Sonderdruck aus OBJEKTspektrum

Modernisierung von Altanwendungen

Pragmatisch, praktisch, erfolgreich mit ADRs!

Die Umsetzung technischer Modernisierungen parallel zu neuen Features stellt Teams vor Herausforderungen. Beide Themen dürfen sich nicht negativ beeinflussen, um die Erreichung der jeweiligen Ziele nicht zu gefährden. Die Kommunikation technischer Änderungen und deren Implikationen für die Entwickler ist daher wichtig. Mit Architecture Decision Records (ADRs) und dem Open-Source-Tool jQAssistant stellt dieser Artikel einen erprobten, leichtgewichtigen Weg vor, um Änderungen zu kommunizieren und abzusichern.

Die Anwendungsmodernisierung ist ein essenzieller Bestandteil des Softwarelebenszyklus und unabdingbar, wenn es darum geht, langlebige Software zu entwickeln. Dennoch muss diese parallel zur funktionalen Weiterentwicklung erfolgen können, um weiter neue fachliche Anforderungen zu erfüllen. Wie können wir aber sicherstellen, dass im Zuge der Modernisierung eingeführte Konzepte nicht versehentlich durch die Weiterentwicklung wieder eingerissen werden? Hier spielt die effiziente Wissensverteilung eine entscheidende Rolle; ein Schlüsselelement ist dabei die Dokumentation und Absicherung der Architekturkonzepte.

Der Weg zur erfolgreichen Modernisierung

Modernisierungsmaßnahmen sind meist aufwendig und langläufig, da sie die Überarbeitung der Architektur und damit von Querschnittsaspekten eines Systems vorsehen. Dazu zählt der Austausch von Technologien genauso wie strukturelle Überarbeitungen oder Änderungen am Deployment-Modell. Durch den großen, breit streuenden Umfang werden Modernisierungsprojekte nicht selten zur unendlichen Geschichte.

Möchte man es besser machen, muss man zuerst die möglichen Probleme und Herausforderungen identifizieren. Kernaspekt ist dabei die Anforderung, dass die Modernisierung parallel zur Weiterentwicklung erfolgen muss. Es kann also zumeist nicht die Entwicklung neuer Features eingefroren werden, um technische Änderungen vorzunehmen.

- Planbarkeit der Gesamtmaßnahme und Teilschritte
- Monitorbarkeit und Absicherung der Fortschritte
- Kommunikation und Dokumentation der Maßnahmen

Kasten 1: Kriterien, die bei der Modernisierung zu berücksichtigen sind

Damit wir dies erfolgreich bewerkstelligen können, müssen wir folgende drei Herausforderungen lösen:

Erstens müssen Modernisierungsprojekte allen Entwicklern, auch denen, welche nicht an der Modernisierung arbeiten, einschließlich der Konsequenzen für ihre Arbeit bekannt sein. Dies ist notwendig, damit neue Features nicht entgegen getroffener Architekturentscheidungen implementiert werden und das Projektziel nicht zu einem Moving Target wird. Im Umkehrschluss bedeutet dies, dass Entscheidungen und Implikationen offen kommuniziert und für alle leicht zugänglich dokumentiert werden müssen.

Zweitens dürfen Modernisierungsmaßnahmen nicht zur unendlichen Geschichte verkommen. Um das Auseinanderdriften der Entwicklungszweige zu vermeiden und eine leichte Integration technischer Änderungen zu ermöglichen, ist eine kleinschrittige, iterative Arbeitsweise von Vorteil. Entsprechend wird eine detaillierte Planung mitsamt der Bestimmung der Konsequenzen für das System benötigt. Im Folgenden ist es notwendig, ein Monitoring für die Fortschritte zu haben.

Drittens müssen geschaffene Änderungen vor erneuter Erosion geschützt werden. Dabei inbegriffen ist ebenfalls die Vermeidung neuer technischer Schuld durch die Implementierung neuer Features entgegen getroffener Entscheidungen. Um dies möglichst effizient und transparent umzusetzen, bietet sich eine Tool-gestützte, automatisierte Absicherung an. Dies ist beispielsweise auch für das Onboarding neuer Entwickler oder das Nachvollziehen der Historie hilfreich.

Kasten 1 fasst noch einmal die Kriterien, um eine Modernisierungsmaßnahme erfolgreich umsetzen zu können, zusammen.

Architekturentscheidungen – Aber bitte leichtgewichtig

Michael Nygard hat mit *Architecture Decision Records* [ADR] eine leichtgewichtige Möglichkeit geschaffen, um Architekturentscheidungen nachvollziehbar und für alle zugänglich zu dokumentieren. Ein ADR dokumentiert immer genau eine signifikante Architekturentscheidung. Zusätzlich zu der Entscheidung selbst werden der Kontext, in welchem diese getroffen wird, und die daraus resultierenden Konsequenzen festgehalten. Damit ist es auch später, zum Beispiel für neue Entwickler, möglich zu verstehen, warum bestimmte Entscheidungen getroffen wurden.

Jedes ADR durchläuft einen eigenen Lebenszyklus, welcher im ADR selbst als Status sichtbar ist. Dieser ist in **Abbildung 1** dargestellt. Dadurch ist es leicht möglich, einen Überblick über aktuell gültige Architekturentscheidungen zu erhalten und die Historie nachzuvollziehen.

Alle ADRs werden aufsteigend nummeriert im Code-Repository abgelegt. Das hat den Vorteil, dass Entwickler leicht Zugriff auf diese haben. Auch können durch die Versionierung in einem Versionskontrollsystem (z. B. Git) Änderungen und die Auswirkungen auf den Code sichtbar gemacht werden. Als Format

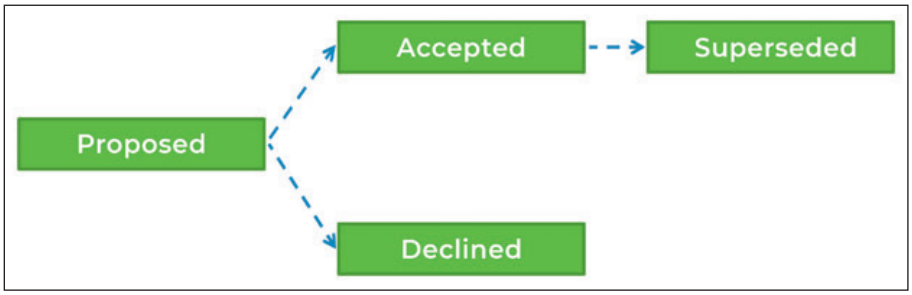

Abb. 1: Statusmodell der ADRs

für die ADRs bietet sich beispielsweise AsciiDoc an, welches insbesondere in Verbindung mit PlantUML ausreichend Möglichkeiten zur Dokumentation bietet. Notwendig ist es zudem, die ADRs während des Build-Prozesses zu rendern und zentral als HTML zu deployen. Damit ist es auch für Nicht-Entwickler leicht möglich, einen Überblick über die Anwendung zu erhalten.
In **Abbildung 2** ist eine reduzierte Version des ADR in seiner gerenderten Form dargestellt, welche wir in diesem Artikel erstellen.

Abb. 2: Das gerenderte ADR

ADRs in der Praxis

Als Beispiel soll ein E-Commerce-System in Java dienen. Dessen Produktkatalog soll aufgrund steigender Skalierungsanforderungen und der Flexibilität der Datenhaltung von einer relationalen zu einer dokumentenbasierten Datenbanktechnologie migriert werden. Der Einfachheit halber nehmen wir an, dass die Evaluation und ein Proof of Concept erfolgreich durchgeführt wurden und damit die Architekturentscheidung bereits getroffen ist. Unter [Git] ist das hier genutzte Beispiel zu finden.
Bei der Evaluierung hat sich herausgestellt, dass die dokumentenbasierte Datenbank die Bildung von Aggregaten im Domänenmodell erfordert und keine Deep Links im Modell erlaubt sind. Das heißt, dass jede Entität nur in genau einem Aggregat referenziert werden darf. Die Modellierung von Aggregaten ist im Katalog bereits umgesetzt und wir können diese manuell identifizieren. Jedoch sind Deep Links im Modell vorhanden, welche aufgelöst werden müssen. In einem ADR wollen wir das dokumentieren. **Abbildung 3** stellt beispielhafte, Aggregatübergreifende Deep Links dar.
Die Evaluierung und Entscheidung sind bereits durch einen kleinen Teil des Teams erfolgt. Somit müssen die Ergebnisse nun dem Rest des Teams präsentiert und zur finalen Diskussion gestellt werden.
Ein Hinweis am Rande: In Kundenprojekten hat es sich bewährt, wenn innerhalb eines gemeinsamen Meetings mit allen Entwicklern (natürlich nur, wenn das Team nicht zu groß ist) das vorbereitete ADR besprochen wird. Zum einen bekommt das Dokument so ein finales Review. Gegebenenfalls kommen noch bis dato unberücksichtigte Punkte auf, die mit aufgenommen werden müssen. Zum anderen wird dadurch jeder Entwickler direkt über die bevorstehende Änderung informiert.
Im Folgenden schauen wir uns die einzelnen Abschnitte genauer an. Dafür nutzen wir ein ADR, welches mit AsciiDoc verfasst wurde.
Listing 1 stellt das Grundgerüst des ADR in AsciiDoc-Notation dar. Zu sehen sind neben dem Titel und Status (hier: Proposed) bereits die Strukturierungselemente für die drei Abschnitte des ADR. Die ADRs sind nummeriert (hier: 001). Kommen weitere ADRs hinzu, würden diese unter der Nummer 002, 003, … geführt werden. ADRs sollten, nachdem sie abgelehnt oder akzeptiert wurden, bis auf den Status nicht mehr geändert werden. Ist eine Änderung oder Erweiterung der Entscheidung notwendig, wird ein neues ADR erstellt, welches das alte quasi überschreibt. Das ersetzte ADR wird anschließend auf „Superseded“ gesetzt. In der Praxis hat es sich jedoch für kleinere Änderungen als pragmatisch erwiesen, das existierende ADR anzupassen. Die Nachvollziehbarkeit der Änderungen darf dadurch aber nicht leiden.
Entstehen über die Zeit immer mehr ADRs, ist die Gruppierung eben dieser in Unterordnern sinnvoll. So könnten zum Beispiel in einem Ordner „DB“ alle die Persistenz betreffenden Entscheidungen abgelegt werden, während im Ordner „Test“ ADRs zum Testen der Anwendung liegen.

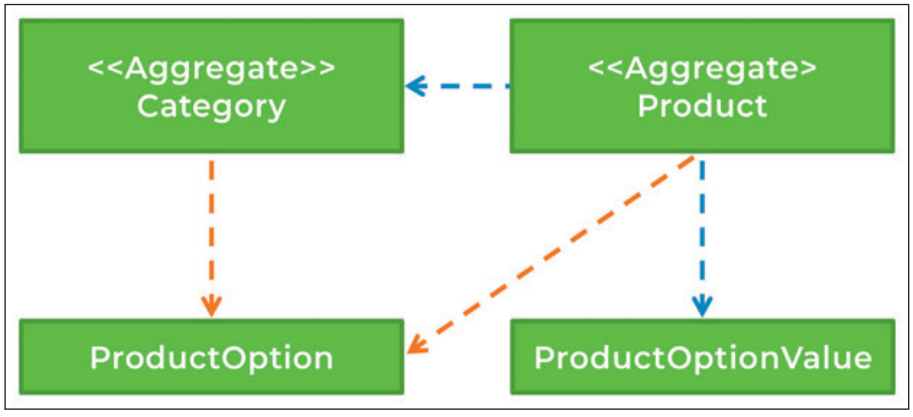

Abb. 3: Deep Linking im Domänenmodell

Ein ADR im Detail

Der Abschnitt „Kontext“ gibt einen Überblick über die Hintergründe und Bedingungen, unter welchen die Entscheidung getroffen wurde und für welche sie gültig ist. Dazu können technische Einschränkungen, Einschränkungen auf bestimmte Anwendungsteile oder auch externe Faktoren gehören. Ziel ist es, dass wir auch in Zukunft den Hintergrund der Entscheidung nachvollziehen können.

```
= 001 – Keine Deep Links
  zwischen Aggregaten
== Status: Proposed
== Kontext
== Entscheidung
== Konsequenzen
```

Listing 1: Grundgerüst des ADR in AsciiDoc

```
== Kontext
* Migration des Produktkatalogs von einer relationalen zu einer dokumentenbasierten Datenbank

== Entscheidung
* Dokumentenbasierte Datenbank erfordert die Bildung von Aggregaten
** Deep Links zwischen Aggregaten sind nicht mehr erlaubt

== Konsequenzen
* Alle Entitäten dürfen nur von einem Aggregate Root aus erreichbar sein
** Existierende Deep Links zwischen Aggregaten müssen aufgelöst werden
```

Listing 2: Ein Beispiel-ADR

Als Nächstes wird im Abschnitt „Entscheidung“ die getroffene Entscheidung formuliert. Hierbei soll klar und in aktiver Sprache ausgedrückt werden, was in Zukunft zu beachten ist.
Abschließend werden im Abschnitt „Konsequenzen“ die Auswirkungen der Entscheidung auf das Projekt aufgeführt. Sollten bestimmte Einschränkungen oder beachtenswerte Punkte aufkommen, sind diese hier zu dokumentieren. Damit dient der Abschnitt den Entwicklern als Blaupause für die weitere Arbeit.
Ebenfalls können hier Aufgaben definiert werden, welche sich aus der Entscheidung ableiten, also zum Beispiel die Refaktorisierung des bestehenden Codes. **Listing 2** zeigt mögliche Inhalte für die einzelnen Punkte.
Die gezeigten Beispieltexte weisen nur einen geringen Umfang auf. In echten Projekten werden diese ausführlicher ausfallen, um ein besseres Gesamtbild zeichnen zu können. Wichtig ist jedoch die klare Fokussierung des ADR auf ein konkretes Thema. Sollten sich mehrere Aspekte ausprägen, die jeweils in unterschiedlichen Entscheidungen oder Konsequenzen resultieren, ist es notwendig, weitere ADRs für diese Aspekte zu erstellen.

Sicher in die Zukunft mit ADRs und jQAssistant

Wir haben gesehen, wie Architekturentscheidungen und deren Konsequenzen effizient mit ADRs dokumentiert werden können. Leider schützt die Existenz dieses Dokuments aber nicht vor erneuten Architekturverletzungen. Auch wäre es schön zu sehen, welche Code-Stellen bereits die neuen Regeln verletzen, um diese Liste an Regelverletzungen zur Planung der Aufräumarbeiten nutzen zu können.
Das Softwareanalyse-Werkzeug jQAssistant [jQA], welches unter Open-Source-Lizenz primär durch die BUSCHMAIS GbR entwickelt wird, hilft dabei, die bestehenden Verletzungen automatisiert zu dokumentieren und vor einer erneuten Erosion zu schützen. Dafür lässt es sich als Maven- oder Gradle-Plug-in in den Build-Prozess integrieren und wird von da an bei jedem Build mit ausgeführt. Bei der Ausführung wird dann der Bytecode gescannt und eine Neo4j-Graphdatenbank aufgebaut. In **Abbildung 4** ist ein Teil dieses Graphen dargestellt. Zu sehen sind Knoten, die Java-Typen darstellen, und Relationen zwischen den Knoten, die die Abhängigkeiten abbilden.
Auf Basis dieser Daten können mit der Neo4j-Abfragesprache Cypher Konzepte und Regeln formuliert werden, die den Graphen weiter anreichern oder Strukturen überprüfen. Für eine Einführung in Cypher sei an dieser Stelle auf [Neo] verwiesen.
Da wir das ADR bereits mittels AsciiDoc formuliert haben, ist die Integration der automatisierten Dokumentation und Absicherung denkbar einfach. Zunächst einmal ist es notwendig, jQAssistant in das Projekt zu integrieren, wie in **Listing 3** am Beispiel von Maven dargestellt. Anschließend muss das ADR im Ordner „/jqassistant” abgelegt und, wie im GitHub-Repository gezeigt, aus der Datei „index.adoc” referenziert werden. Wird das Projekt nun mit `mvn clean install` gebaut, kann die gerenderte Datei unter `target/jqassistant/report/asciidoc` gefunden werden.
Im nächsten Schritt wollen wir das ADR um Regeln zur Prüfung der Einhaltung anreichern. Dazu müssen wir uns zunächst überlegen, wie wir dies prüfen können.

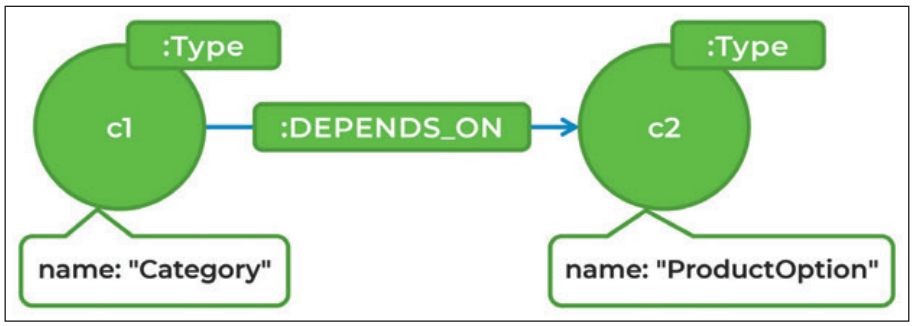

Abb. 4: Abbildung in Neo4j

Architekturkonzepte im Graphen abbilden

Wie schon erwähnt, ist es möglich, den Graphen weiter um eigene Konzepte anzureichern. Beispiele für Konzepte sind in unserem Fall das Aggregat sowie die Entität. Die Anreicherung im Graphen ist an dieser Stelle hilfreich, da sich so spätere Abfragen auf dem Graphen leichter umsetzen lassen.
Es ist notwendig, dass wir die Konzepte im Code identifizieren können, um den Graphen anzureichern. Dafür kommen grundsätzlich zum Beispiel Package- oder Klas-

```
<build>
  <plugins>
    <plugin>
      <groupId>
        com.buschmais.jqassistant
      </groupId>
      <artifactId>
        jqassistant-maven-plugin
      </artifactId>
      <version>1.8.0</version>
      <executions>
        <execution>
          <goals>
            <goal>scan</goal>
            <goal>analyze</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Listing 3: Integration in den Build-Prozess

```
import org.jqassistant.contrib.plugin.ddd.annotation.DDD;

@DDD.AggregateRoot
public class Product { }
```

Listing 4: Identifikation von DDD-Konzepten

```
[[adr:DeepLinking]]                                                    //1
[role=group,includesConstraints="adr:*"]                               //2
== 001 - Keine Deep Links zwischen Aggregaten

// Inhalte
[[adr:NoDeepLinkingBetweenAggregates]]                                 //3
[source,cypher,role=constraint,requiresConcepts="java-ddd:*"]          //4
.Findet alle Entitäten, welche in mehreren Aggregaten referenziert werden.
----
MATCH
  shortestPath((a:DDD:AggregateRoot)-[:DEPENDS_ON*]
  ->(e:DDD:Entity))
WITH
  e, collect(a.name) AS aggregates
WHERE
  size(aggregates) > 1
RETURN
  e.name AS Entity, aggregates AS Aggregates
----
```

Listing 5: Definition des Constraints

```
= Beispielprojekt für ADRs
[[default]]
[role=group,includesGroups="adr:*"]
- <<adr:DeepLinking>>

include::001-Deep-Linking.adoc[]
```

Listing 6: Referenzierung in der index.adoc

sennamen, Vererbungshierarchien oder auch Annotationen infrage. Möglich ist es aber auch, zum Beispiel Maven-Modulstrukturen zu nutzen. Das ist abhängig vom Projekt und kann auch bedeuten, dass entsprechende Eigenschaften noch geschaffen werden müssen. Die Aggregate und Entitäten in unserem Beispiel sind manuell identifizierbar.

Eine sehr einfache Variante zur Anreicherung von Konzepten aus dem Domain-Driven Design (DDD) im Code stellt das jQAssistant-DDD-Plug-in mit passenden Annotationen bereit. Nachdem es ins Projekt eingebunden wurde, können Klassen und Packages mit den bereitgestellten DDD-Konzepten annotiert werden. Anschließend erledigt das Plug-in die Anreicherung im Graphen für eben diese Konzepte. Auf diesem Weg ist, abgesehen von dem Annotieren der Klassen mit den verfügbaren Annotationen, keine zusätzliche Arbeit notwendig, um DDD-Konzepte sichtbar zu machen. Eine detaillierte Einführung in das Plug-in kann unter [101] gefunden werden. Dies muss im relevanten Code umgesetzt werden, wie **Listing 4** zeigt.

Einmal eine automatisierte Absicherung, bitte

Um sicherzustellen, dass kein Deep Linking zwischen Aggregaten erfolgt, müssen wir noch ein sogenanntes Constraint definieren. Dabei handelt es sich um eine Cypher-Abfrage, welche, je nach Konfiguration, den Build fehlschlagen lassen kann, sollte bei ihr die Ergebnismenge nicht leer sein. Das heißt, es müssen entsprechend alle Regelverletzungen zurückgegeben werden. Diese Regeln können wir direkt im AsciiDoc-Dokument verfassen. Als Bonus bekommen wir nach dem Build mit Maven das Resultat der Datenbankabfrage in der generierten HTML-Datei gerendert. Somit sind Monitoring und die Identifikation von Verletzungen leicht für alle möglich.

Wir integrieren diese Regeln bewusst direkt im ADR, da es sich in Projekten als hilfreich erwiesen hat, Architekturentscheidungen und deren Validierung auf einen Blick sehen zu können. Damit wird es außerdem unwahrscheinlicher, dass ADR und Constraint unabhängig voneinander geändert werden und es somit zu Inkonsistenzen kommt.

Listing 5 stellt die Integration des Constraints in AsciiDoc dar:

- An Position //1 wird eine jQAssistant-Regelgruppe mit dem Namen `adr:DeepLinking` definiert. Diese ist hilfreich, um auszuführende Gruppen zu definieren. So können bei bestimmten Builds auch einzelne Gruppen deaktiviert werden, zum Beispiel, wenn sie sehr lang laufende Abfragen beinhalten.
- In //2 wird definiert, welche Constraints, also Regeln, diese Gruppe beinhaltet.
- Punkt //3 wiederum definiert den Namen des Constraints und
- Punkt //4 fügt neben ein paar Metainformationen (`source, cypher`) die Abhängigkeit zu dem DDD-Plug-in hinzu. Damit wird sichergestellt, dass die Konzepte aus diesem Plug-in vorher ausgeführt werden.

Die `index.adoc` müssen wir entsprechend **Listing 6** um die Information der auszuführenden Regelgruppe anpassen.

Bauen wir das Projekt nun mit Maven, werden wir feststellen, dass das erstellte HTML eine Liste an Verletzungen der

Abb. 5: Darstellung der Verletzungen

Literatur & Links

[101] jQAssistant, Einstieg in das DDD-Plug-in, siehe: **https://101.jqassistant.org/ddd-plugin/**

[ADR] M. Nygard, Documenting Architecture Decisions, siehe: **http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions**

[Git] GitHub, Beispielprojekt, siehe: **https://github.com/jQAssistant/jqassistant-101/tree/master/src/main/tutorials/architecture-decision-records**

[jQA] jQAssistant, Startseite, siehe: **https://jqassistant.org/**

[Neo] Neo4j, Einstieg in Cypher, siehe: **https://neo4j.com/developer/cypher-basics-i/**

Regel enthält. Diese Übersicht kann nun als Grundlage für die Refaktorisierung genutzt werden. **Abbildung 5** zeigt die Verletzungen, welche im konkreten Fall die Referenzierung der `ProductOption` durch `Category` und `Product` sind.

Zusätzlich zu der Ausgabe im Report schlägt das Bauen des Projekts mittels Maven fehl, da das Constraint verletzt wird. Sichtbar wird das im Terminal. Mittels des Konfigurationsparameters `severity` können Regeln jedoch so konfiguriert werden, dass nur eine Warnung, jedoch kein Fehlschlagen entsteht. So können die Regeln bereits in das Projekt integriert werden, selbst wenn die Aufwände zur Behebung der bestehenden Verletzungen noch nicht abgeschlossen sind. Wurden einmal alle Verletzungen aufgelöst, kann die Severity des Constraints auf „Blocker" gestellt werden, um den Build bei erneuter Verletzung wieder fehlschlagen zu lassen.

Ausblick

Im Artikel wurde gezeigt, wie Architekturentscheidungen mit ADRs sowie den Tools AsciiDoc und jQAssistant leichtgewichtig dokumentiert werden können. Wir haben uns die Möglichkeiten zur automatisierten Validierung und Dokumentation von jQAssistant zunutze gemacht, um die Einhaltung dieser Entscheidungen direkt prüfen zu können und eine erneute Erosion zu vermeiden. Mit diesem Tooling können wir Modernisierungsprojekte effektiv und effizient umsetzen. Gleichzeitig können wir dank der Möglichkeit zur Dokumentation Architekturentscheidungen perfekt nachvollziehen. ||

Der Autor

Stephan Pirnbaum
(stephan.pirnbaum@buschmais.com)
ist Consultant bei der BUSCHMAIS GbR. Er beschäftigt sich leidenschaftlich gern mit der Analyse und strukturellen Verbesserung von Softwaresystemen im Java-Umfeld. In Vorträgen und Workshops präsentiert er seine gesammelten Erfahrungen und genutzten Methodiken.