Domain Driven Design

Lukas Panni
TINF18B5

DHBW Karlsruhe
Vorlesung Advanced Software Engineering Semester 5/6

Inhaltsverzeichnis

1 Analyse der Ubiquitous Language

Kern des *Domain Driven Design* ist das gemeinsame Verständnis der Problemdomäne. Dies ist notwendig, um ein präzises Modell der Domäne erstellen zu können. Um die Kommunikation zwischen Entwicklern und Domänenexperten zu ermöglichen wird die sogenannte **Ubiquitous Language** verwendet. Da weder Domänenexperten die Sprache der Entwickler, noch umgekehrt die Entwickler die Sprache der Domänenexperten verstehen ist die Ubiquitous Language als gemeinsame Sprache unerlässlich. Diese wird von Entwicklern und Domänenexperten gleichermaßen verwendet, um Begriffe, Prozesse und Konzepte eindeutig zu bezeichnen. Wenn die Ubiquitous Language korrekt eingesetzt wird werden in der Domäne und im Quellcode der Anwendung die gleichen Begriffe für die gleichen Konzepte verwendet. So können Missverständnisse und Mehrdeutigkeiten sicher vermieden werden, was gerade bei komplexen Domänen den Softwareerstellungsprozess erleichtert. Beim definieren der Ubiquitous Language sollte der Fokus auf die Begriffe und Konzepte der Kerndomäne gelegt werden. Andere Bereiche der Domäne können auch weniger genau modelliert werden, damit der Aufwand angemessen bleibt.

Die Kerndomäne dieses Projekts ist die Übersicht über Beteiligungen (Contributions) an Öffentlichen Repositories auf der Plattform *GitHub*. Die Ubiquitous Language ist dabei vergleichsweise einfach zu definieren, da sich die Anwendung an Entwickler richtet, die in diesem Fall gleichzeitig auch Domänenexperten sind. Trotzdem muss darauf geachtet werden, dass einheitliche Begriffe verwendet werden. Da die Plattform GitHub eindeutige Begriffe für die Bezeichnung von verschiedenen, wichtigen Konzepten verwendet und kein Einfluss auf diese Bezeichnungen besteht, wurde entschieden diese Bezeichnungen möglichst zu verwenden. Dabei werden diese Bezeichnungen nicht nur im Code sondern auch im User-Interface eingesetzt.

Repository / Repositories

Der Begriff Repository beschreibt in diesem Kontext ein Verzeichnis, das zur Ablage von z.B. Programmcode verwendet werden kann. Dabei bietet ein Repository bei GitHub aber auch noch weitere Funktionen, die hier aufgrund der Fehlenden Relevanz für das System nicht genauer erläutert werden. Das Verständnis eines Repositoriy als Verzeichnis zur Ablage von Daten ist ausreichend. Der Begriff wird sowohl im Code als auch im User-Interface verwendet. Beispiele für die Verwendung des Begriffs im Code finden sich in der Klasse *RepositoryDataResponse*, die Daten zu einem Repository speichert, die von der API abgerufen wurden und in der Klasse *ContributionRepositories*, die beschreibt, in welchen Repositories Contributions durchgeführt wurden.

Beim Begriff Repository existiert allerdings eine Doppeldeutigkeit, da der Begriff Repository innerhalb des Systems auch als Bezeichnung für Klassen, die das *Repository-Pattern* implementieren verwendet wird. Das führt zum Beispiel zu der Mehrdeutigen Bezeichnung der Klasse RepositoryDataRepository. Diese Mehrdeutigkeit kann nur schwer beseitigt werden, da einerseits der Repository-Begriff aus der Domänensprache verwendet werden soll, andererseits aber auch Repository-Pattern-Klassen eindeutig gekennzeichnet werden sollen. Deshalb wird diese Mehrdeutigkeit akzeptiert und festgelegt, dass Klassen die das Repository-Pattern implementieren, den Begriff Repository am Ende des Klassennamens enthalten. Kommt der Begriff an anderen Stellen vor, kann mit hoher Wahrscheinlichkeit davon ausgegangen werden, dass ein Repository im Sinne der Domäne gemeint ist.

Contribution / Contributions

Beteiligungen an Repositories werden auf GitHub als Contributions bezeichnet. Die API, die verwendet wird um Daten abzurufen, verwendet ebenfalls diese Bezeichnung. Folglich soll auch im Code diese Bezeichnung für das gleiche Konzept verwendet werden.
Der Begriff Contributions ist allerdings nur ein Sammelbegriff für vier weitere Konzepte, die von großer Bedeutung für die Ubiquitous Language der Problemdomäne sind. Diese Konzepte sind *Commits*, *Issues*, *Pull Requests* und *Pull Request Reviews*, und werden im Folgenden beschrieben. Die jeweilige Anzahl in einem bestimmten Zeitraum lässt sich über die GitHub-API abrufen und ist innerhalb dieser Anwendung von hoher Relevanz, da aus diesen Daten verschiedene Statistiken erstellt und dem Nutzer angezeigt werden.

Commit / Commits

Ein Commit kann in diesem Kontext als Prozess vorgenommene Änderungen am Quellcode zu bestätigen verstanden werden. Die Anzahl der Commits in einem bestimmten Zeitraum erfasst so, wie viele unterschiedliche Änderungen ein Nutzer in diesem Zeitraum auf GitHub öffentlich gemacht hat.

Issue / Issues

Issues werden bei GitHub verwendet um Ideen, neue Features, Aufgaben und Bugs zentral zu verwalten und stellen die zweite Art der Contribution dar. Die Anzahl der Issues beschreibt, wie viele neue Issues ein Nutzer in einem bestimmten Zeitraum erstellt hat. Diese Beschreibung reicht für das Verständnis im Rahmen dieses Projekts aus.

Pull Request / Pull Requests

GitHub verwendet sogenannte Pull Requests um Änderungen Code aus einem Branch in einen anderen, z.B. den main-Branch, zu übernehmen. Dabei bietet ein Pull Request die Möglichkeit weitere Tests oder auch ein Code-Review durchzuführen, bevor der Pull Request angenommen wird. Damit stellen Pull Requests ein wichtiges Konzept dar, das die Kollaboration erleichtert. Die Anzahl an Pull Requests beschreibt, wie viele Pull Requests eine Nutzer in einem bestimmen Zeitraum erstellt hat.

Pull Request Review / Pull Request Reviews

Ein Pull Request Review beschreibt bei GitHub ein Code-Review im Rahmen eines Pull Requests. Das Kommentieren von Änderungen eines anderen Nutzers, die dieser in Form eines Pull Requests einreicht, wird dabei als Pull Request Review gewertet. Die Anzahl von Pull Request Reviews beschreibt, wie viele Code-Reviews von Pull Requests anderer Nutzer durchgeführt wurden.

Die Klasse *ContributionCount* ist ein Beispiel dafür, wie alle Bezeichnungen, die im Zusammenhang mit Contributions stehen, auch im Code verwendet werden. Dies wird in Listing 1 nochmals verdeutlicht. Die Klasse ContributionCount hat je eine Instanzvariable für die Anzahl von Commits, Issues, Pull Requests und Pull Request Reviews und verwendet somit alle vorgestellten Bezeichnungen, die in der Domäne im Zusammenhang mit Contributions verwendet werden.

```
public final class ContributionCount {

    private final int commitCount;
    private final int issueCount;
    private final int pullRequestCount;
    private final int pullRequestReviewCount;

    public ContributionCount(int commits, int issues, int
       pullRequests, int pullRequestReviews) {
        this.commitCount = Math.max(commits, 0);
        this.issueCount = Math.max(issues, 0);
        this.pullRequestCount = Math.max(pullRequests, 0);
        this.pullRequestReviewCount =
           Math.max(pullRequestReviews, 0);
    }
```

```
[...]
}
```

Listing 1: Auszug aus der Klasse ContributionCount

Auch im User-Interface werden diese Bezeichnungen gemeinsam verwendet, wie zum Beispiel die UI-Klasse *OverviewCard* zeigt, die ebenfalls all diese Bezeichnungen verwendet. Generell sind diese Begriffe von großer Bedeutung für das System und werden deshalb noch an vielen weiteren Stellen verwendet. Weitere Beispiele sind unter anderem die Klasse *ContributionCountChange*, die eine Veränderung der Contribution-Anzahl im Vergleich zu einem anderen Zeitraum beschreibt und die zugehörige UI-Klasse *ProgressCard*.

2 Analyse und Begründung von Value Objects

ValueObjects sind vergleichsweise einfache Objekte, die nur durch ihre Werte beziehungsweise Eigenschaften beschreiben werden. Das hat unter anderem zur Folge, dass ValueObjects als gleich angesehen werden, wenn ihre Werte gleich sind. Allgemein beschreiben ValueObjects oft eine bestimmte Sache durch die Kombination mehrerer Attribute dieser Sache. Typische ValueObjects sind unter anderem Adressen und Geldbeträge. Dabei sind ValueObjects außerdem unveränderbar, so dass alle Werte bereits im Konstruktor gesetzt werden müssen. Diese scheinbare Einschränkung der Unveränderlichkeit kann vorteilhaft sein, da zum Beispiel die Gültigkeit von Werten nur an einer Stelle (im Konstruktor) geprüft werden muss und ein so erstelltes ValueObject zu jeder Zeit gültig ist. Damit sind ValueObjects auch leicht testbar, da nur die Gültigkeitsprüfungen im Konstruktor getestet werden müssen. Die Test-Klassen ContributionCountUnitTest und ContributionCountChangeUnitTest sind gute Beispiele für Tests von ValueObjects, die hier verwendet wurden.
Umgesetzt werden ValueObjects, indem jegliche Änderungen an Werten des Objekts eingeschränkt werden. Dazu bieten verschiedene Programmiersprachen verschiedene Sprachfeatures.
Konkret bedeutet das für die hier verwendete Sprache Java:

- Klasse muss als final markiert sein, sodass auch keine Veränderungen durch Vererbung möglich sind
- Alle Membervariablen müssen ebenfalls als final gekennzeichnet werden, sodass die Variablen spätestens im Konstruktor gesetzt werden und sonst nicht mehr geändert werden können
- Dabei ist darauf zu achten, dass über den Konstruktor auch nur gültige Objekte erstellt werden können
- Die *equals* und auch die *hashCode* Methoden müssen geeignet überschrieben werden, sodass die Gleichheit durch gleiche Werte der Membervariablen festgelegt wird

2.1 Verwendete ValueObjects

Im *data*-Package existieren mehrere Klassen, die **ValueObjects** darstellen.
Diese Klassen sind:

- **ContributionCount**
  Stellt die Anzahl von GitHub-Contributions dar, ausgedrückt als Sammlung der Anzahl von Commits, Issues, Pull-Requests und Pull-Request-Reviews.

- **ContributionCountChange**
  Repräsentiert die Differenz zweier ContributionCount-Objekte und bietet die Möglichkeit die Unterschiede zwischen aktueller und vorangegangener Periode in Bezug auf Commits, Issues Pull-Requests und Pull-Request-Reviews zu berechnen.

- **ContributionRepositories**
  Sammelt die Repositories, für die Commits, Issues, Pull-Requests und Pull-Request-Reviews durchgeführt wurden.

- **TimeSpan**
  Repräsentiert eine Zeitspanne, bestehend aus einem Start- und einem Enddatum.

- **RepositoryName**
  Repräsentiert den vollen Namen eines GitHub-Repositories, der sich aus Besitzer des Repositories und Bezeichnung des Repositories zusammensetzt.

Verwendet werden diese ValueObjects aufgrund ihrer Vorteile, wie zum Beispiel der leichten Testbarkeit. So kann leicht sichergestellt werden, dass das Verhalten der Objekte den Erwartungen entspricht.
Objekte der Klasse *TimeSpan* zum Beispiel werden für das Caching, für das Abrufen von Daten und auch im User-Interface für die Anzeige von Zeiträumen eingesetzt. Die gute Testbarkeit von ValueObjects ist dadurch an mehreren Stellen von Vorteil und erleichtert die Entwicklung, da davon ausgegangen werden kann, dass sich TimeSpan-Objekte korrekt verhalten. Für die anderen genannten ValueObjects sind die Vorteile ähnlich.

3 Analyse und Begründung Entities

Entities haben im Gegensatz zu ValueObjects innerhalb der Domäne eine eindeutige ID, über diese sie identifiziert werden können, und die die Gleichheit von Entities bestimmt. Zusätzlich können sich Entities auch während ihrer Lebenszeit verändern und sind nicht immutable. Auch bei Entities ist es wichtig, dass sie nicht mit unerlaubten Werten initialisiert werden können und darüber hinaus auch nicht durch eine Aktion ungültige Zustände erreichen dürfen.
Aktuell konnten keine Objekte innerhalb des Projekts identifiziert werden, die die Eigenschaften einer Entity erfüllen. Der Grund dafür ist vermutlich, die vergleichsweise geringe Komplexität des Gesamtsystems.

4 Analyse und Begründung von Aggregates

Aggregates sind ein Konzept zur Zusammenfassung von mehreren Entities und ValueObjects zu einer Einheit. Als Einheit werden Aggregates auch immer nur vollständig geladen und gespeichert, sodass ein Aggregate immer nur vollständig vorhanden sein kann. Eingesetzt werden Aggregates um die Beziehungen zwischen einzelnen Entities und ValueObjects zu vereinfachen, indem mehrere Entities und ValueObjects in einem Aggregate zusammengefasst werden und Beziehungen zu den enthaltenen Entities und ValueObjects nur über das Aggregate realisiert werden. Außerdem bildet auch ein einzelne Entity bereits ein Aggregate. Zur Verwaltung des Zustands eines Aggregates gibt es immer eine Aggregate Root Entity, über die alle Zugriffe auf das Aggregat durchgeführt werden. So können an einer Stelle alle Zugriffe auf Validität geprüft werden, was die Prüfung von Domänenregeln einfacher macht.
Aggregates sind vergleichsweise komplex und kommen deshalb auch häufig nur in komplexen Systemen eingesetzt. Es gibt im Projekt keine klassischen Aggregates, die mehrere Entities und ValueObjects zusammenfassen, da es bereits keine echten Entities gibt. Die Klasse *UserContributionsResponse* ist einem Aggregate ähnlich, da hier mehrere ValueObjects zusammengefasst werden. Da diese Klasse allerdings kein Verhalten auf den enthaltenen ValueObjects durchführt, sondern diese nur erstellt, speichert und bei Bedarf zurückgibt, kann man diese Klasse nicht als klassisches Aggregate ansehen.

5 Analyse und Begründung Repositories

Repositories stellen die Schnittstelle zwischen Domäne und Datenmodell dar und stellen demnach Methoden bereit um Daten zu lesen, speichern oder zu löschen. Dabei werden Repositories häufig eingesetzt, um einen einfachen Datenzugriff für die Domäne zu ermöglichen und die technischen Details, wie zum Beispiel Datenbankzugriffe, zu kapseln. Normalerweise operieren Repositories auf der Basis von Aggregates. Um die Domäne frei von technischen Details zu halten werden Interfaces für Repositories innerhalb der Domäne verwendet und die eigentliche Implementierung findet außerhalb der Domäne statt. Zusätzlich bieten Repositories die Möglichkeit nicht nur allgemein Daten abzurufen, sondern Daten anhand von bestimmten Kriterien zu finden, was den Zugriff auf Daten für die Domänenobjekte erleichtert.
Da, wie bereits beschrieben, keine Aggregates verwendet werden, finden sich auch keine typischen Repositories. Stattdessen gibt es im *repository*-Package mehrere Klassen, die die Bezeichnung Repository tragen. Diese erfüllen einen ähnlichen Zweck und dienen ebenfalls der Kapselung technischer Details des Datenzugriffs. Sie rufen Daten aus einem internen Cache ab oder verwenden die GitHub-API um die Daten zu laden und im Cache zu speichern. Allerdings bieten sie nicht die Möglichkeit Daten zurückzugeben. Stattdessen werden den Datenzugriffsfunktionen Callback-Funktionen übergeben, die mit den abgerufenen Daten aufgerufen werden. Dies ist eine bewusste Entwurfsentscheidung, die getroffen wurde, um nicht nur die Anbindung an die verwendete GitHub-API und die gleichzeitige Verwendung eines eigenen Caching-Mechanismus zu erleichtern. Zusätzlich wird so eine Asynchrone Datenverarbeitung gefördert, die gerade bei hohen Netzwerklatenzen bei Mobilgeräten, und vielen einzelnen API-Anfragen die Wartezeit für den Nutzer merklich verringern kann.

6 Fazit Domain Driven Design

Domain Driven Design umfasst verschiedene Konzepte, die genutzt werden können, um den Fokus auf die eigentliche Domäne zu steigern und die zusätzliche Komplexität zu verringern. Viele dieser Konzepte sind allerdings sehr aufwändig und der Nutzen wird häufig erst in größeren Anwendungen wirklich sichtbar. Im Rahmen dieser Arbeit wurden verschiedene Konzepte des Domain Driven Design genutzt. In *Analyse der Ubiquitous Language* wurde gezeigt, wie das Konzept einer einheitlichen Sprache sinnvoll genutzt wurde um Mehrdeutigkeiten möglichst zu vermeiden. Auch ValueObjects wurden genutzt, um bestimmte Dinge, wie zum Beispiel GitHub-Contributions, abbilden zu können und die Vorteile von unveränderlichen Objekten zu nutzen. Andere Konzepte, wie zum Bei-

spiel Entities, Aggregates und Repositories wurden bewusst nicht, beziehungsweise nicht im eigentlichen Sinne, verwendet, da der Aufwand für die Verwendung dieser Konzepte in Relation zum Nutzen für ein vergleichsweise kleines System nicht gerechtfertigt ist. Zusätzlich beruht diese Entscheidung auf auf der Tatsache, dass die Verwendung eines Konzepts nur des Konzepts wegen nicht sinnvoll ist.