www.javaspektrum.de

Ausgabe 6/2021
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 06

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Besser programmieren –
Neue Tools für Architekten und Entwickler

Von der Idee bis zur Umsetzung – den Big Ball of Mud aufräumen

DDD in der Praxis – Erfahrungen mit einem Open-Source-Tool

Interview

Joachim Schmider, Vice President Enterprise IT Architecture bei der Schaeffler AG erklärt, wie eine neue Art von Enterprise Architecture den Entwicklern die Arbeit erleichtert.

Fachthemen

Kostenfalle Cloud-Datenbanken – ein Reisebericht voller (Fehl)-Entscheidungen

Grundlagen einer agilen Softwareentwicklung in hochdynamischen Umgebungen

Ein Rückblick: 25 Jahre JavaSPEKTRUM

Besser programmieren – Neue Tools für Architekten und Entwickler

G30325

Sonderdruck aus
JavaSPEKTRUM 6/2021

Domain-Driven Design in der Praxis

Erfahrungen mit dem Open-Source-Tool Context Mapper

Stefan Kapferer, Olaf Zimmermann

Domain-Driven Design (DDD) hat sich als eine vielseitige Methode zur Dekomposition von Softwaresystemen und Modellierung von Domänen etabliert. DDD-Modelle werden typischerweise am Whiteboard oder mit Post-it-Notizen skizziert. Nur wenige Software-Tools unterstützen deren Erstellung und Wartung. Die Autoren berichten über ihre Projekterfahrungen mit einem Open-Source-Werkzeug für strategisches und taktisches DDD, ihrem Context Mapper.

DDD fußt auf zwei Kernprinzipien, erstens die Sprache der Fachexperten aufgreifen und durchgängig nutzen sowie zweitens mit diesem Vokabular Domänen-Fachlichkeit und Software-Technik modellieren. Für Nichtinformatiker verständlich kommunizieren wir Softwaremenschen hoffentlich auch ohne DDD, und das Thema könnte einen eigenen Artikel füllen. Also fokussieren wir hier auf den zweiten Punkt, die Modellierung.

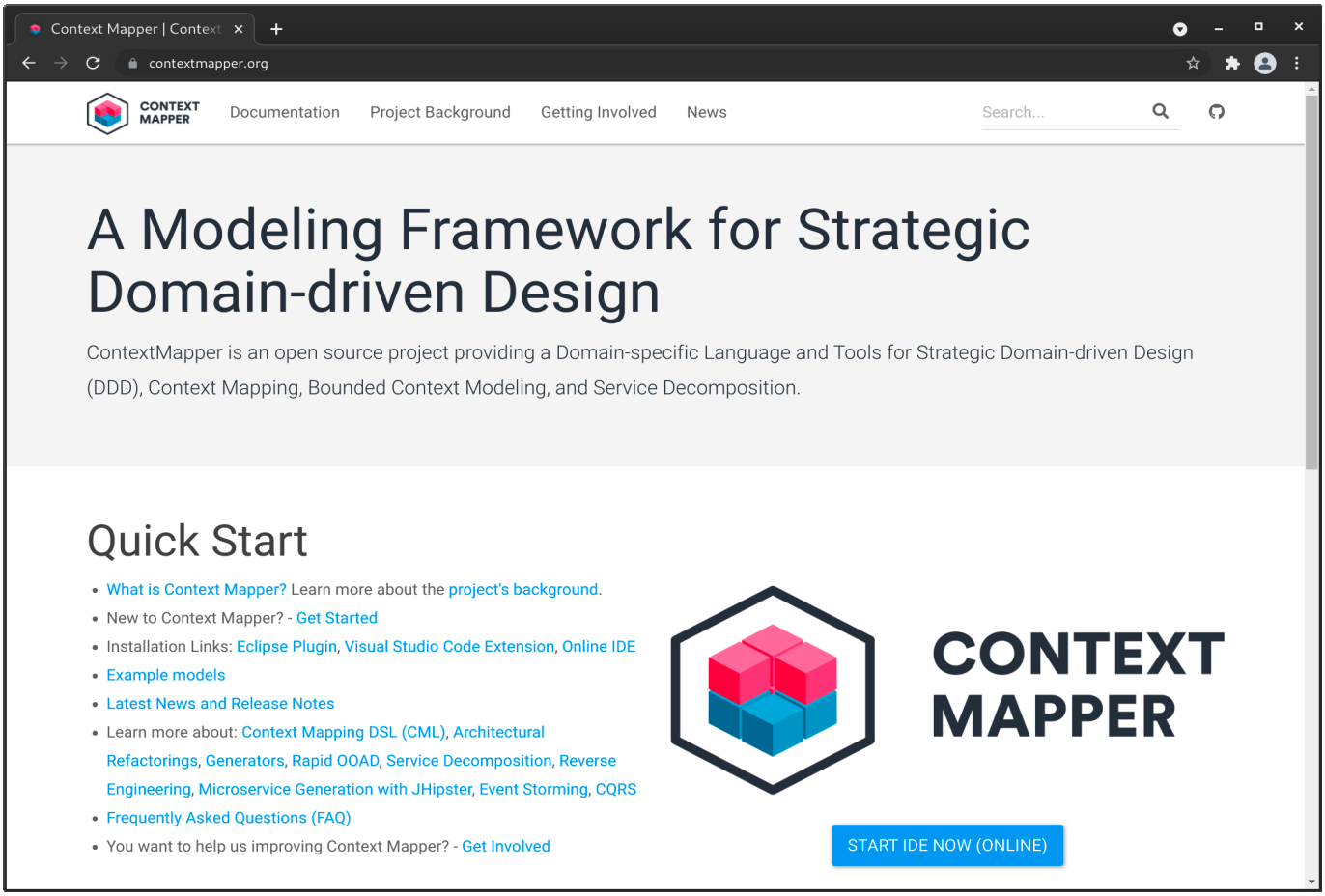
Warum DDD?

Es gibt (mindestens) drei Gründe, warum wir, wie von DDD postuliert, in unseren Projekten gerne und viel modellieren:

- Um aktiv zu lernen und die richtigen Fragen zu stellen (sich und anderen).
- Um Zwischenstände von Analyse und Design zu teilen und gemeinsam zu verbessern, beispielsweise die Arbeitsteilung in Subsystemen oder Integrationsarchitekturen und API-Design.
- Um schlank und angemessen zu dokumentieren für spätere Wartungs- und Evolutionsphasen.

Um Grady Booch zu zitieren: Der Code ist die Wahrheit, aber nicht die einzige Wahrheit.

Taktisches DDD innerhalb einer Anwendung ist unserer Auffassung nach ein sowohl moderner als auch reifer Ansatz für objektorientierte Analyse und Design [Lar04]. Und wir nutzen strategisches DDD als pragmatisches Stadtplanungsinstrument, also für das Unternehmensarchitektur-Management. Es gilt zu verstehen oder festzulegen, welches taktische Design wo und wie weit gilt – und was dabei die Macht- und Einfluss-Beziehungen zwischen den Modellen und ihren Erstellenden sind. Es wird also gar nicht erst versucht, unternehmensweit Einigung über Datenstrukturen, Wertebereiche und fachliche Beziehungen zu erzielen. Bank ist bei-



Stefan Kapferer ist Softwarearchitekt bei der mimacom ag in Zürich. Unter anderem beschäftigt er sich seit einigen Jahren intensiv mit dem Thema DDD. Context Mapper hat seinen Ursprung in seinen Masterstudiumsprojekten (M.Sc. in Informatik) und wird von ihm seither kontinuierlich weiterentwickelt.
E-Mail: stefan.kapferer@mimacom.com

Prof. Dr. Olaf Zimmermann ist Architektur-Berater und Professor an der OST – Ostschweizer Fachhochschule. Als zertifizierter Chief/Lead IT Architect gibt er die Insights-Kolumne in IEEE Software mit heraus. Context Mapper nutzt er seit Version 0.1 und trägt seither zur Entwicklung von DSL und Tools bei.
E-Mail: zio@ozimmer.ch

spielsweise nicht gleich Bank: Eine Privat-, eine Investment- und eine Parkbank unterscheiden sich je nach Anwendungskontext doch erheblich in ihrem/n Verhalten, Eigenschaften und Beziehungen.

Warum haben wir eine werkzeugunterstützte Domain-Specific Language (DSL) für DDD entwickelt und praktizieren DDD nicht einfach nur? Agile Modeling und Event Storming stellen das Whiteboard in den Mittelpunkt. Das ist gut und richtig, aber ist es nicht etwas schade um die dabei entstehenden Erkenntnisse und Ergebnisse? Der Stand der Technik ist „merken", „abfotografieren" oder „bei null losprogrammieren". Das mühsam erarbeitete Domänenverständnis, aber auch die Bedeutung der DDD-Muster, können unserer Erfahrung nach dabei schnell verloren gehen. Ist beispielsweise ein Aggregat einfach nur ein Datentopf? Oder ein Objektcluster mit koordiniertem Verhalten und gemeinsamer Persistenz sowie zugehörigem Konsistenz-, Integritäts- und Invarianten-Management? Was genau unterscheidet die Published Language von der Ubiquitous Language? Und wann wählt man die Rollen Open Host Service plus Conformist statt Customer/Supplier? Wir erklären diese Patterns aus Platzgründen hier nicht, sondern verweisen auf das Buch „Domain-Driven Design Distilled" [Ver16] sowie die Context-Mapper-Webseite und die Hover Helps im Tool selbst.

Mit der DDD-DSL wollen wir schneller und besser modellieren sowie andere Darstellungen und Input für weitere Werkzeuge anforderungs- und kontextgerecht liefern können. Die Präzisierung von Konzepten, die eine DSL mit sich bringt, kann außerdem helfen, eine Domäne noch besser fachlich zu durchdringen und die DDD-Semantik zu klären. Schließlich sehen wir DSL und Werkzeug auch als Ausbildungsmittel für den Unterricht.

```
ContextMap InsuranceContextMap {
  contains CustomerManagementContext
  contains CustomerSelfServiceContext
  contains PrintingContext
  contains PolicyManagementContext
  contains RiskManagementContext

  CustomerSelfServiceContext [D,C]<-[U,S] CustomerManagementContext
  CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext
  PrintingContext [U,OHS,PL]->[D,ACL] PolicyManagementContext
  RiskManagementContext [P]<->[P] PolicyManagementContext
}
```

Listing 1: Context Map in CML geschrieben (vereinfacht)

Das Werkzeug: Context Mapper

Die meisten der heute am Markt verfügbaren Modellierungswerkzeuge sind nicht für das domänengetriebene Design (DDD) ausgelegt, zum Teil teuer oder lernintensiv. Diese Lücke schließt Context Mapper [ContextM]. Dieses Open-Source-Werkzeug (Apache 2) ermöglicht es, Softwaresysteme auf Basis der strategischen und taktischen DDD-Patterns zu modellieren.

Context Mapper bietet eine formale und damit maschinenlesbare Sprache für die DDD „Context Maps" an. Für diese Diagramme, welche die einzelnen „Bounded Contexts" eines Systems darstellen, gab es bislang keine Tools. Mit der *Context Mapper DSL (CML)* können wir Context Maps sowie einzelne Bounded Contexts und deren Beziehungen textuell formulieren, darstellen, und uns grafische Darstellungen automatisch generieren lassen. Das Tool unterstützt aber auch taktische Domänenmodellierung innerhalb von Bounded Contexts. Diese Domänenmodelle werden mittels der taktischen DDD-Patterns ausformuliert (Aggregate, Entitäten, Wert-Objekte usw.).

Abbildung 1 zeigt eine beispielhafte Context Map aus der Versicherungsbranche mit verschiedenen Kontexten dieser Domäne; jeder dieser Kontexte beinhaltet ein in sich geschlossenes taktisches Domänenmodell. Ein derartiges Modell findet sich in der Abbildung als exemplarischer Zoom-in in den „Customer Management Context".

Listing 1 stellt dar, wie eine solche Context Map in CML modelliert werden kann (das Beispiel ist aus Platzgründen vereinfacht). CML unterstützt alle strategischen Patterns und Konzepte, die wir aus dem „Blue Book" [Eva03] und dem „Red Book" [Ver13] kennen: Upstream (U), Downstream (D), Customer (C), Supplier (S), Open-Host-Service (OHS), Published Language (PL), Anticorruption-Layer (ACL), Conformist (CF), Shared Kernel (SK) und Partnership (P).

Listing 2 modelliert einen einzelnen Bounded Context aus. Ein Kontext besteht typischerweise aus einem oder mehreren Aggregaten. Die taktische DDD-Syntax unterstützt die wichtigsten taktischen Patterns wie Entity, Value Object, Domain Event, Ser-

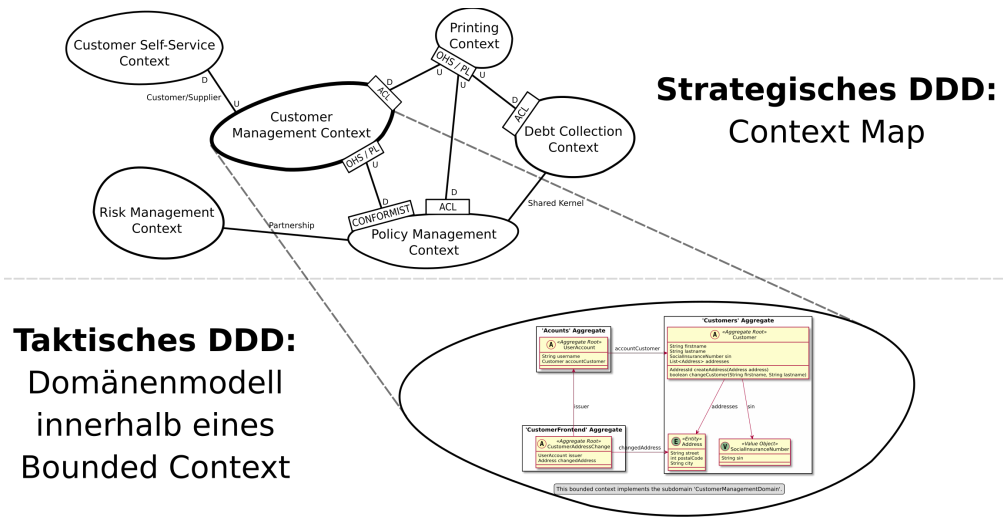


Abb. 1: Strategisches vs. taktisches Domain-Driven Design (DDD)

```
BoundedContext CustomerManagementContext {
  Aggregate Customers {
    Entity Customer {
      aggregateRoot

      - CustomerId id
      String firstname
      String lastname
      - List<Address> addresses
    }

    ValueObject CustomerId {
      String value
    }

    Entity Address {
      String street
      String city
    }
  }
}
```

Listing 2: Beispiel-Domänenmodell (taktisches DDD)

vice und Repository, aus deren Anwendungen sich Aggregate zusammensetzen.

Das Context Mapper Tool ist aktuell für Visual Studio Code (Extension) und Eclipse (Plug-in) verfügbar. Mittels Browser-IDEs wie Gitpod [GitPod] können die Modelle sogar online erstellt werden. Mit verschiedenen Generatoren lassen sich die Modelle in unterschiedliche grafische Repräsentationen transformieren. Beispiele sind grafische Context Maps [Ver13], PlantUML-Komponentendiagramme [PlantUML], UML-Klassendiagramme für die Domänenmodelle der Bounded Contexts sowie noch plattformunabhängige API-Schnittstellenbeschreibungen. Außerdem kommt das Tool mit automatisierten Architektur-Refactorings und einem Command Line Interface (CLI). Die Refactorings und weitere Transformationen ermöglichen es den Benutzern, ihre Modelle agil zu konkretisieren und kontinuierlich mit den Domänenkonzepten weiterzuentwickeln.

Ein Tool zu entwickeln, ist eine Sache, aber funktioniert dieses auch im realen Projektleben?

Projekt 1: Digitalisierungsprojekt in der Print-Branche (eIFU)

Stefan Kapferer hat DDD und Context Mapper kürzlich in einem Digitalisierungsprojekt in der Print-Branche angewendet. Der Kunde, die Paul Büetiger AG aus der Schweiz, ist eine auf sogenannte „Instructions for Use" (IFUs) in der MedTech-Branche spezialisierte Druckerei. Diese Beipackzettel und Gebrauchsanweisungen, oder eben IFUs, dürfen heute in bestimmten Fällen auch digital bereitgestellt werden. Die Büetiger AG möchte eine Plattform aufbauen, die es Kunden erlaubt, IFUs online anzubieten [eIFU]. Der Endkunde kann sich die IFUs auf der Plattform als PDF herunterladen oder mit wenigen Klicks eine gedruckte Version nach Hause bestellen. Die IFUs werden auf der Plattform strukturiert abgelegt und mit branchenspezifischen Metadaten angereichert. Diese erlauben es den Endkunden, IFUs über verschiedene Suchparameter zu finden (Volltext-Indexierung).

In diesem Projekt konnte von der „grünen Wiese" gestartet werden. Bereits in der Anfangsphase, in welcher die Anforderungen mit dem Kunden erarbeitet wurden, sind wir domänengetrieben vorgegangen und haben ein Domänen-Modell erarbeitet. Zu Beginn wurde dieses noch mit einem grafischen UML-Tool erstellt und gewartet.

Bei der Implementierung waren aufgrund der Erfahrungen des Architekten und des Know-hows innerhalb der Organisation „Onion Architecture" und taktisches DDD gesetzt. Die Java-Klassen im Core wurden mit jMolecules-Annotationen [jMolecules] versehen, um den Objekten den Ring der Onion-Architektur [Gra] und das entsprechende taktische DDD-Pattern (Entity, Value Object usw.) zuzuweisen. Erst gegen Ende der Entwicklung des Minimum Viable Product (MVP) haben wir aus zwei Gründen begonnen, das Modell mit Context Mapper im Git-Repository zu pflegen.

Der erste Grund war die Dokumentation. Die Context Mapper-Generatoren, vor allem der PlantUML-Generator, wurden in die Architekturdokumentation, welche in AsciiDoc geschrieben wird, integriert. Dadurch mussten die Entwickler und Architekten keine grafischen Modelle mehr nachpflegen; eine Änderung des CML-Modells führte automatisch (über die CI/CD-Pipeline) zu einer neuen Version des Architekturdokuments.

Der zweite Grund: Ein grafisches Domänenmodell, welches manuell gepflegt werden muss, und der Code laufen typischerweise schnell auseinander. Im Laufe des Projekts werden Änderungen vorgenommen, welche es nur in den Code schaffen, nicht aber in das grafische Modell und die Dokumentation. Mit einer neuen ArchUnit-Erweiterung haben wir erreicht, dass das DDD-Modell jederzeit den Code widerspiegelt: Fügt ein Entwickler beispielsweise im Code eine neue Entität oder ein neues Value Object ein, das im Modell nicht vorhanden ist, schlägt ein ArchUnit-Test fehl. Erweiterungen im Code führen zwingend auch zu einem Update des Modells. Dies wiederum initiiert automatisch eine Aktualisierung der Dokumentation.

In diesem Projekt wurde Context Mapper vorwiegend vom Architekten verwendet. Ziel ist, dass zukünftig auch Requirements Engineers, die etwas weiter weg von der Technik operieren, das Modell pflegen können. Mit diesem Ansatz und dem Einsatz des Tools stellen wir sicher, dass der Code mit den Vorstellungen des Business übereinstimmt und dieselbe Terminologie verwendet wird.

Projekt 2: Innovationsprojekt „Wearable to Wallet" (FinTech-Banche)

Fachlich geht es in diesem Szenario um ein „Fulfillment-as-a-Service" im Bereich von „Wearables" als Zahlungsmittel, also zum Beispiel Uhrenarmbänder, die wie Kreditkarten eingesetzt werden können. Im Anwendungskontext finden sich Kartenherausgeber, Banken und Wearables-Hersteller. Komplexe Abläufe in Hard- und Software sind zu bewältigen; Menschen sind als Dienstleister in teilautomatisierte Prozesse in Lagerhaltung, Bestellungsdisposition und Lieferlogistik eingebunden.

Das Geschäftsmodell des Projektpartners wearonize AG [wearonize], eines FinTech-Start-ups, entwickelt sich zurzeit sehr dynamisch, auch mithilfe von Simulationen. Die Geschäftsprozesse und ihre Software- und API-Unterstützung sollen schnell änderbar gestaltet werden, um auf Simulationserkenntnisse und Marktereignisse, wie zum Beispiel Kundenfeedback oder Konkurrenzverhalten, reagieren zu können. Nach ersten Markteintrittserfolgen wird eine Delivery Center Software für den Fulfillment Service produktiv in einer Public Cloud betrieben; sie darf somit bereits als Legacy-System bezeichnet werden [Fea]. Ein „Third-Party API" ist integriert; diese Programmierschnittstelle ist nicht immer für alle Team-Mitglieder in Entwicklung und Test vollumfänglich verfügbar. Ein eigenes DevOps-Team betreut das Delivery Center; kleinere Änderungen in Daten und Abläufen sollen nicht jedes Mal zu aufwendigen Entwicklungsvorhaben führen.

In diesem Brownfield-Szenario wurden fachliche Abläufe ereignisgetrieben modelliert sowie Schnittstellen und Domänenmodelle aus JSON-Schemata und Java-Klassen zurückgewonnen (und dann wiederum für Rapid Prototyping und Testen von APIs verwendet). Ein Nebenziel war es, die Architektur- und Schnittstellen-Do-

kumentation zu verbessern. DDD und Event Storming dienten zunächst der Einarbeitung, dann auch der gemeinsamen Kreativarbeit. Da Java und Spring Boot bereits im Einsatz sind, wurde für das Prototyping JHipster [JHip] ausgewählt. Olaf Zimmermann agierte als Coach, Moderator und Coding-Architekt in Workshops mit dem Product Owner und weiteren Domänen-Experten.

In wenigen Minuten entstand eine erste laufende Anwendung, in wenigen Stunden ein mittel-komplexes Domänenmodell mit UML-Dokumentation, zwei abstrakte, aber realistische Geschäftsprozessmodelle und eine API-Spezifikation (18 Operationen mit geschachtelten JSON-Nachrichtenschemata). Es konnte eine vollständige Abdeckung der Implementierung im DDD-Modell mit angemessener Modellierungstiefe (einem Kompromiss zwischen abstrakter Flughöhe und Detailtreue) erreicht werden. Das generierte UML wurde als nützlich eingestuft – mit dem Wunsch, zukünftig auch Use-Case-Modelle zu generieren.

Als Fazit aus diesem Innovationsprojekt lässt sich sagen, dass Event Storming und DDD sich positiv auf Produktivität und Qualität auswirkten; es konnte schnell und effektiv iteriert werden. Context Mapper stellte die Stimmigkeit und fachliche Tiefe der DDD-Modelle sicher. Es gab nur wenige kleine Change Requests für DSL und Tools, beide ja schon seit 2018 öffentlich verfügbar und seither kontinuierlich erweitert und verbessert. Eine Schwierigkeit war das nur eingeschränkt mögliche Model-to-Code Round Tripping im Zusammenspiel mit JHipster.

Erfahrungen mit DDD-Praktiken, DSLs und Tool

Wir erlebten einige DDD-Pattern-Namen als etwas zu esoterisch oder schwammig. Beispielsweise wird die Bedeutung von „Open" in OHS und „Published" in PL nicht sofort klar; es gibt mehrere mögliche Interpretationen. Auch „Aggregate" ist zwar expressiv und korrekt, aber nicht unbedingt ein gut zugänglicher und intuitiver Name (Chemiker und Elektrotechnikern verstehen beispielsweise etwas ganz anderes darunter). Der Nutzen von DDD, insbesondere die im Context Mapper unterstützten strategischen und taktischen Patterns, ist unserer Meinung nach aber unstrittig. Die Applikationsschicht (also z. B. Prozesse, Zustände, Vor- und Nachbedingungen) ist dabei eventuell noch unterrepräsentiert. Dies kann von den DDD-Schöpfern so gewollt sein – ein Modell darf, ja muss sogar weglassen. Es bedeutet aber auch, dass anderswo weiter entworfen werden muss (eine Möglichkeit dafür sind Sprachen und Werkzeuge für das Workflow-Management).

Schließlich konnten wir beobachten, dass Business-Fachexperten in Event Stormings nicht immer Stimulus und Response (bzw. Command und Event) unterscheiden können oder wollen. Sie sagen dann einfach, was wann passieren soll; die Ausarbeitung in Event-Command-Paare erfolgt dann im Rahmen der Ergebnissicherung oder einem Folgeworkshop.

Text-DSLs wie CML sind gegenüber Bildern im Vorteil bei der Modell-Erstellung. Diese geht schnell von der Hand, man fokussiert auf Inhalte statt Layout. Grafische Darstellungen werden aber unserer Meinung nach weiter einen prominenten Platz im Werkzeugkoffer von „Reflective Practitioners" haben – denn ein Bild sagt bekanntlich mehr als 1000 Worte. Unser Context Mapper Tool zeigte sich stabil, gut benutzbar und nutzenstiftend; einige zusätzliche Quick Fixes für schnelles Forward Engineering und weitere Discoverer können zukünftig die Arbeit mit dem Werkzeug noch angenehmer und effizienter machen.

Die von uns von einem anderen Open-Source-Projekt „geerbte" T-DDD-Syntax ist manchmal etwas unintuitiv, aber auch sehr ausdrucksstark. Die zahlreichen Beispiele und konstruktiven Fehlermeldungen im Context Mapper helfen beim Einstieg. Einige der T-DDD-Sprachkonstrukte wirken auf Fachexperten und Requirements Engineers, die im Alltag nicht programmieren, recht technisch und (fast schon zu) ähnlich zu Programmcode.

Sowohl bei der Weiterentwicklung von Sprache und Werkzeug als auch im Projekteinsatz gilt es, eine Balance zu finden zwischen Genauigkeit und Einfachheit von Modellen und Sprachkonzepten. Häufig reicht ein Modell alleine nicht aus, um die ganze Breite der Fachlichkeit zu erfassen. Requirements Engineers beispielsweise beschreiben bestimmte Aspekte, wie zum Beispiel fachliche Regeln, Abläufe oder Begründungen für Designentscheidungen, dann gerne zusätzlich in einfachem Fließtext. Es bleibt offen, wie sich diese beiden Welten noch besser integrieren lassen („hybrides Domänen-Engineering").

Unser Fazit: DDD ist eher „underrated" als „overrated"[1]. Es hat großes Anwendungs-, aber auch noch Forschungs- und Entwicklungspotenzial [ZiSt]. Context Mapper ist mittlerweile zu einem Domain Modeller, Architecture Validator (ArchUnit Extension [GitH] und jQAssistant-Plug-in [jQAss] verfügbar) und Integration Flow Designer herangereift. Mit unserem DDD-Werkzeug hoffen wir, die Zugänglichkeit zentraler Software-Engineering-Themen wie gemeinsame, fachlich orientierte Sprache und Modellierung weiter zu erhöhen und dabei den zugehörigen Nutzen der DDD-Konzepte aufzeigen zu können.

Literatur und Links

[ContextM] Context Mapper – A Modeling Framework for Strategic Domain-driven Design, https://contextmapper.org/
[eIFU] eIFU - Ihre Gebrauchsanweisung. Aber elektronisch, Paul Büetiger AG, https://www.buetiger.ch/kernkompetenz/e-ifu
[Eva03] E. Evans, Domain-Driven Design – Tackling Complexity in the Heart of Software, Addison-Wesley, 2003
[Fea] M. Feathers, Looking Back at Working Effectively with Legacy Code, 15.3.2021, https://www.infoq.com/podcasts/working-effectively-legacy-code/
[GitH] Context Mapper ArchUnit Extension, https://github.com/ContextMapper/context-mapper-archunit-extension
[GitPod] Gitpod - Always Ready to Code, https://www.gitpod.io/
[Gra] H. Graça, DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together, 16.11.2017, https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/
[JHip] JHipster, https://www.jhipster.tech/
[jMolecules] Architectural abstractions for Java, https://github.com/xmolecules/jmolecules
[jQAss] jQAssistant Context Mapper Integration, https://jqassistant.org/jqassistant-context-mapper-integration-released
[Lar04] C. Larman, Applying UML and Patterns, Prentice Hall, 3. Aufl., 2004
[PlantUML] PlantUML, https://plantuml.com/
[Til] St. Tilkov, Is Domain-driven Design overrated?, 2.3.2021, /https://www.innoq.com/en/blog/is-domain-driven-design-overrated/
[Ver13] V. Vernon, Implementing Domain-Driven Design, Addison-Wesley, 2013
[Ver16] V. Vernon, Domain-Driven Design Distilled, Addison-Wesley, 2016
[wearonize] https://www.wearonize.com/
[ZiSt] Olaf Zimmermann und Mirko Stocker, Design Practice Reference, eBook, https://leanpub.com/dpr

1 um einen Blog-Post von Stefan Tilkov [Til] zu zitieren, der dann aber eine ähnliche Botschaft sendet wie wir