Gut bewertet

Domain-Driven Design in Clojure

Phillip Ghadir, Philipp Schirmacher

In dieser Kolumne setzen wir die Prinzipien des Domain-Driven Designs (DDD) mit Clojure um. Wir stellen die Domäne Rating als Beispiel vor und demonstrieren strategisches sowie taktisches DDD. Wir zeigen, wie sich das Domänenmodell in Clojure implementieren lässt.

Obwohl Domain-Driven Design (DDD) meist im Zusammenhang mit Objektorientierung genannt wird, sind die Prinzipien auch gut mit einer funktionalen Sprache umsetzbar. Eine Einführung in Clojure, dem Lisp für die JVM, bieten beispielsweise Neppert und Tilkov in einer Artikelserie in Java-SPEKTRUM. Auch in der Praktiker-Kolumne selbst war Clojure und die Integration mit Java bereits ein Thema [Ghad12]. Das Domain-Driven Design ist aber in dieser Kolumne noch neu. Daher stellen wir das nun kurz vor.

Domain-Driven Design

Domain-Driven Design bietet zwei Facetten: Das strategische DDD liefert Hilfestellungen für die Systementwicklung im Großen. Das taktische DDD ist unserer Erfahrung nach deutlich bekannter und ähnelt der objektorientierten Analyse und Design, geht aber darüber hinaus.

Ein wesentliches Konzept des Domain-Driven Designs ist die Definition von klar abgegrenzten fachlichen Bereichen. Diese nennt man Bounded Contexts. Für einen solchen Kontext modellieren Techniker und Domänenexperten gemeinsam, wie die Software die fachlichen Prozesse am Besten umsetzt.

Größere Systeme decken meist mehrere (Sub-)Domänen ab, die dann abhängig von organisatorischen Rahmenbedingungen über verschiedene Strategien integriert werden. Wir werden im Folgenden die Shared-Kernel-Strategie kennenlernen. Der Textkasten „Integrationsstrategien im DDD" liefert einen Überblick über die verschiedenen Integrationsstrategien. Weitere Informationen bieten [Evans03] oder auch [Vernon13].

Beim *strategischen DDD* konzentriert man sich in erster Linie auf das Identifizieren und Abgrenzen der fachlichen Kontexte (der Bounded Contexts) und das Identifizieren von (Sub-)Domänen.

Für die Modellierung einer Domäne stehen beim *taktischen DDD* verschiedene Stereotype zur Verfügung: Aggregat, Entität, Wertobjekt, Service, Repository und einige mehr. Jede dieser Arten unterliegt eigenen Regeln für zulässige Abhängigkeiten und Verantwortung im System.

Im Folgenden stellen wir die Domäne Rating als Beispiel vor und beschränken uns zu Beginn auf das strategische Modellieren.

Isolationsschicht (Anticorruption Layer)	Änderungen aus fremden Modellen können in einer separaten Schicht gekapselt werden, sodass sich die internen und externen Modelle unabhängig voneinander weiterentwickeln können
Konformist (Conformist)	Ohne Mitbestimmungsrecht und ohne gesonderte Strategie zur Entkopplung wird das externe Domänenmodell direkt verwendet
Open Host Service	Die im Domänenmodell definierten Dienste können über eine Interprozess-Schnittstelle integriert werden
Kunde/Lieferant (Customer/Supplier)	Zwar trägt der Lieferant die Verantwortung für die Umsetzung der Anforderungen, aber beide Parteien haben Einfluss auf die Gestaltung des Modells
Publizierte Sprache (Published Language)	Über Service-Schnittstellen oder Dateiaustausch werden gemeinsam genutzte Datenformate verwendet, um Informationen zwischen Domänen auszutauschen
Gemeinsamer Kern (Shared Kernel)	Verschiedene Modelle verwenden einen gemeinsamen Code. Die Abhängigkeiten können dabei von Aufruf- und Benachrichtigungs- bis hin zu Vererbungsbeziehungen reichen
Keine Integration (Separate Ways)	Manchmal erlaubt der fachliche Zusammenhang, Domänenmodelle vollständig zu separieren und auf eine Integration zu verzichten

Integrationsstrategien im DDD

Beispiel-Domäne Rating

Rating-Vorgang

Im Alltag eines Unternehmens – zum Beispiel in einer Bank – gibt es Vorgänge, in denen die Preisgestaltung sowie die Entscheidung über den Ausgang des Vorgangs von dem konkreten Rating-Ergebnis abhängt. Das Rating-Ergebnis beinhaltet einerseits die Ausfallwahrscheinlichkeit und andererseits auch eine Rating-Klasse, die ähnlich einer Ampel signalisiert, wie das Vorhaben des Antragstellers/Kunden einzuschätzen ist.

Im Rahmen des Rating-Vorgangs wird sichergestellt, dass die relevanten Daten und Nachweise des Antragstellers/Kunden zusammengetragen werden, um mit Hilfe des Formelsystems das Rating-Ergebnis berechnen zu können.

Der Rating-Vorgang erlaubt im Rahmen einer Feedback-Schleife das Berechnen von vorläufigen Rating-Ergebnissen. Durch das Anpassen der Datenbasis für die Berechnung in gewissen Parametern können Kundenberater auf ein für den konkreten Fall angemessenes Ergebnis hinwirken.

Portfolio-Zuordnung

Ein Portfolio gruppiert Verträge, Kunden, Pakete mit gemeinsamer Risikoklasse und Anlagen. Ein Portfolio kann in Subsegmente unterteilt werden. Ein Rating-Formelsystem bezieht sich häufig auf ein Portfolio und Subsegment.

Mit der Portfolio-Zuordnung wird ausgewählt, nach welchen Regeln, mit welchen konkreten Formeln und mit welchen erforderlichen Daten das Rating tatsächlich berechnet werden muss.

Der Gestaltungsspielraum des Kundenbetreuers innerhalb eines Rating-Vorgangs hängt auch vom Portfolio (und dem konkreten Subsegment) ab. Beispielsweise besitzen Kundenbetreuer in einer Rückversicherung (Underwriter genannt) oder Berater für VIP-Kunden häufig einen ganz anderen Handlungsspielraum als der gemeine Kundenberater für Retail-Kunden, also Privat- beziehungsweise Kleinkunden.

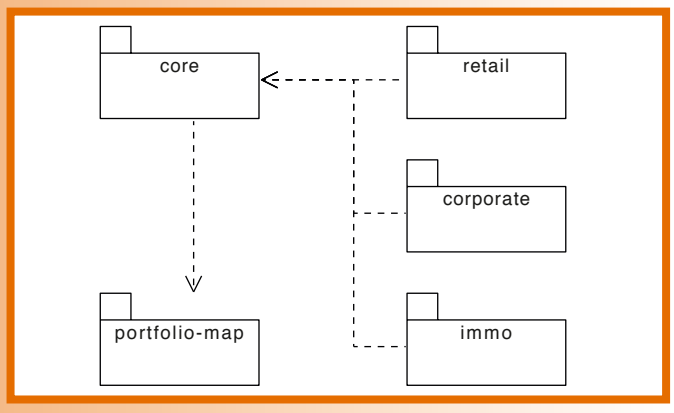

Abb. 1: Überblick: Context-Mapping der Domäne Rating und ihre Subdomänen

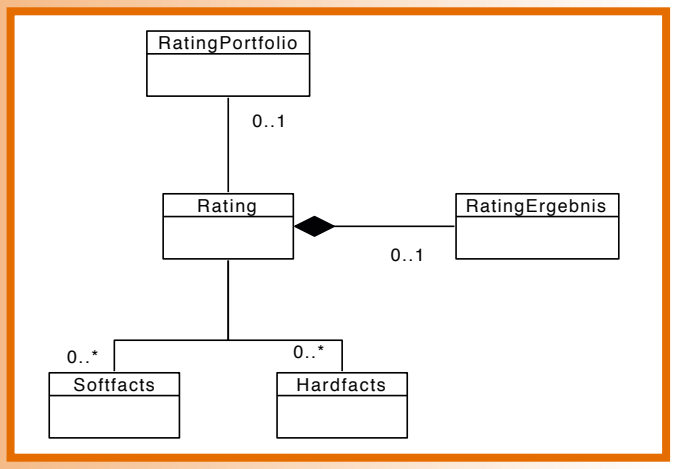

Abb. 2: Domänenmodell der Subdomäne rating-core

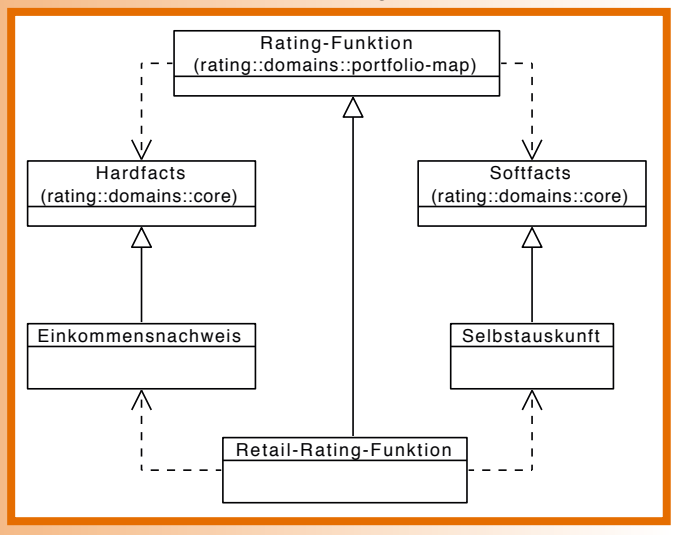

Abb. 3: Domänenmodell der Subdomäne retail

Subdomänen

Wir können die Domäne Rating in verschiedene Subdomänen unterteilen. Für unser Beispiel unterscheiden wir die Subdomänen Retail, Corporate und gewerbliche Immobilienfinanzierung mit jeweils eigenen Anforderungen an die Datenbasis und die Berechnungen (s. Abb. 1).

Bei der Portfolio-Zuordnung muss anhand aussagekräftiger Entscheidungsmerkmale erkannt werden, in welcher Subdomäne die Berechnung für einen aktuellen Vorgang definiert ist.

Strategischer Entwurf der Domäne Rating

Als Integrationsstrategie für die Subdomänen innerhalb der Domäne Rating wählen wir die Shared-Kernel-Strategie: Die Subdomänen teilen sich einen gemeinsamen Kern. Diesen kapseln wir in einem Paket namens core.

Rating-Subdomäne core

Die Subdomäne core repräsentiert die Informationen, die allgemein für die Berechnung eines Rating-Ergebnisses benötigt werden. Das sind typischerweise Hard- und Softfacts. Hardfacts sind belastbare, klar belegbare Daten wie zum Beispiel ein Einkommensnachweis für Angestellte oder betriebswirtschaftliche Auswertungen für Gewerbebetriebe. Softfacts sind eher weichere Einflussfaktoren wie zum Beispiel die Branchenerfahrung, die Einschätzung der Marktsituation oder die Bewertung der Führungs- oder Vertriebskompetenz bei Gewerbetreibenden.

Abbildung 2 zeigt das Modell, mit dem die zu einem Rating-Vorgang gehörigen Daten sowie die Information über die zu verwendende Berechnungsfunktion verwaltet und gespeichert werden können. Diese Daten sollten abstrakt ausreichen, um im Rahmen eines Rating-Vorgangs ein Rating-Ergebnis zu berechnen.

Subdomäne Retail

Die Domäne Retail dient hier als Beispiel für die Portfolio-spezifischen Subdomänen. Retail-Rating erfordert für die meisten Vorgänge einen Einkommensnachweis (als Spezialisierung von Hardfacts) sowie das wahrheitsgemäße Ausfüllen eines Fragebogens (als Spezialisierung der Softfacts). Dies wird in der entsprechenden Subdomäne definiert.

Gemäß DDD stimmen die Techniker und die Fachleute gemeinsam ab, wie die Domäne technisch umgesetzt werden soll. Durch Verzicht auf technische Details können auch Beteiligte des Fachbereichs verstehen, wie sie durch die Geschäftsvorfälle softwaretechnisch unterstützt werden.

Nachdem wir uns über den Zusammenhang zwischen Portfolios, Subsegmenten, Rating-Funktionen, Rating-Vorgängen und den dafür erforderlichen Daten grundsätzlich klar geworden sind, können wir über die konkrete Umsetzung nachdenken. Im Folgenden orientieren wir uns an dem Grobentwurf und vertiefen ihn an der einen oder anderen Stelle sinnvoll, um eine mögliche Umsetzungsstrategie mit Clojure vorzustellen. Wir wenden hier das taktische DDD an.

Umsetzung in Clojure

Rating-Service

Der Rating-Service definiert im Wesentlichen eine Funktion `rate(vorgang, kunde, fakten): RatingErgebnis`. In Clojure ist der Service wie in Listing 1 dargestellt realisiert.

```
(ns ddd-clojure-rating.domain.rating-service
  (:require
    [ddd-clojure-rating.domain.validation :as v]))

(defmulti required-facts (fn [case customer]
  [(:portfolio case) (:segment customer)]))

(defmulti calculate-rating-score (fn [case customer facts]
  [(:portfolio case) (:segment customer)]))

(defn rate [case customer facts]
  (let [req-facts (required-facts case customer)
    missing-facts (v/validate req-facts facts)]
  (if missing-facts
    (throw (IllegalArgumentException. "missing facts"))
    (calculate-rating-score case customer facts))))
```

Listing 1: Rating-Service in Clojure

Zu Beginn erfolgt die Namensraumdefinition sowie der Import des `validation`-Namensraums (in Clojure per `:require`). Die dann folgenden `defmulti`-Ausdrücke definieren für die Portfolio-spezifischen Subdomänen die Schnittstelle, die hier im core/rating-service benötigt wird, um ein Rating-Ergebnis zu berechnen.

In der Funktion `rate` werden in der `let`-Anweisung die benötigten und die fehlenden Daten ermittelt. Sollten Daten fehlen, wirft die Funktion eine `IllegalArgumentException`. Wenn alle erforderlichen Daten vorhanden sind, wird ein transientes Rating-Ergebnis erzeugt und zurückgeliefert.

Case-Aggregat

Das Aggregat (s. Listing 2) wird in seinem eigenen Namensraum als Menge von Funktionen definiert.

```
(ns ddd-clojure-rating.domain.case
  (:require [ddd-clojure-rating.domain.rating-service :as r]
    [schema.core :as s]
    [ddd-clojure-rating.domain.validation :as v]))
```

Listing 2: Namensraumdefinition des Case-Aggregats

Zur Laufzeit wird ein Rating durch eine Map – eine Standard-Datenstruktur in Clojure – repräsentiert. Die Definitionen in Listing 3 beschreiben mit Hilfe der Bibliothek Prismatic Schema [Schema] die zulässigen Strukturen für ein Portfolio und einen Rating-Vorgang in Clojure. Die Funktionen aus Listing 4 dienen der Zustandsmanipulation innerhalb eines Rating-Vorgangs.

```
(def valid-portfolio (s/enum :real-estate :instant-loan))

(def valid-case {:id v/valid-id
  :customer-id v/valid-id
  :credit-amount v/positive-amount
  :portfolio valid-portfolio
  :created long
  (s/optional-key :latest-rating) r/valid-rating})
```

Listing 3: Strukturen für ein Portfolio und einen Rating-Vorgang in Clojure

```
(defn create [id credit-amount portfolio customer-id]
  (s/validate v/valid-id id)
  (s/validate v/positive-amount credit-amount)
  (s/validate valid-portfolio portfolio)
  (s/validate v/valid-id customer-id)
  {:id id
    :credit-amount credit-amount
    :portfolio portfolio
    :customer-id customer-id
    :created (System/currentTimeMillis)})

(defn rate [case customer-provider facts-provider]
  (s/validate valid-case case)
  (let [cid (:customer-id case)
      customer (customer-provider cid)
      facts (facts-provider cid)
      rating-score (r/rate case customer facts)]
    (assoc case :latest-rating
      {:facts facts
       :rating-score rating-score
       :on (System/currentTimeMillis)})))
```

Listing 4: Zustandsmanipulation

Multimethods für Portfolio-spezifische Rating-Funktionen

Der in Listing 1 vorgestellte Rating-Service implementiert den Ablauf eines Rating-Vorgangs, der für alle Vorhaben – unabhängig vom Portfolio – gleich ist. Zu diesem Zweck definiert er die Schnittstelle, die alle Portoflio-spezifischen Rating-Vorgänge erfüllen müssen. Neben den aus der Java-Welt bekannten Mitteln stehen in Clojure zwei weitere Möglichkeiten zur Verfügung: Protocols und Multimethods.

Die Portfolio-spezifischen Funktionen zum Prüfen auf Vollständigkeit der Datenbasis (`required-facts`) und der Berechnung des Rating-Ergebnisses (`calculate-rating-score`) setzen wir hier mit Multimethods um. Wie in Listing 1 zu sehen ist, werden Multimethods mit `defmulti` deklariert.

In der Deklaration einer Multimethod wird direkt eine Dispatch-Funktion angegeben, mit deren Hilfe zur Laufzeit bei jedem Aufruf der Multimethod die konkrete Implementierung ausgewählt wird. Im Unterschied zu gewöhnlicher Polymorphie objektorientierter Sprachen kann eine Dispatch-Funktion alle Funktionsargumente auswerten. In unserem Beispiel wird anhand des Portfolios des Vorgangs und des Marktsegments des Kunden die Implementierung gewählt.

Für jedes Portfolio werden nun mittels `defmethod` die gewünschten Implementierungen vorgenommen. Jede Implementierung der Funktion `required-facts` deklariert die benötigten Hard- und Softfacts in Form einer Map (s. Listing 5). Nach erfolgreicher Prüfung der Datenbasis kann das Rating-Ergebnis einfach berechnet werden (s. Listing 6).

```
(defmethod r/required-facts [:real-estate :retail]
 [case customer]
 {:personal-data c/valid-personal-data
  :property-location {:postal-code Integer}})
```

Listing 5: Erforderliche Hard- und Softfacts

```
(defmethod r/calculate-rating-score [:real-estate :retail]
  [case customer facts]
  let [postal-code (get-in facts [:property-location :postal-code])]
    (if (>= postal-code 60000)
      (r/rating 7 0.5)
      (r/rating 5 0.6))))
```

Listing 6: Berechnung des Rating-Ergebnisses

Anwendungsfälle realisieren

Nun haben wir die Domäne Rating nach DDD modelliert und entsprechend unserer Konstruktionsregeln in Clojure umgesetzt. Es fehlt jetzt nur noch eine Anwendung, die auf der Funktionalität aufsetzt. Die in der Anwendung realisierten Anwendungsfälle verwenden das Domänenmodell. Den Einstiegspunkt bilden die Aggregate. Um ein Rating-Ergebnis zu berechnen, laden wir die zum Rating gehörigen Daten aus dem Repository. Mit Hilfe der Aggregatsfunktionen (`create` und `rate`)

können Ratings (mit `create`) nun konsistent angelegt und mit `rate` berechnet werden. Diese Änderungen werden danach einfach wieder über eine Repository-Funktion persistiert.

Fazit

Die Grundprinzipien des Domain-Driven Designs sind auch in einer funktionalen Programmiersprache anwendbar. Die Verwendung von Standard-Abstraktionen wie Maps, Collections und Sets fördert auf den ersten Blick das Erstellen anämischer Domänenmodelle, bei denen die Aggregate und Entitäten tendenziell nur zum Datentransport taugen und selbst keinerlei Fachlogik kapseln.

In Clojure kann dies durch geeignetes Gruppieren in Namensräumen leicht vermieden werden. Sind die Abbildungsvorschriften einmal bekannt, ist das Finden von zugehörigen Methoden zu Entitäten und Aggregaten einfach.

Literatur und Links

[BED2013] Ph. Schirmacher, Clojure Web Development, BED-Con 2013, `http://bed-con.org/2013/wp-content/uploads/2013/04/WebDevelopmentmitClojure.pdf`

[Evans03] E. Evans, Domain-Driven Design, Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

[Ghad12] Ph. Ghadir, Java-Programme mit Clojure würzen, in: JavaSPEKTRUM, 4/2012

[NeTi10a] B. Neppert, St. Tilkov, Einführung in Clojure – Teil 1: Überblick, in: Java SPEKTRUM, 2/2010, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2010/02/neppert_tilkov_JS_02_10.pdf`

[NeTi10b] B. Neppert, St. Tilkov, Einführung in Clojure – Teil 2, in: Java SPEKTRUM, 3/2010, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2010/03/neppert_tilkov_JS_03_10.pdf`

[NeTi10c] B. Neppert, St. Tilkov, Clojure – Teil 3: Nebenläufigkeit, in: JavaSPEKTRUM, 4/2010, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2010/04/neppert_tilkov_JS_04_10.pdf`

[Schema] Prismatic Schema, `https://github.com/prismatic/schema`

[Vernon13] V. Vernon, Implementing Domain-Driven Design, Addison-Wesley Longman, 2013

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com

Philipp Schirmacher arbeitet als Senior Consultant bei innoQ. Sein Schwerpunkt liegt auf der Entwicklung von Java-Applikationen in agilen Teams. Darüber hinaus interessiert er sich insbesondere für moderne Programmiersprachen wie Clojure und Scala.
E-Mail: philipp.schirmacher@innoq.com