Domain-Driven Design* . Sie stellt das Bindeglied zwischen Domänenexperten und Anwendungsentwicklern dar. Die Umsetzung dieser im Programmcode führt zu einem Modell der Domäne. Für die Realisierung der *Ubiquitous Language* auf Ebene des Programmcodes stellt Evans Muster und Strategien bereit.

3 Ein Modell aus Code

Die wichtigste Unterscheidung die Evans zwischen Modellelementen [1] macht, ist ob ein Element eine Identität hat oder nicht. Objekte müssen beispielsweise identifizierbar sein, um sie von einander unterscheiden zu können oder zu einem späteren Zeitpunkt wieder zu finden um weitere Operationen mit diesen durchführen zu können. Andere Objekte stellen nur die Repräsenation einer Eigenschaft dar. Im sogenannte *Buidling Block* stellt Evans einge Muster

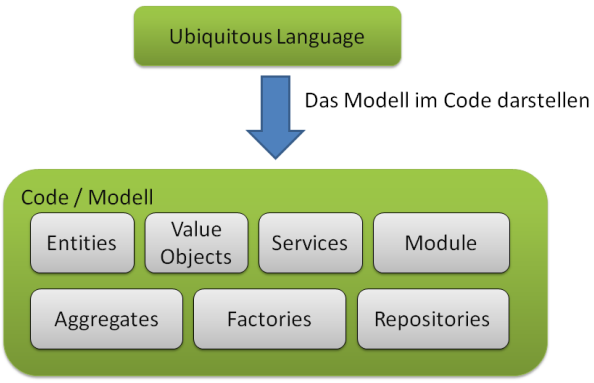

Abbildung 2: Einführung einer separaten Schicht für ein Domänen-Modell.

zusammen die für die Umsetzung eines Modell aus Code-Ebene relevant sind.

3.1 Entitäten

In einem Online-Shop kommt es schnell zu Stande, dass wir mehrere Kunden mit dem gleichen Namen wieder finden. Um diese trotzdem klar voneinander unterscheiden zu können verwenden wir häufig Identifikationsnummern (IDs). Solche Objekte bezeichnet Evans als *Entity* . *Entities* repräsentieren zumeist die releventesten Businessobjekte einer Domäne. Die Identifizierbarkeit muss dabei nicht von einer einzigen Eigenschaft abhängen. Ähnlich wie bei Datenbanken kann diese auch durch eine Menge von Eigenschaften realisiert sein. Wichtig ist dabei nur, dass wir gleiche Objekte voneinandere unterscheiden können.

3.2 Value Objects

Im Gegensatz zu *Entities* stellen *Value Objects* eine Klasse von Objekten dar die nicht identifizierbar sind. Meist handelt es sich dabei um Werte die durch einfache Datentypen der Progammiersprache nicht repräsentiert werden können. Ein Beispiel hierfür wäre die Repräsentation von einer Adresse welche sich

[1] Ein Modellelement im Kontext von Domain-Driven Design ist als ein Muster/Klasse zu verstehen welches ein Konzept aus der Domäne im Programmcode wiederspiegelt.

aus den drei Werten für die Straße, die Stadt und Postleitzahl zusammensetzen könnte. Um den Charakter eines Wertes zum Ausdruck zu bringen ist die Änderung eines *Value Objects* in den meisten Fällen nicht vorgesehen. Das zwingt uns dazu, dass für die Änderung einer Adresse das entsprechende Objekt durch ein neues ersetzt werden muss. Im Programmcode sind wir so gezwungen explizit aufzuschreiben, dass wir gerade die Adresse einer *Entity* ändern.

Listing 1: Value Objects ersetzen

```
1 Address address = AddressFactory.getNewAddress(street, city, zip);
2 customer.changeShippingAddress(address);
```

3.3 Services

Entities wie auch *Value Objects* bewahren selbstverständlich Methoden auf die zum Beispiel für die Modifikation oder den Vergleich dieser gebraucht werden. Funktionalitäten die von Natur aus, also durch die Eigenschaften der Domäne diesen nicht zugeteilt werden können, werden in sogenannten *Services* untergebracht. Die Extraktion solcher Funktionalitäten führt unter anderem dazu, dass man explizit diese aufrufen muss und somit keine impliziten Annahmen an das Verhalten einer Methode machen muss. Zudem verdeutlichen diese domänenspezifische Eigenschaften.

Listing 2: Ein Services zu Ermittlung von Geo-Daten.

```
1 GeoPosition position = GeoService.getPositionOf(address);
```

Das Beispiel für einen GeoService macht sofort deutlich, dass die Ermittlung von Geo-Daten zu einer Adresse offenbar mit Kosten für die Laufzeit verbunden ist. Wäre die Ermittlung dieser Daten z.B. durch das Proxy (siehe [GHJV04]) Muster in dem entsprechenden *Value Object* untergebracht worden, könnte man dies ohne Wissen über die Implementierung einer entsprechenden Methode nicht erahnen.

3.4 Aggregate

Mit Hilfe von *Entities* und *Value Objects* können wir also ein Geflecht von Objekten erstellen die ein Modell der Domäne repräsentieren. Bei größeren Modellen gehen hierbei jedoch Konturen verloren die nicht nur Einfluss auf das Verständnis des Modells haben, sondern auch die Implementierung stark beeinflussen. Das Muster *Aggregate* stellt diese Konturen wieder her. *Aggregates* fassen mehrere Elemente (*Entities* und *Value Objects*) zusammen und repräsentieren diese nach außen hin als ein Modellelement. Da *Aggregates* in der

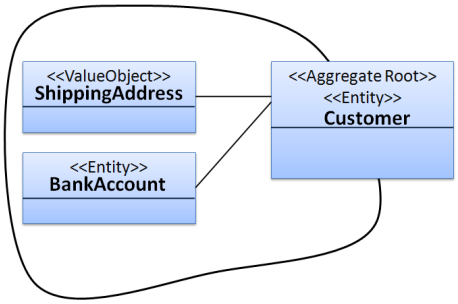

Abbildung 3: Beispiel für ein *Aggregate* .

Regel identifizierbar sind übernimmt eine *Entity* die Rolle der sogenannte Aggregatwurzel. Dem Entwurfsmuster Fassade (siehe [GHJV04]) entsprechend, stellt dann diese Wurzel Funktionalitäten bereit um interne Elemente zu modifizieren.

3.5 Factory

Bereits mit der Einführung der *Aggregates* nehmen wir Einfluss auf den Lebenszyklus eines Objektes. Denn die Existenz von internen Elemente eines *Aggregate* hängt von der Existenz der Wurzel dessen ab. Wird diese gelöscht werden auch alle anderen Objekte des *Aggregate* nicht mehr benötigt. In Abbildung 4 können wir einen stark vereinfachten Lebenszyklus eines Objektes sehen. Das Löschen eines Elements wird zum großen Teil durch Abhängigkeiten zu *Entities* realisiert. Für die Erzeugung sieht Evans das Muster *Factories* (siehe [GHJV04]) vor. Im Grunde handelt es sich bei diesen um *Services* die einem ganz bestimmten

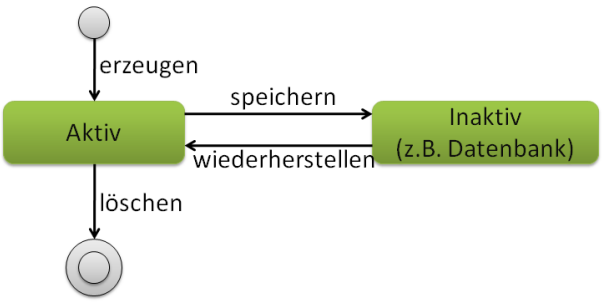

Abbildung 4: Stark vereinfachter Lebenszyklus eines Objektes.

Zweck dienen; valide Objekte zu erzeugen. Wie *Aggregates* bereits zeigen, kann es sich bei Elementen um eine Komposition von mehreren Elementen handeln. Die Zusammensetzung dieser kann gewissen Regeln folgen, die nach außen hin nicht relevant sein sollen. Darum wird die Konstruktion von neuen Objekten

in *Factories* gekapselt.

3.6 Repositories

Mit *Factories* haben wir ein Mittel, dass uns die Konstruktion von neuen komplexen Objekten ermöglicht. Diese Objekte bleiben vermutlich eine Zeit lang in einem aktiven Zustand, in dem diese modifziert oder andersweitig ausgewertet werden. Objekte die wir zu einem späteren Zeitpunkt wieder benötigen, werden in einen inaktiven Zustand versetzt (siehe Abb. 4). Um jedoch auf Objekte die z.B. in einer Datenbank abgelegt wurden, wieder zurückgreifen zu können, müssen wir zunächst die Daten aus der Datenbank verarbeiten, um somit das gewünschte Objekt wiederzubeleben. Dabei müsste die Anwendungsschicht auf Funktionalitäten zurückgreifen die spezifisch für Datenbanken wären. Damit würden wir das Vokabular der Domäne verlassen und Konzepte benutzen die so in dieser nicht vorkommen. Zudem verbergen sich in der Selektion von Objekte aus der Persistenzschicht Eigenschaften die womöglich relevant für die jeweilige Domäne sind. Um auch diese relevanten Eigenschaften mit in die Domänen-Schicht aufzunehmen, führt Evans das Muster der *Repositories* ein. Diese stellen Funktionalitäten bereit die nach außen hin (also zur Anwendungsschicht) eine In-Memory-Repräsenation der Objekte simuliert. Zudem haben wir so die Möglichkeit die Selektion von Objekten in der Sprache der Domäne zu abstrahieren. Das führt dazu, dass die Anwendungsschicht nicht mehr die Terminologie der Domäne verlässt. Somit stellen *Repositories* ein wichtiges Werkzeug dar um die *Ubiquitous Language* zu realisieren.

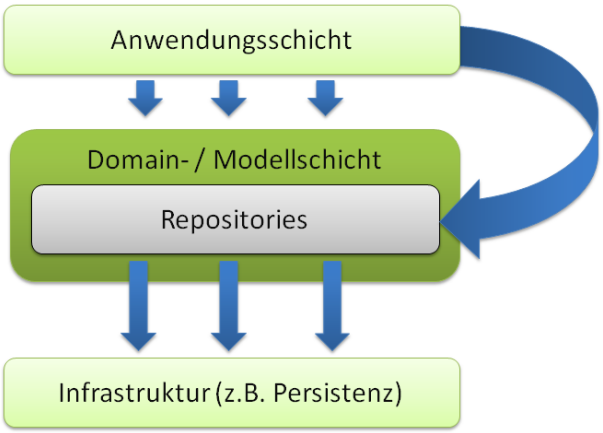

Abbildung 5: Repositories

3.7 Module

Als letztes Element des sogenannten *Buidling Blocks* seien hier noch die Module erwähnt. Diese stellen ähnlich wie *Aggregates* die Möglichkeit bereit weitere Konturen zu schaffen. Dabei wird das Modell in Form von Packeten unterteilt.

4 Geschmeidiges Design

Neben den Mustern zur Erstellung eines Modells (*Buidling Block*) auf Ebene des Programmcodes, gibt Evans noch Muster für das Design (*Supple Design*) der Domänschicht an. Zweck dieser ist es, dass die Benutzer der Domänenschicht klare Annahmen darüber treffen können wie die Funktionalitäten zu verstehen sind und welche Wirkungen diese haben. Die drei wichtigsten Prinzipien sind hier:

1. Intention-Revealing Interfaces
2. Side-Effect-Free Functions
3. Assertions

Damit Benutzer von Schnittstellen nicht zusätzlich Dokumentation über die das Verhalten dieser lesen müssen, sollten diese soweit es geht sich selbst erklären. Dabei sollen Intention-Revealing Interfaces nicht nur durch den Namen der Methoden oder Klassen, sondern auch durch Abbildung domänenrelevanter Aspekte den Benutzer zu klaren Annahmen über die Wirkung führen. Side-Effect-Free Functions sollen das implizite Verändern von Objekten verhindern. Das heißt, dass der Aufruf solcher Methoden immer zur Rückgabe eines neuen Objektes führt. Gerade weil *Value Objects* in der Regel unveränderbar sind, bieten sich für diese seitenfreien Funktionen an. Ein weiterer Vorteil solcher Funktionen ist der, dass diese wesentlich leichter miteinander kombiniert werden können.

Listing 3: Side-Effect-Free Functions

```
1 GeoPosition newPosition = anGeoPosition.addToNorth(2.0);
```

Funktionalitäten die auf *Entities* operieren, können in diesem Sinne nicht implementiert werden. Hier schlägt Evans vor, dass die Wirkung dieser durch explizite Assertions verdeutlicht wird. Dies könnte z.B. durch Annotationen oder Unit-Tests geschehen.

Listing 4: Assertsions

```
1 ...
2 @Test
3 public void test() {
4     // changing the address don't changes the geo-position ...
5     assertEquals(48.12, customer.getPosition().getLongitude());
6     customer.changeAdress(newAddress);
7     assertEquals(48.12, customer.getPosition().getLongitude());
8 }
```

4.1 Beispiel: Fluent Interfaces

Fluent Interfaces stellen ein Beispiel für Schnittstellen dar, die alle drei Kriterien Intention-Revealing Interfaces, Side-Effect-Free Functions und Assertions erfüllen. Es handelt sich hierbei um Schnittstellen die Funktionalitäten beinah in Satzform bereitstellen können. Eine Möglichkeit solche *Fluent Interfaces* zu realisieren sind Methodenketten, also die Aneinenaderreihung von Methodenaufrufen. Der Vorteil solcher flüssige Schnittstellen ist, dass man explizit erkennen kann welche Wert was für eine Rolle bei der Auswertung übernehmen.

Listing 5: Beispiel für ein Fluent Intface

```
1 Date result;
2 result = newDate().years(2009).months(06).days(07).o();
```

Zudem kann man bei der Realisierung eine Grammatik umsetzen, die dazu führt, dass nur gültige Methodenketten verasst werden können. Damit erklären nicht nur die Namen der Methoden die Benutzung dieser Schnittstellen.

Als ein konkretes Beispiel für ein Fluent Interface sei hier das Entwurfsmuster Specification genannt, welches von Martin Fowler und Eric Evans erfasst wurde. Specifications haben zum Zweck Bedingungen explizit zu machen. Häufig finden sich solche Prüfungen in Form von if-else-Klausel im Programmcode wieder. Specifications kapseln diese Klausel als relevante Domämenenbedingungen. Fowler und Evans zeigen in [MF] eine mögliche Art der Implementierung, bei der Specifications flüssig miteinander kombiniert werden können, um neue Specifications zu erzeugen.

Listing 6: Specifications

```
1 Specification colorSpec = new ColorSpecification();
2 Specification lengthSpec = new LengthSpecification();
3
4 if(colorSpec.and(lengthSpec).isSatisfiedBy(car)) {
5 ...
6 }
```

Da es sich bei Specifications um *Value Objects* handelt können die Methode seitenfrei implementiert werden. Die Bedienung dieser Schnittstelle erklärt sich beinah selbst und lässt uns klare Annahmen über die Wirkung der Aufrufe treffen [2].

[2] Unter http://www.fluent-interfaces.com finden sich weitere Details zu *Fluent Interfaces*

5 Struktur und Kontext

Neben den Mustern aus dem *Buidling Block* und den Design Mustern gibt es noch Architekturmuster die Evans in [DDD] beschreibt. Die meisten befassen sich mit weit abstrakteren Formen als die vorigen Muster.

Häufig wird es in einem Projekt vorkommen, dass nicht nur ein Domänen-Modell entsteht. Die Domänenschicht wird wohl eher aus mehreren Modellen bestehen. Diese bezeichnet Evans als *Bounded Context* als eine abgeschlossene Einheit, in der alle nötigen *Entities* , *Value Objects* , *Services* , *Repositories* etc. gebündelt werden.

Solche Kontexte können auch für die Aufteilung in der Teamarbeit dienen. Wie in der Teamarbeit kann es bei mehreren *Bounded Contexts* zu Übebrschneidungen kommen. Diese Überschneidungen werden als *Shared Kernel* bezeichnet. Ein solcher hat zum Zweck explizit zu machen, um welchen Teil der jeweiligen Kontexte es geht. Zudem macht die Bezeichnung eines solchen klar worüber es gilt zu reden. Eventuell erkennt man auch so, ob vielleicht für das Projekt statt zwei doch nur ein *Bounded Context* angebracht wäre.

Die Wiederverwendung von altem oder fremden Programmcode könnte dazu führen, dass die *Ubiquitous Language* durchbrochen wird und Konzepte in die Sprache Einzug halten, die für die Problem-Domäne nicht relevant sind. Für diesen Fall sieht Evans einen sogenannten *Anticorruption Layer* vor, der solchen Code (wie z.B. Frameworks) kapselt und in die *Ubiquitous Language* einbettet.

5.1 Verschiedene Domänen

Auf einer noch weiter abstrakteren Ebene beschreibt Evans welche verschiedene Arten von Domänen in der Domänschicht vorkommen können. So sollte es in der Regel eine kleine Kerndomän (*Core Domain*) geben die ein Destillat der wichtigsten Konzepte für das Projekt darstellt. Überschneiden tun sich mit dieser verschiedene andere Arten von Domänen, die der Einfachheit halber hier nicht weiter erwähnt werden. Eine Bezeichnung die für das Verständnis einer Domän wichtig ist, ist die der *Generic Subdomain*. Dabei handelt es sich um Domänen die unabhängig von der aktuellen Problem-Domän in vielen Projekten gleich verwendet werden kann. Ein Beispiel für eine solche *Generic Subdomain* ist die Repräsentation von Geld.

6 Fazit/Einordnung

Mit seinem Buch *Domain-Driven Design* sammelt Eric Evans Muster und Strategien die vor allen Dingen eins zum Zweck haben. Eine Sprache im Projekt zu etablieren die Kunden und Entwickler von den gleichen Konzepten reden lässt. Somit sollen die verschiedenen Vorstellungen dieser beiden Gruppen näher zusammengebracht werden. Dabei stellt Evans ein kleines Vokabular bereit, welches sich unter anderem dadurch auszeichnet, dass es sich nicht der Terminilogie einer Programmiersprache bedient. Somit können Anwendungsentwickler auf abstrakter Ebene über die Strukter, also das Modell, des Codes reden.

Literatur

[Eva04] Eric Evans. *Domain-Driven Design - Trackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[GHJV04] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, München, Juli 2004.

[MF] Eric Evans Martin Fowler. Specifications. http://martinfowler.com/apsupp/spec.pdf Zuletzt aufgerufen am 12.02.2009.