Modernization Work Ahead

Den Big Ball of Mud mit Strategic DDD und Software Analytics aufräumen

Stephan Pirnbaum

Aus dem einst überschaubaren System ist ein unüberblickbarer Big Ball of Mud geworden. Die Entwickelbarkeit verschlechtert sich kontinuierlich, Fehler in Produktion häufen sich und Deadlines können immer seltener eingehalten werden. Um dem zu begegnen, muss die Anwendung grundlegend strukturell modernisiert werden. Doch wie? Da die Lösung nicht mit der Umsetzung beginnt, skizziert dieser Artikel den Weg von der Idee bis zur Umsetzung und stellt hilfreiche, erprobte Methoden und Werkzeuge vor.

Jedes Softwareprojekt kann aus unterschiedlichen Perspektiven betrachtet werden. Auf der einen Seite stehen Entwickler und Architekten, die Code-nah arbeiten und die damit verbundenen Probleme Tag für Tag schmerzlich spüren. Auf der anderen Seite stehen für Projektleitung und Management nicht sichtbare strukturelle Probleme und ihre negativen Auswirkungen auf das Projekt.

Nicht eingehaltene und nicht existierende Architektur- und Implementierungsvorgaben sind typische Ursachen, wenn Entwickler erst während der Implementierung auf schlecht erweiterbaren Code stoßen. Doch wie kann man als Entwickler damit umgehen? Für Teilbereiche ist es möglich, die technische Schuld direkt abzubauen. Besonders aber bei Querschnittsaspekten, dazu gehören Exception Handling und Security, oder aber bei der technischen und insbesondere fachlichen Strukturierung (Stichwort: DDD) ist dies nicht ohne Weiteres möglich. Für das Projekt bedeutet es im besten Fall nur eine leichte Zeitverzögerung oder gar die Nichtumsetzbarkeit von neuen Features.

Wie bekommt man Zeit von der Projektleitung eingeräumt, um notwendige Umbaumaßnahmen ganzheitlich und nachhaltig umzusetzen? Als Erstes ist es wichtig, die folgenden drei Fragen zu beantworten:

- Welche Probleme sind bereits entstanden und welche potenziellen Risiken existieren?
- Was sind mögliche Lösungen und welche Vorteile sowie Umsetzungsrisiken bringen diese mit?
- Wie viel Aufwand entsteht, um die Änderungen umzusetzen, und welche Auswirkung hat dies auf die sonstige Entwicklung?

Dabei stellen sich für die Projektleitung und das Management folgende Fragen:

- Welche Kosten entstehen kurzfristig durch einen nachhaltigen Umbau des Systems?
- Welcher Nutzen entsteht langfristig durch die Änderungen?
- Welche Risiken entstehen bei einem „weiter so“?

Stephan Pirnbaum ist Consultant bei der BUSCHMAIS GbR. Er beschäftigt sich leidenschaftlich gern mit der Analyse und strukturellen Verbesserung von Softwaresystemen im Java-Umfeld.
E-Mail:
stephan.pirnbaum@buschmais.com

Vom Brownfield ins Cleanfield

Wenn das Brownfield-Projekt wieder aufblühen soll, muss das „große Ganze" wieder in den Fokus gerückt werden. Ein Mikromanagement technischer Schulden ist an der Stelle nicht sinnvoll, da dafür die strukturelle Basis fehlt und erst grundlegende architektonische Probleme gelöst werden müssen.

Dieser Prozess ist vergleichbar mit dem Start eines Projekts auf der grünen Wiese. Auch hier ist es unerlässlich zu definieren, welche fachlichen, technischen und geschäftlichen Anforderungen mit der Architektur umsetzbar sein müssen. Aus den definierten Zielen lassen sich anschließend die Zielarchitektur und notwendige Änderungen ableiten. Durch regelmäßige Architekturarbeit wird es möglich, wohlstrukturierte, wart- und erweiterbare Systeme jenseits der grünen Wiese dauerhaft fortzuführen. Die einzelnen Strukturebenen nach ihrer Bedeutung sind in Abbildung 1 dargestellt.

Der Startpunkt zur Analyse der Qualitätsziele für die technische Architektur stellt die ISO/IEC 9126 [ISO] dar, in welcher mögliche Qualitätsziele in acht Kategorien wie zum Beispiel Funktionalität, Wartbarkeit und Sicherheit eingeteilt werden. Beachtet werden muss hier, dass die Funktionalität die technischen Qualitätsziele maßgeblich bestimmt.

Kenne deine Ziele

Bei all den Möglichkeiten, etwas zu verbessern, ist es immer wichtig, die Ziele zu kennen. Und nein: Hier geht es nicht um die Minimierung von SonarQube-Issues. Es geht darum, zu wissen, wie sich die Zukunft des Systems gestaltet.

Ein System, welches zeitnah abgelöst werden soll, muss nicht noch aufwendig umgebaut werden. Vielmehr müssen bestehende Funktionalitäten gesichert und das Finden und Beheben von Fehlern vereinfacht werden, um die Restlaufzeit des Systems erfolgreich über die Bühne zu bringen. Soll hingegen funktionelles Wachstum erfolgen und eine Grundlage für die Entwicklungen der kommenden Jahre geschaffen werden, schaut es schon anders aus.

Zugegeben, Entwickler werden die Entscheidung nicht allein treffen. Es handelt sich hier um eine Mischung aus Geschäftsvision und sinnvoller technischer Umsetzung. Jedoch zeigt die Erfahrung, dass Entwickler, die eine Vision für das Projekt haben, selbst am besten einschätzen können, welche Änderungen notwendig sind. Transparenz ist an dieser Stelle wichtig und kann zum Beispiel mit einem Business Model Canvas geschaffen werden. Schon dessen Erstellung im Team erzeugt neue Einblicke und räumt Unklarheiten aus dem Weg.

So viel zu tun ...

Nachdem geprüft wurde, ob ein System überhaupt noch große Änderungen erfahren sollte, kommt nun die Frage: Was zuerst? Softwaresysteme sind komplexe Wesen und so ist es nicht verwunderlich, dass diese sehr vielschichtige Probleme aufweisen. Um sinnvoll entscheiden zu können, wo zuerst Änderungen vorgenommen werden sollen, müssen folgende Aspekte beachtet werden:

- Identifizierung der entsprechenden Komponenten im System,
- Überprüfung der relevanten Probleme und deren Dringlichkeit.

Softwaresysteme, selbst wenn sie einem monolithischem Big Ball of Mud gleichen, bestehen aus technisch und fachlich motivierten Komponenten, wie zum Beispiel Security-Querschnittskomponenten oder Bounded Contexts. Jede dieser Komponenten stellt eine potenzielle Baustelle dar, an der Änderungen umgesetzt werden können.

Um hier eine Reihenfolge festzulegen, ist zu betrachten, wie die Komponenten hinsichtlich der Erreichung der Geschäftsvision zu priorisieren sind. Dafür eignen sich Core Domain Charts [CDC] ausgesprochen gut, da hierbei die einzelnen Bereiche in die Kategorien „Core", „Supporting" und „Generic" eingeteilt werden. Dabei wird typischerweise auch berücksichtigt, wie sich die Zuordnungen über die Zeit verändern sollen, wenn die Geschäftsvision dies vorgibt.

Abb. 1: Die Ebenen strukturierter Projekte

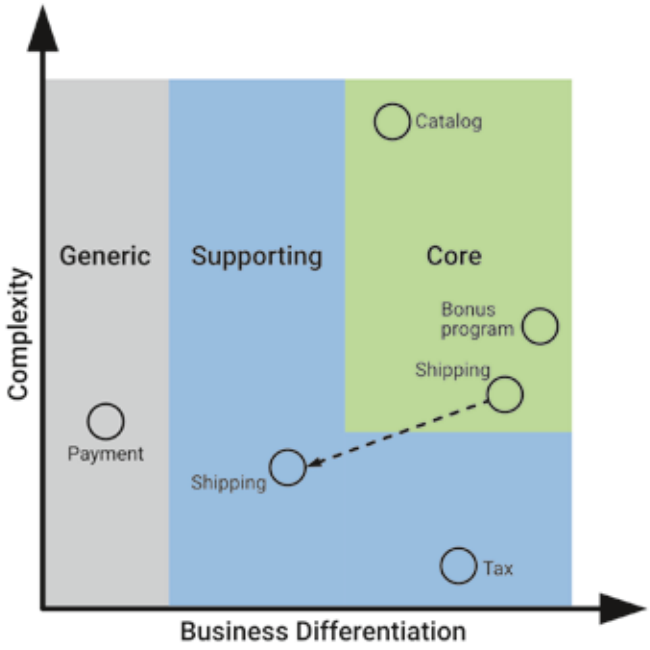

Abb. 2: Core Domain Charts für ein Shop-System

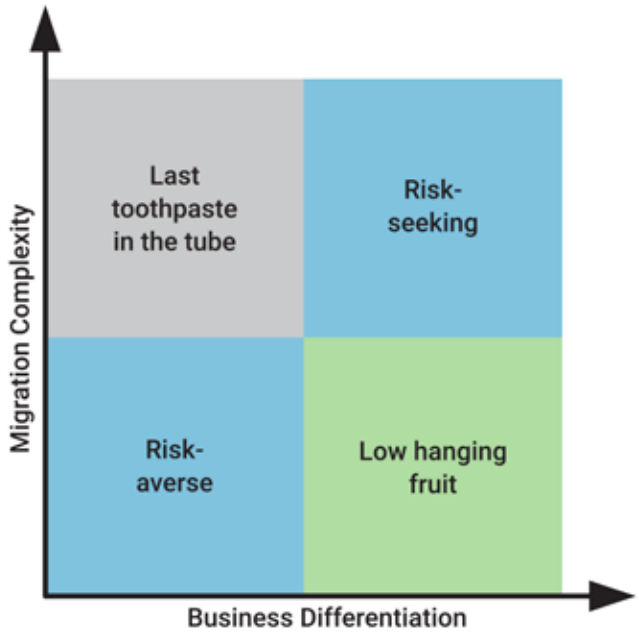

Abb. 3: Core Domain Charts zur Priorisierung

Abbildung 2 zeigt die Einteilung nach Komplexität und Produktdifferenzierung. Daraus lässt sich direkt eine Priorisierung ableiten. Bereiche, durch die sich das Unternehmen von Konkurrenten absetzen kann und die Teil der Kern-Domäne („Core") sind, erfordern eine höhere Investitionsbereitschaft als generische Domänen, für die Lösungen zugekauft werden können.

Die zweite Stufe der Priorisierung bezieht sich auf die Probleme innerhalb der jeweiligen Komponenten. Zum einen, was produktionskritisch ist und vielleicht schon Kosten verursacht hat. Zum anderen, was einfach zu lösen ist und einen großen Benefit bringt. Auch hier helfen uns wieder die Core Domain Charts, diesmal in einer angepassten Variante (s. Abb. 3). Je höher der Nutzen und je geringer der Aufwand, desto früher sollte die Umsetzung beginnen. Im Unterschied zu Abbildung 2 werden nicht mehr die Komponenten, sondern die konkreten Probleme eingeteilt.

Dabei ist es möglich, je Komponente ein weiteres Core Domain Chart zu erstellen oder aber ein gemeinsames Chart zu nutzen und die farblich unterschiedlich zu codieren. Letzteres hat den Vorteil, dass die gesamte anstehende Arbeit auf einen Blick erkennbar ist und eine Priorisierung leichter erfolgen kann. Nachteilig hingegen kann sein, dass es mit zunehmender Größe der Charts unübersichtlich wird.

Die erarbeitete Priorisierung wird natürlich nicht auf alle Ewigkeit Bestand haben, da sich durch Änderungen in den Anforderungen auch die Prioritäten verschieben. Ein regelmäßiges Review der Priorisierung, im besten Fall immer nach der Umsetzung einer Modernisierung, ist daher zwingend notwendig.

Und was jetzt?

Wenn die Priorisierung initial steht, kann es doch losgehen, oder? Fast! Natürlich muss auch die Umsetzung geplant werden. Das beinhaltet die Abschätzung von Aufwand und Risiko, die Schritte zur Umsetzung und besonders die Form der Dokumentation und Absicherung der Änderungen vor erneuter Erosion.

Also, wie zeichnet sich eine erfolgreiche Umsetzung aus? Die Erfahrung zeigt, dass vier Eigenschaften der Schlüssel zum Erfolg sind:

- kleinschrittig,
- planvoll,
- nachhaltig und
- iterativ.

Ein kleinschrittiges Vorgehen hilft, Änderungen häufig zu integrieren und so Teilschritte abschließen zu können. Damit wird vermieden, dass „endlose" Tickets entstehen. Voraussetzung dafür ist ein planvolles Vorgehen. Getreu dem Motto „Proper Preparation Prevents Poor Performance" (James Baker) hilft eine gezielte Vorbereitung, die Umsetzung effizient zu gestalten und in Summe weniger Aufwand als ohne Vorbereitung zu haben.

Wird Aufwand spendiert, um strukturelle Verbesserungen umzusetzen, ist es natürlich wünschenswert, diese auch zu konservieren und eine erneute Erosion zu vermeiden. Sprich, es sollte nachhaltig sein. Schlussendlich sollten Probleme nacheinander iterativ angegangen werden, um die Effektivität bei der Umsetzung sicherzustellen.

Ein möglicher Prozess zur Umsetzung einzelner Iterationen ist der AISE-Prozess [AISE], der sich aus einer Vielzahl von Kundenprojekten entwickelt hat. Zunächst entworfen, um die Migration vom Monolithen zu Microservices umzusetzen, spielt dieser auch in sonstigen Modernisierungsprojekten seine Stärken aus (s. Abb. 4).

Mit AIS(E) in die Zukunft

Der AISE-Prozess ist ein iterativer, vierstufiger Prozess, wobei der vierte Schritt nur im Rahmen der Transformation hin zu Microservices angewendet wird. Die ersten drei Schritte „Analyse", „Isolation" und „Absicherung" (Secure) implementieren die zwei Vor-

Abb. 4: Der AISE-Prozess als Grundlage zur Softwaremodernisierung

aussetzungen „planvoll“ und „nachhaltig“. Die Kernidee ist, dass die Umsetzung fest von Vorbereitung sowie Dokumentation und Absicherung begleitet wird, um so effizienter die Änderungen am System vornehmen zu können. Zusätzlich werden die Lieferung von Features aufgrund kürzerer Umsetzungen weniger eingeschränkt und erreichte Ergebnisse vor erneuter Erosion gesichert.

Die Analyse-Phase vermittelt einen Eindruck vom Zustand des Softwaresystems, um das Bauchgefühl bei der Schätzung von Aufwänden und Risiken gegen belastbares Wissen einzutauschen. Die Modernisierung wird am folgenden Beispiel illustriert.

Ein existierendes Shop-System soll intern nach fachlichen Domänen strukturiert werden, sodass diese zwar nicht separat ausrollbar, aber unabhängig voneinander entwickelbar sind. Die im Artikel beschriebenen Schritte könnten dazu führen, dass ein fachlicher Kontext, zum Beispiel der Produktkatalog, separiert werden soll. Zur Umsetzung stellen sich nun unter anderem die folgenden Fragen:

- Welche Code-Bestandteile des Systems sind zugehörig zu dem Bounded Context?
- Welche Abhängigkeiten existieren zu und von diesem Bounded Context?
- An welchen Stellen müssen Schnittstellen eingeführt oder überarbeitet werden?

Entscheidend ist es, diese Fragen vor der Umsetzung so präzise wie möglich zu beantworten, um die Aufwände verlässlich abschätzen zu können.

Mit der guten Vorbereitung ist die Umsetzung, bei AISE „Isolation“ genannt, mit vergleichsweise wenigen Überraschungen umsetzbar. Unerlässlich ist es, die Veränderungen im Anschluss zu dokumentieren und gegen erneute Erosion abzusichern. Das ist Bestandteil des Schritts „Absicherung“. Ein guter Input ist dafür das Ergebnis der Analyse.

Ich hab‘ ein Problem – ich brauch‘ ein Tool

Nun sind noch zwei Fragen ungeklärt:

- Wie können Aufwand und Risiko abgeschätzt werden, ohne auf das Bauchgefühl vertrauen zu müssen?
- Wie werden die Änderungen vor erneuter Erosion abgesichert?

Für beides ist das Open-Source-Tool jQAssistant [jQA] die beste Wahl. Mit jQAssistant lassen sich Strukturen in Java-Softwaresysteme analysieren, indem sie in eine Neo4j-Graphdatenbank eingelesen werden.

Reichert man im Graphen beispielsweise die Soll-Architektur an, in unserem Beispiel die identifizierten Bounded Contexts und deren erlaubte Abhängigkeiten, kann gegen diese kontinuierlich der Ist-Zustand geprüft werden. Dadurch ist es einfach möglich, Abweichungen zu identifizieren und unsere zur Aufwandsschätzung gestellten Fragen zu beantworten. Listing 1 zeigt eine mögliche Abfrage in der Sprache Cypher, welche die definierten Bounded Contexts und deren erlaubte Abhängigkeiten, also die Soll-Architektur, zurückgibt.

```
MATCH
  (b1:BoundedContext)
OPTIONAL MATCH
  (b1)-[d:DEFINES_DEPENDENCY]->
    (b2:BoundedContext)
RETURN
  b1, d, b2
```

Listing 1: Abfrage der Soll-Architektur

```
ContextMap ShopToBeMap {
  state = TO_BE
  contains CatalogContext
  contains OrderContext
  CatalogContext [U]->[D] OrderContext
}
BoundedContext CatalogContext
BoundedContext OrderContext
```

Listing 2: Definition der Soll-Architektur

```
@BoundedContext(name="OrderContext")
package com.example.shop.order;
import org.jmolecules.ddd.annotation.
  BoundedContext;
```

Listing 3: Annotation von Bounded Contexts mit jMolecules

```
= Documentation

[[ddd:Default]]
[role=group,includesConcepts="ddd:*",includesConstraints="ddd:*"]
== jQAssistant Examples

[[ddd:ToBeContextMapReport]]
[source,cypher,role=concept,reportType="context-mapper-diagram"]
.Defined Bounded Contexts and Dependencies.
----
MATCH
  (b1:ContextMapper:BoundedContext)
OPTIONAL MATCH
  (b1)-[d:DEFINES_DEPENDENCY]->
    (b2:ContextMapper:BoundedContext)
RETURN
  b1, d, b2
----

[[ddd:UndefinedDependency]]
[source,cypher,role=constraint,requiresConcepts="jmolecules-ddd:*"]
.Report of unallowed dependencies.
----
MATCH
  (c1:ContextMapper:BoundedContext),
  (c2:ContextMapper:BoundedContext),
  (b1:JMolecules:BoundedContext{name: c1.name}),
  (b2:JMolecules:BoundedContext{name: c2.name}),
  (b1)-[:DEPENDS_ON]->(b2)
WHERE
  NOT (c1)-[:DEFINES_DEPENDENCY]->(c2)
RETURN
  b1 AS Source, b2 AS Target
----
```

Listing 4: AsciiDoc-Datei

Zur Analyse werden entweder der von der Datenbank mitgelieferte Browser oder Jupyter Notebooks genutzt. Mit Letzteren ist es besonders einfach möglich, Software Analytics strukturiert und reproduzierbar umzusetzen und Ergebnisse effizient zu dokumentieren. Unter [SAS] finden sich Beispiele, die direkt im Web-Browser nachvollzogen werden können. Eine ausführliche Anleitung ist dafür in der README hinterlegt. Die Herausforderung ist, dass jede Analyse natürlich eine gewisse Unschärfe aufweist. Erfahrungen zeigen aber, dass die mit Software Analytics begründeten Schätzungen eine hinreichende Genauigkeit liefern.

Nach der Umsetzung, im Beispiel dem Herauslösen des Produktkatalogs in separate Package- oder Modulstrukturen, folgen die Absicherung und Dokumentation der Strukturen. Ziel ist es, diese Schritte zu automatisieren, um einen kontinuierlichen Abgleich der Ist- gegen die Soll-Architektur zu ermöglichen. Hier wird häufig von lebendiger Dokumentation gesprochen, also Dokumenta-

```
<build> <plugins> <plugin>
  <groupId>com.buschmais.jqassistant</groupId>
  <artifactId>jqassistant-maven-plugin</artifactId>
  <executions> <execution>
  <configuration>
    <scanIncludes> <scanInclude>
    <path>
      ${project.basedir}/jqassistant/contextmapper
    </path> </scanInclude> </scanIncludes>
    <groups>
      <group>ddd:Default</group>
    </groups>
  </configuration> </execution> </executions>
  <dependencies>
    <dependency>
      <groupId>
        org.jqassistant.contrib.plugin
      </groupId>
      <artifactId>
        jqassistant-jmolecules-plugin
      </artifactId>
    </dependency>
    <dependency>
      <groupId>
        org.jqassistant.contrib.plugin
      </groupId>
      <artifactId>
        jqassistant-context-mapper-plugin
      </artifactId>
    </dependency>
  </dependencies>
  </plugin> </plugins>
</build>
```

Listing 5: Integration von jQAssistant

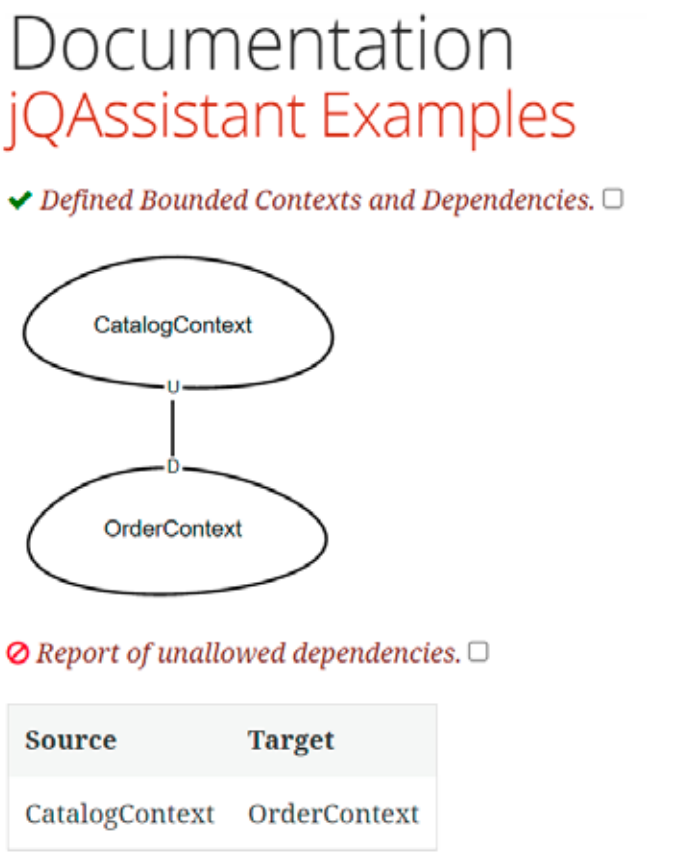

Abb. 5: Gerendertes Ergebnis der AsciiDoc-Dokumentation

tion, die sich auf Basis der Implementierung kontinuierlich selbst aktualisiert.

Der Gedanke ist, neben textueller Beschreibung auch Abfragen wie oben gezeigt in der mit AsciiDoc erstellten Dokumentation zu definieren. Diese Abfragen werden während des Builds von jQAssistant automatisch ausgeführt und die Ergebnisse tabellarisch oder grafisch dargestellt. Neben der lebendigen Dokumentation ermöglicht dieses Vorgehen auch, Architekturverletzungen frühzeitig festzustellen, wie später noch gezeigt wird.

Mit verfügbaren Integrationen von Context Mapper [CM], einem Modeling-Framework zur textuellen Definition von Context Maps, sowie jMolecules [jMol], einer Bibliothek zur Annotation von Architekturkonzepten im Quellcode, ist die Definition von Soll- und Ist-Architektur äußerst einfach. Für die Soll-Architektur wird ein Context Mapper-Dokument angelegt, das die erlaubten Kontexte und deren Abhängigkeiten enthält. Diese Datei könnte wie in Listing 2 aussehen.

Zur Identifizierung der Ist-Architektur kann in diesem Beispiel jMolecules genutzt werden. Diese Open-Source-Bibliothek stellt Annotationen und Schnittstellen zur Anreicherung der Architektur im Code bereit. Annotationen können im Fall von Bounded Contexts direkt an Java-Packages im Rahmen von `package-info.java`-Dateien annotiert werden. Ein Beispiel ist in Listing 3 zu sehen.

Um diese Informationen in die lebendige Dokumentation zu integrieren, muss eine Datei namens index.adoc in `/jqassistant` angelegt werden. In dem Beispiel in Listing 4 sind zwei Abfragen hinterlegt: Die erste fragt die in der Context Map definierten Bounded Contexts und deren Beziehungen ab. Die in eckigen Klammern geschriebenen Meta-Informationen definieren die Abfrage als jQAssistant-Concept. Mit der Angabe des `reportType` wird das Ergebnis als Context Map gerendert. Die zweite Abfrage ist ein jQAssistant-Constraint, welches fehlschlägt, sollte es eine nicht-leere Ergebnismenge zurückliefern. Es prüft, ob Abhängigkeiten zwischen Kontexten implementiert sind, die nicht in der Soll-Architektur definiert wurden.

Um die Dokumentation ausführen zu können, müssen jQAssistant sowie die Integrationen für Context Mapper und jMolecules in den Maven-Buildprozess integriert werden. Ausschnittsweise ist dies in Listing 5 dargestellt. Das gerenderte Ergebnis nach der Ausführung von `mvn verify` ist in Abbildung 5 dargestellt. Ein vollständiges Beispiel für lebendige Dokumentation der Corona-Warn-App findet sich unter [CWA].

Fazit

Komplexe Softwaresysteme zukunftssicher zu machen, ist nicht unmöglich. Mit einem strukturierten Vorgehen bestehend aus etablierten Methoden des Strategic DDD, dem eingebetteten AI-SE-Prozess und modernen Tools ist diese Herausforderung effektiv und nachhaltig lösbar. Die größte Herausforderung bleibt, den ersten Schritt zu gehen. Seien Sie mutig, es lohnt sich!

Literatur und Links

[AISE] https://www.buschmais.de/aise-prozess/
[BBoM] B. Foote, J. Yoder, Big Ball of Mud, 4th Conf. on Patterns Languages of Programs (PLoP '97/EuroPLoP '97), September 1997, s. a. http://www.laputan.org/mud/mud.html#BigBallOfMud
[CDC] https://github.com/ddd-crew/core-domain-charts
[CM] https://contextmapper.org/
[CWA] https://github.com/jqassistant-demo/cwa-server
[ISO] https://de.wikipedia.org/wiki/ISO/IEC_9126
[jMol] https://github.com/xmolecules/jmolecules
[jQA] https://jqassistant.org/
[SAS] https://github.com/buschmais/software-analytics-starter