Erfahrungsbericht Datenbankbasierte Metrikverarbeitung für Clean Code Development in Brownfieldprojekten

Jens Nerche

Kontext E GmbH
Fetscherstr. 10
01307 Dresden
j.nerche@kontext-e.de

Abstract: In einem neuen Projekt ("Greenfield Project") gleich mit Clean Code Development (CCD) zu beginnen, ist der Wunschtraum eines jeden Entwicklers. Was aber, wenn man ein Legacy Project hat? Unit Tests – wenn sie überhaupt vorhanden sind – laufen nicht, sind langsam oder bieten eine zu geringe Testabdeckung. Zyklen in Abhängigkeiten, wohin man schaut. Zu große Packages, zu lange Methoden, zu hohe zyklomatische Komplexität sind nur einige der häufig auftretenden Symptome. Einige CCD-Tugenden wie Pfadfinderregel, Pair Programming, Iterationen, VCS oder CI sind davon nicht betroffen. Sobald man aber Metriken erhebt und ggf. den Build abbricht, stößt man an die Grenzen. Erstmal ein Riesenrefactoring zu machen, ist wirtschaftlich ausgeschlossen. Die Investition in Codequalität ist nur in kleinen Schritten möglich. Wichtig ist auch die psychologische Komponente: im Team muss der Fortschritt sichtbar sein. Im Beitrag werden Erfahrungen vorgestellt, die dabei gesammelt wurden. Zentraler Bestandteil ist das Sammeln aller verfügbaren Informationen in einer (Graphen-)Datenbank: Codestatistiken, Modulabhängigkeiten, Unit Test Abdeckung, Ergebnisse der Statischen Codeanalyse, VCS-Metadaten, Tracing-Logs aus Systemtest und Betrieb usw. Hinzu kommen Informationen aus einer Basiserhebung des Qualitätsstands des Brownfieldprojekts. Damit kann die datenbankgestützte Verarbeitung und Auswertung erhobener Metriken und Softwarestrukturinformationen unter besonderer Berücksichtigung der Besonderheiten von Brownfieldprojekten erfolgen. Es ist die sinnvolle Definition eines Quality Gates im CI-Server sowie die Festlegung organisatorischer und technischer Maßnahmen möglich.

1 Einleitung und Motivation

Kontext E ist ein kleines Unternehmen mit Sitz in Dresden. Zu den Geschäftsaktivitäten gehören die Produktentwicklung, Projektrealisierung im Haus und Beratungsleistungen beim Kunden. Durch diese breite Aufstellung haben wir eine große Bandbreite in Bezug auf die Qualität von Quellcode und den Umgang mit den nichtfunktionalen Anforderungen, die die Codequalität beeinflussen, kennenlernen können. Insbesondere finden wir oft eine existierende Codebasis vor, die nicht nach dem Wertesystem des Clean Code

Development [CCD14] mit seinen Tugenden, Prinzipien und Praktiken entwickelt wurde, im Folgenden “Legacy Code” genannt. Neue Features zu erstellen dauert relativ lange und ist fehlerträchtig, da sehr viel “Technical Debt” entstanden ist.

Zwei der vier Aussagen des „Manifesto for Software Craftsmanship“ [MSC09] lauten:

- "Not only working software, but also well-crafted software."
- "Not only responding to change, but also steadily adding value."

In der professionellen Softwareentwicklung soll eine stetige wertsteigernde Weiterentwicklung ohne Geschwindigkeitsverlust stattfinden. Systeme sollen nicht zu Altsystemen verkommen, die dann abgelöst werden müssen, sondern kontinuierlich weiterentwickelt werden. Damit stellt sich die Frage, an welchen Stellen sich eine Investition in Codequalität am besten auszahlt. Es existieren zahlreiche Einzellösungen, die jeweils einen kleinen Ausschnitt betrachten wie z.B. zyklische Abhängigkeiten zwischen Paketen oder die Testabdeckung durch Unit Tests. Darauf gehen wir in Kapitel 2 ein.

Aus der Unzufriedenheit mit diesem Stand erwachsen Anforderungen an ein neues Werkzeug. Sie werden in Kapitel 3 dargestellt, verbunden mit den Zielen, die erreicht werden sollen. Kapitel 4 beschreibt die Entwicklung der Lösung. Die damit erreichten Ergebnisse werden in Kapitel 5 diskutiert.

2 Stand der Technik

In diesem Kapitel wird kurz beschrieben, welche der erwähnten Einzellösungen Kontext E bisher eingesetzt hat. Wir verwenden sie, weil sie im Java-Ökosystem als Standard gelten, gute Ergebnisse liefern, als Open Source Tools leicht verfügbar sind und sich gut in den Build-Prozess integrieren lassen.

2.1 Unit Tests und Test Coverage

Für Unit Tests wird das Testframework JUnit [JU14] eingesetzt. Die Testabdeckung wird mit Cobertura [Co14] oder JaCoCo [JCC14] ermittelt. Die Ergebnisse können u.a. in XML-Dateien geschrieben werden.

2.2 Statische Codeanalyse

Zur statischen Codeanalyse setzten wir ein:

1. **Checkstyle** [Ch14] führt vornehmlich Codestilprüfungen durch, erkennt aber auch Fehlermuster
2. **FindBugs** [FB14] erkennt typische Fehlermuster
3. **JDepend** [JD14]: ermittelt Paketabhängigkeiten und erkennt zyklische Abhängigkeiten

4. **JavaNCSS** [JN14] ermittelt Sourcecodestatistiken wie Anzahl der Zeilen pro Klasse, Klassen pro Paket usw.

2.3 Architektur- und Designvorgaben

Bisher setzten wir zwei Mechanismen ein, um Architektur- und Designvorgaben zu prüfen. Einerseits sind dies „Reflective Tests“ als spezielle Form des Integrationstests: compilierte Klassen werden per Reflection untersucht, die Regeln werden als JUnit-Testmethoden geschrieben. Der größte Nachteil ist, dass der Aufwand zur Prüfung einer Regel relativ hoch ist.

Andererseits verwendeten wir AspectJ [AJ14]. Diese Java-Erweiterung zur Aspektorientierten Programmierung ist auch in der Lage, selbst definierte Compilerwarnungen und -fehler zu generieren. Beispiele sind: System.out.println sollte nicht benutzt werden oder Modul A darf nicht von Modul B abhängen. Die Hauptnachteile liegen in der Beschränkung bzgl. der Möglichkeiten und in der Notation. Außerdem ist man an den AspectJ-Compiler gebunden, was in manchen Projekten mit den Kundenvorgaben kollidiert.

2.4 SonarQube

SonarQube [SQ14] ist eine offene Plattform zur Codequalitätsverwaltung. Es werden sieben Bereiche der Codequalität betrachtet: Architektur und Design, Duplikate, Unit Tests, Komplexität, potenzielle Fehler, Stilvorgaben und Kommentare. Das Qualitätsmodell von SonarQube ist für den praktischen Einsatz in Brownfieldprojekten nicht flexibel genug, so dass wir dieses Werkzeug nur zur manuellen Beobachtung langfristiger Trends einsetzen konnten.

2.5 Continuous Integration

Für die Inhouse-Projekte setzen wir CruiseControl [CC14] und Jenkins [Je14] als Continuous Integration Server ein.

3 Anforderungen und Ziele

Das Zusammentragen aller verfügbarer Metriken und Quellcodemetadaten in einer gemeinsamen Datenbank ermöglicht eine optimale fortlaufende Verbesserung des Quellcodes und stetige wertsteigernde Weiterentwicklung. Die Ablösung vieler getrennter Insellösungen mit verschiedenen Technologien (Scripte, XML-Verarbeitung) durch eine einheitliche, datenbankbasierte Lösung verringert den Entwicklungs- und Wartungsaufwand und erleichtert die Kombination und Verarbeitung der Rohmetriken erheblich. Weil es nun viel leichter ist, neue automatisiert prüfbare Regeln zu definieren, schreiben die Entwickler mehr von diesen. Außerdem ist es sehr einfach, für im Brownfield-Quellcode bestehende Unzulänglichkeiten Ausnahmen von den Regeln zu definieren. So werden auch Regeln definiert, die sonst höchstens manuell (und damit zu aufwändig und zu selten)

geprüft würden. Als Folge kann die Codebasis stetig verbessert werden, indem gezielt die Ausnahmen bearbeitet werden. Wird nicht eine firmeninterne Lösung erstellt, sondern auf ein Standardprodukt aufgesetzt, können extern erstellte Queries und Regeln wiederverwendet werden, der Entwicklungsaufwand sinkt weiter. Wenn mächtigere Prüfmöglichkeiten zur Verfügung stehen, können CCD-Prinzipverletzungen erkannt werden, die bisher unerkannt blieben, z.B. Objekterzeugung vs. Dependency Injection.

Wichtig ist zudem, dass unbekannter Code interaktiv erkundet werden kann, aber auch Regeln beim Continuous Integration automatisiert geprüft werden. Es muss eine einfache Möglichkeit geben, pragmatische Grenzen für Metriken zu definieren, die den CI-Build abbrechen lassen. Für den CI-Build müssen auch gängige Buildwerkzeuge wie ant [an14], maven [ma14] oder gradle [gr14] unterstützt werden.

Eine besondere Mächtigkeit entstünde durch die leichte Möglichkeit der Verarbeitung und Auswertung, insbesondere Kombination von Metriken untereinander und mit Informationen aus dem VCS, Tracing während der Ausführung und anderen Quellen. Damit ergibt sich die Forderung, auch komplexe Suchen und Regeln definieren zu können.

Besonders durch den geplanten Einsatz in Brownfieldprojekten mit Legacy Code gibt es weitere Anforderungen:

- Die Migration von einem Framework, einem Modul, einer Pattern Language oder einer API zu einer anderen erfolgt langsam Schritt für Schritt. Das bedeutet, dass eine Zeit lang parallele Verwendungen im Code existieren. Man möchte vermeiden, dass neuer Code die alte Lösung benutzt. Dafür müssen Regeln definierbar sein.
- Bei einem Architekturreview ist zu prüfen, ob die dokumentierte Architektur mit der struktureller Organisation des Quellcodes und der tatsächlichen/faktischen Architektur übereinstimmt. Indizien sind gemeinsam geänderte Dateien in VCS-Commits und Ablaufpfade im Code für Use Cases: wird in einer Komponente gearbeitet oder wild zwischen Komponenten hin- und hergesprungen?
- Die fortlaufende Verschlechterung der Codequalität soll aufgehalten oder zumindest die Geschwindigkeit verringert werden.
- Erreichte Verbesserungen sollen abgesichert werden können. Zum Beispiel sollen Bereiche defniniert werden können, in denen es keine zyklischen Abhängigkeiten mehr geben soll.
- Es sollen Daten erhoben werden können, die helfen, einen Kulturwandel einzuleiten. Solche Daten können bezogen auf ein VCS die Check-In-Häufigkeit, die Commit Size oder regelmäßige Uhrzeiten sein, zu denen eingecheckt wird.

Um eine flüssige Arbeitsweise zu erlauben und das manuelle Übertragen von Fundstellen zu vermeiden, sollte es eine IDE-Integration geben. Zur Visualisierung und Weiterverarbeitung mit anderen Werkzeugen soll ein Export von ausgewählten Daten sowie das Erzeugen von Reports während des CI-Prozesses möglich sein.

4 Lösungsentwicklung

4.1 Testabdeckung

Fast ausnahmslos ist in Legacy Code eine zu geringe Testabdeckung durch Unit Tests vorhanden. Micheal Feathers geht sogar so weit, dass er Legacy Code durch die Abwesenheit von Tests definiert [Fe05]. Für die gesamte Codebasis eine einzuhaltende Mindesttestabdeckung neu vorzugeben ist nicht sinnvoll, da sehr viele Tests nachgeschrieben werden müssten. Das ist für die Entwickler langweilig, führt zu Frust und der Nutzen ist sehr begrenzt. Die Entwicklung neuer Features würde sehr ausgebremst.

Würde die Mindesttestabdeckung hingegen nur für geänderte Klassen gelten, ergeben sich andere Probleme. Die Entwickler führen weniger Refactorings durch, wenn dadurch sekundär Klassen geändert werden, an denen sie nicht primär entwickeln. Beispiele sind “Rename Method”, “Remove Unused Parameter”, “Introduce Parameter” oder “Move Class”. Die Pfadfinderregel würde zu sehr eingeschränkt.

Statt die Pfadfinderregel einzuschränken sollte im Gegenteil ein Mechanismus gefunden werden, wie sie auf die Testabdeckung handlich angewendet werden kann. Wir suchen Kriterien, anhand derer der Quellcode unterteilt werden kann in Bereiche, die eine Investition in Testabdeckung erscheinen lassen und solche, in denen sich die Investition nicht auszahlt. Eric Evans unterteilt in [Ev04] die Domäne in “Core Domain” und “Generic Subdomains”. Entscheidend für den Erfolg ist die Core Domain. Sie bildet im Allgemeinen nur einen relativ kleinen Teil der Codebasis.

Des weiteren kann man eine risikoorientierte Unterteilung treffen. In Code, der häufig geändert wird oder der komplexe Algorithmen beinhaltet, ist die Wahrscheinlichkeit höher, dass Fehler beinhaltet sind oder entstehen. Um mit Hilfe eines Reports und des CI-Servers eine kontinuierliche Verbesserung erreichen zu können, haben wir folgendes Vorgehen entwickelt:

- Unit Tests ausführen
- Testabdeckung in eine XML-Datei schreiben
- Auswertung mittels eines Skriptes selbst vornehmen
- Komplexität nach McCabe [Mc76] messen und in Bereiche aufteilen

Dabei werden folgende Regeln angewendet:

- Gemessen wird auf Methodenebene – es wird nicht der Durchschnitt einer Klasse o.ä. gebildet.
- Es wird eine Anzahl von Methoden erlaubt, die nicht die Mindesttestabdeckung erreicht. Die Anzahl wird durch die Basiserhebung ermittelt.
- Es werden Pakete (Java) oder Namensräume (C#) definiert, in denen keine Verletzung der Mindesttestabdeckung erlaubt ist. Das kann einerseits die Core Domain sein, andererseits aber auch Bereiche, für die schon hinreichend viele Tests geschrieben wurden und die vor Regression geschützt werden sollen.
- Verbessert wird nun fortlaufend, die Fortschritte werden durch das Skript geschützt.

In Abbildung 1 sehen wir ein Beispiel der Mindesttestabdeckungskonfiguration für ein Java-Projekt. Zuerst werden die zwei Komplexitätsbereiche "low" und "medium" definiert. Implizit entstehen dadurch zwei weitere Bereiche "trivial" mit einer geringeren Komplexität als "low" und "high" mit einer höheren Komplexität als "medium". Für diese vier Bereiche wird danach die minimale Zweigabdeckung in Prozent definiert. Es werden 18 Ausnahmen toleriert, d. h. es darf 18 Methoden im ganzen Projekt geben, die die Mindesttestabdeckungsregel verletzen. In diesen tolerierten Ausnahmen nicht mit gezählt werden die beiden als "toleratedExceptions" definierten Ausnahmen – hier können einzelne Methoden, aber auch ganze Packages aufgeführt werden. Es werden jedoch keine Ausnahmen in "sauberen Bereichen" geduldet, angegeben bei "packagesWithoutTolerations". Damit kann man sowohl die Kerndomäne als auch Bereiche schützen, in denen schon Technical Debt abgebaut wurde, so dass sie sauber bleiben. Auch hier sind von Methoden bis Packages alle Namensräume erlaubt.

```
class CoverageCheckerConfiguration {
  static def rangeOfLowComplexity    = 2..3
  static def rangeOfMediumComplexity = (rangeOfLowComplexity.last()+1)..5

  static def minBranchCoverageForHighComplexity    = 90
  static def minBranchCoverageForMediumComplexity  = 80
  static def minBranchCoverageForLowComplexity     =  0
  static def minBranchCoverageForTrivialComplexity =  0

  static def toleratedViolations = 18

  static def toleratedExceptions = [
         "de.kontext_e.tim.jsf_ui.apps.LogExportServlet.readLogfileFromPath", // some comment
         "de.kontext_e.tim.infrastructure.tools.BuildVersion", // some comment
  ]

  static def packagesWithoutTolerations = [
         "de.kontext_e.tim.entities",
         "de.kontext_e.tim.domain",
         "de.kontext_e.tim.infrastructure.queries.QueryBuilder",
  ]
}
```

Abbildung 1: Konfiguration des Testauswertungsskripts

An diesem Beispiel sehen wir die Kombination zweier Metriken: die zyklomatische Komplexität nach McCabe und die Testabdeckung bzgl. durchlaufener Zweige. Dieses wird weiterhin kombiniert mit Design- und Architekturvorgaben: in der Kerndomäne wird keine Regelverletzung geduldet.

Somit bieten sich mehrere Parameter an, die für eine fortlaufende Verbesserung angepasst werden können:

- die Komplexitätsbereiche
- die Mindesttestabdeckung pro Komplexitätsbereich
- die Anzahl der erlaubten Schwellwertunterschreitungen
- die Definition von Paketen, in denen keine Schwellwertunterschreitung toleriert wird

4.2 Statische Codeanalysen

Mit den Metriken der statischen Codeanalysen verhält es sich äquivalent. Jedes Tool ist zwar in der Lage, den Build abzubrechen, wenn Regelverletzungen festgestellt werden. In Brownfieldprojekten ist dies wenig hilfreich, da es zahlreiche Regelverletzungen gibt. Somit werten wir die Rohdaten wieder mit Skripten aus. Da dies sehr ähnlich zu der Testabdeckung geschieht, wird hier auf eine ausführliche Darstellung verzichtet.

4.3 Zusammenführung aller Metriken sowie der Architekturstrukturinformationen

Die bisher beschriebene Vorgehensweise hat sich sehr gut bewährt. Um die Codequalität noch besser beherrschbar zu machen, sollte die Auswertung auf eine neue Stufe gehoben werden. Das Ziel war, nicht nur einige Metriken zu kombinieren, sondern mit einem universellen Werkzeug alle erhobenen Daten miteinander verknüpfen zu können. Für diesen Zweck etablierte Standardsoftware sind die Datenbanken. Neben den weit verbreiteten relationalen Datenbanken ist eine Vielfalt an sogenannten NoSQL-Datenbanken [SF12] verfügbar. Somit kann anhand der Natur der Daten die am besten passende Speichertechnologie gewählt werden. Auf Grund der starken Vernetzung der Daten fiel die Entscheidung auf eine Graphendatenbank [RWE13].

Eine Untersuchung des Marktes zeigte, dass es bereits ein Produkt gibt, welches sich sehr gut als Basis für das angestrebte Ziel eignet: jQAssistant [jQA14].

jQAssistant wird seit März 2013 durch die Firma Buschmais entwickelt. Es steht gegenwärtig unter der Apache 2.0 Lizenz und wird auf GitHub gehostet. Erweiterbarkeit ist explizites Designziel, Plugins zum Datenimport und zur Auswertung in Form von Reports sind sehr leicht erstellbar. In jQAssistant schon vorhanden sind Importer für Javastrukturinformationen (Pakete, Klassen, Methoden, Felder, Annotationen usw.), CDI, EJB 3, JAX RS, JPA 2, JUnit 4, Maven 3 und OSGi. Als Datenbank wird Neo4j [Ne14] verwendet. Dies ist eine Graphendatenbank. Deren Abfragesprache heißt “Cypher” [Cy14].

Für die erhobenen Daten zur Testabdeckung, der statischen Codeanalyse sowie aus der Versionsverwaltung haben wir zusätzliche Import-Plugins erstellt, die bei Maven Central [MC14] veröffentlicht sind:

- **FindBugs** Plugin
- **Checkstyle** Plugin
- **Jacoco** Plugin
- **Git** Plugin
- **PMD** Plugin

4.4 IDE-Integration

Ein Ziel war, die gesammelten Daten aus der Datenbank so einfach und schnell wie möglich in der IDE nutzbar zu machen. Dies geschieht durch ein IDE-Plugin, welches wir erstellt und im JetBrains Plugin Repository [jP14] veröffentlicht haben.

4.5 Die Basiserhebung

Bei Projektbeginn ohne vorhandenen Code sind die Regeln für die Codequalität einfach zu definieren: keine zyklischen Abhängigkeiten, keine verbotenen Methoden verwenden, Mindesttestabdeckung etc. Es ist (noch) kein Technical Debt vorhanden, der die Regeln brechen würde. Anders im Legacy Code: werden Regeln nachträglich eingeführt, gibt es zahlreiche Stellen, die dagegen verstoßen. All diese Stellen zu ermitteln ist Ziel der Basiserhebung. Diese Daten werden mit in die Metrik-Datenbank eingepflegt, so dass sie bei der Formulierung der Regeln beachtet werden können. Damit verschiebt sich die Baseline für die Codequalität. Wie in Abbildung 2 dargestellt wird sie auf den symbolischen Wert X normiert.

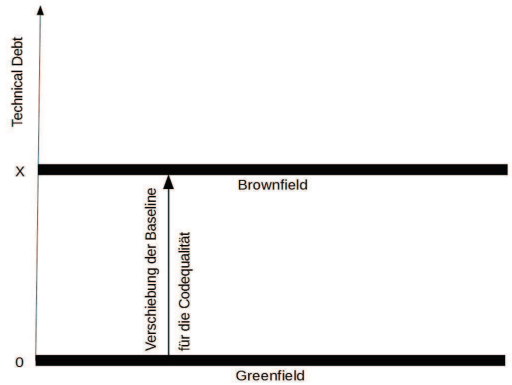

Abbildung 2: Basiserhebung des Technical Debt

5 Ergebnisse

Der Erfahrungsbericht bezieht sich auf zwei Projekte, in denen wir die datenbankgestützte Verarbeitung der erfassten Metriken eingeführt haben.

Das erste war ein internes Java-Projekt mit ca. 10.000 Codezeilen, welches gute Eigenschaften hat, um für Experimente mit neuen Technologien zu dienen: es ist größer als ein akademisches Beispiel, im internen produktiven Einsatz, also mit praxisrelevanten funtkionalen und nichtfunktionalen Anforderungen und beinhaltet verschiedene Technologien: JSF, JavaScript, Spring Framework, Spring Data JPA sowie diverse Drittbibliotheken. Die Metriken wurden bereits erhoben und verarbeitet. Die Screenshots, Beispielregeln usw. beziehen sich auf dieses Projekt.

Das zweite war ein kommerzielles Kundenprojekt mit ähnlichen Technologien, aber größerem Umfang (ca. 500.000 Codezeilen). Mit der in Kapitel 4 erarbeiteten Lösung konnten die Anforderungen aus Kapitel 3 erfüllt werden. Die Neo4j-Cypher-Queries der jQAssistant-Datenbank sind viel einfacher pflegbar als ein Zoo von Insellösungen. Es sind viele neue Regeln entstanden. Auch Queries aus Drittquellen wurden angepasst, wiederverwendet und in Regeln umgewandelt. Mit einer Basiserhebung wurden die initialen Grenzwerte und Regelausnahmen ermittelt. Durch fortlaufende Bearbeitung des Technical Debt wurden die Grenzen und Ausnahmen immer wieder angepasst.

Gleichzeitig entstanden neue, mächtige Möglichkeiten, da Daten viel einfacher miteinander verknüpft werden können. Insbesondere in langjährig entwickeltem Legacy Code, wenn Wissensträger das Projekt verlassen haben, oder durch Zukauf eine vorhandene Codebasis ins Projekt kommt, spielt diese neue gewonnene Dimension der Erforschung der Codebasis eine große Rolle (“Softwarearcheologie”, “Data Mining”). Da die Neo4j-Datenbank auch ohne jQAssistant verwendet werden kann, steht mit dem Neo4j-Browser [NB14] ein Werkzeug zur interaktiven Erkundung der Codebasis bereit. Zur Illustration seien hier drei typische Beispiele genannt:

- Codereview: finde alle Dateien, die zu allen Commits gehören, die in den Commit-Kommentaren die Kennzeichnung für die bearbeitete Bugtracking-ID enthalten.
- Finde alle Klassen, in denen FindBugs oder Checkstyle einen bestimmten Satz von Fehlermustern gefunden haben
- Kleines Cleanup während der Wartezeit auf Zuarbeit: finde alle Klassen, deren “Toxicity”-Metrik (vgl. [Dö13]) einen bestimmten Schwellwert überschritten hat und die in den letzten 12 Monaten bearbeitet wurden, geordnet nach der Anzahl der Änderungen und beschränkt auf die ersten 10 Fundstellen

Die Investition in die Plugins für das Werkzeug und die Einführung in die Projekte lag im Bereich einiger Personentage (Schätzwert, da nicht per Zeiterfassungswerkzeug gemessen). Jedoch ist der Return on Investment schwierig zu ermitteln, da der eigentliche Nutzen ist, dass durch die ständigen Codeverbesserungen der Aufwand für neue und geänderte Features niedriger ist als ohne die Codeverbesserungen. Man müsste in bestehen-

der und verbesserter Codebasis mit statistisch relevanter Entwickleranzahl Versuche durchführen. Dies wurde nicht gemacht.

5.1 Ersetzen von Tools

```
<constraint id="tim:PackageCycles">
    <requiresConcept refId="dependency:Package"/>
    <description>There must be no cyclic package dependencies.</description>
    <cypher><![CDATA[
        MATCH
            (p1:Package)-[:DEPENDS_ON]->(p2:Package),
            path=shortestPath((p2)-[:DEPENDS_ON*]->(p1:Package))
        WHERE
            p1<>p2
            and not(p1.fqn =~ 'de.kontext_e.tim.application.database.*')
            and not(p1.fqn =~ 'de.kontext_e.tim.application.domain_model.*')
            and not(p1.fqn =~ 'de.kontext_e.tim.application.entities.*')
            and not(p1.fqn =~ 'de.kontext_e.tim.application.executionlog.*')
            and not(p1.fqn =~ 'de.kontext_e.tim.application.lowlevel_services.*')
            and not(p1.fqn =~ 'de.kontext_e.tim.tests.*')
            and not(p1.fqn =~ 'de.kontext_e.common.tests.*')
        RETURN
            p1 AS package, EXTRACT(p IN NODES(path) | p.fqn) AS cycle
        ORDER BY
            package.fqn;
    ]]></cypher>
</constraint>
```

Abbildung 3: Regeldefinition zyklische Abhängigkeiten

```
<constraint id="tim:MethodsPerClass">
    <description>There must be 66 or less methods per class.</description>
    <cypher><![CDATA[
      MATCH
        (c:Class)-[:DECLARES]->(m:Method)
      WITH
        count(m) as numberOfMethods, c
      WHERE
        numberOfMethods>66
                and not(c.fqn =~ 'de.kontext_e.tim.ic.interfaces.jsfwebapp.*')
                and not(c.fqn =~ 'de.kontext_e.tim.tests.*')
                and not(c.fqn =~ 'de.kontext_e.common.tests.*')
      RETURN
        c.fqn, numberOfMethods
      ORDER BY
        numberOfMethods DESC
    ]]></cypher>
</constraint>
```

Abbildung 4: Regeldefinition maximale Anzahl von Methoden pro Klasse

Ein Ziel war, die Anzahl der eingesetzten Werkzeuge zu reduzieren. Dies ist gelungen, JDepend und JavaNCSS konnten komplett ersetzt werden. Positiv in Bezug auf Brownfieldprojekte ist, dass Ausnahmen einfacher zu definieren sind. Sehen wir uns das Beispiel in Abbildung 3 an. Es wird geprüft, dass Java-Pakete nicht zyklisch voneinander

abhängig sind. In bestehendem Code ist dies nicht immer der Fall, und man sieht, dass die Ausnahmen (de.kontext_e.tim...) sehr einfach zu definieren sind. Abbildung 4 zeigt das Äquivalent für das Ersetzen von JavaNCSS. Es wird geprüft, dass die Anzahl der Klassen pro Methode eine Höchstzahl nicht überschreitet, Ausnahmen für Pakete sind genau so einfach definiert.

5.2 Nutzung der erweiterten Analysemöglichkeiten bzgl. Architektur und Design

- Unter Nutzung der nun in der Datenbank vorliegenden Strukturinformationen konnten wir wie oben genannt leicht neue Regeln definieren. Im Folgenden werden Beispiele für solche Regeln aufgezählt:
- Jedes Entity (vgl. [Ev04]) muss "toString" überschreiben: Entities sind Objekte, die Daten enthalten. Damit in Traceausgaben sinnvolle Informationen stehen, muss die "toString"-Methode überschrieben werden.
- Entities können in Parent-Child-Beziehungen miteinander stehen und Aggregate bilden (vgl. [Ev04]). Üblicherweise enthält die "toString"-Methode des Parent die Collection der Children, bei denen implizit auch "toString" aufgerufen wird. Die Children ihrerseit dürfen nicht den Parent in der "toString"-Methode enthalten, da sonst eine unendliche Aufrufkette entsteht.
- Es ist bekannt, dass die Java-Klassenbibliothek defekte Klassen und Methoden enthält. Insbesondere die Date- und Calendar-API ist davon betroffen. Diese Methoden dürfen nicht verwendet werden. Gleiches gilt für Mutatoren, die nicht verwendet werden dürfen, wenn Klassen Immutable sein sollen, deren Code nicht geändert werden kann sowie für die unerlaubte Konstruktion von Objekten, wenn Dependency Injection verwendet werden soll. Wir haben einen Satz von Regeln definiert, der den Aufruf von "verbotenen Methoden" prüft.
- Wenn eine Codebasis eine neuere Version einer Bibliothek verwendet, kann es vorkommen, dass verwendete Methoden nun als "deprecated" gekennzeichnet sind. Eine Regel zur abnehmenden Verwendung dieser Methoden kann bei der langsamen Migration zur neuen API helfen.
- Ähnlich verhält es sich bei der Migration von einem Framework zu einem anderen. Wir migrieren beispielsweise vom Mocking-Framework "jMock" [jM14] zu "Mockito" [Mo14]. Neue Tests dürfen jMock nicht mehr benutzen, aber bestehender Code wird nur umgestellt, wenn er in Folge von Funktionsänderungen angepasst werden muss (Pfadfinderregel).

5.3 Nutzung der erweiterten Analysemöglichkeiten durch zusammengeführte Daten

- Die Zusammenführung der Daten aus verschiedenen Quellen hat es sehr einfach gemacht, Fragen nachzugehen, die ansonsten sehr viel aufwändiger zu recherchieren wären. Der gesunkene Aufwand macht es im realen Projektalltag überhaupt erst möglich, Fragen zu untersuchen wie:

- Haben häufig geänderte Klassen eine genügend große Testabdeckung und weisen sie eine geringe Anzahl an Warnungen durch statische Codeanalysen auf?
- Haben Schulungsmaßnahmen die erwünschte Wirkung gezeigt, d.h. treten bestimmte Bug Pattern (statische Codeanalyse) bei Commits von den Schulungsteilnehmern nach dem Schulungstermin nicht mehr auf?
- Hängen instabile Tests mit Änderungen an bestimmten Klassen zusammen oder treten sie willkürlich auf?
- Welches war der Commit, der die Testausführung so viel langsamer gemacht hat?

Diese und viele weitere bisher verborgene Zusammenhänge technischer und organisatorischer Natur können nun leicht untersucht werden.

6 Zusammenfassung und Ausblick

Die Einführung des Wertesystems des Clean Code Developments in Brownfieldprojekte stellt viele Herausforderungen. Einige davon lassen sich gezielt adressieren, indem bisher getrennt betrachtete Metriken, VCS-Daten, Laufzeitinformationen usw. in einer gemeinsamen Datenbank gesammelt werden. Die Informationen dieser Datenbasis können genutzt werden, um sowohl explorativ Einsichten in den Quellcode und dessen zeitliche Veränderung gewinnen zu können als auch um zugeschnittene Regeln zur Codequalitätsprüfung im CI-Prozess zu definieren.

Erste Erkenntnisse wurden in diesem Erfahrungsbericht niedergelegt. Die volle Mächtigkeit einer offenen Plattform und die Auswirkungen auf die Art und Weise des Programmierens sind noch nicht absehbar, in den kommenden Jahren sind dahingehend spannende Entwicklungen zu erwarten. Daher sind weitere Evaluierungen nötig:

- Aus der Vielzahl der verfügbaren Metriken diejenigen auswählen, deren Relevanz und Nutzen für das CCD am höchsten ist
- Visualisierung und Data Mining sind im Bereich des Business Intelligence schon lange etabliert und sollten auch in der professionellen Softwareentwicklung stärker angewendet werden
- Erkenntnisse über die Kohäsion von Klassen würde die Analyse der Clusterung von gemeinsam eingecheckten Klassen in VCS-Commits liefern.
- Bisher wurde nur die Find-Funktion realisiert. Für ein genau so mächtiges Replace müssten AST-basierte Scripte geschrieben werden. Einen Prototyp haben wir mittels Dmitry Kandalovs live-plugin [lp14] realisiert und als Gist veröffentlicht [De14]
- Der Einsatz in großen monolithischen Projekten mit mehreren Millionen Codezeilen, wo Skalierungseffekte zum Tragen kommen

Für eine Weiterentwicklung kommen mehrere Bereiche in Betracht. Im Gegensatz zum Datenimport ist der Datenexport noch wenig entwickelt. Naheliegend sind:

- Die automatische Erstellung von Grafiken für die Architekturdokumentation.
- Die Visualisierungen von Metriken, z.B. durch CodeCity, vgl. [We10]
- Der Export der Daten zur Weiterverarbeitung durch andere Werkzeuge.

Eventuell ist es sinnvoll, den kompletten Abstrakten Syntaxbaum in die Graphendatenbank zu importieren, um noch feingranularere Auswertungen und Regeln zu ermöglichen.

Nicht zuletzt ist auch eine Portierung auf andere Plattformen als .NET sinnvoll.

Literaturverzeichnis

[Ad11] Adzic, Gojco: Specification by Excample, Manning Publications 2011.
[ACM01] Alur, Deepak, Crupi, John, Malks: core J2EE Patterns – Best Practices and Design Strategies, Sun Microsystems Press 2001.
[AJ14] AspectJ, https://eclipse.org/aspectj/ 2014.
[an14] Apache Ant, http://ant.apache.org/ 2014.
[Bi12] Bien, Adam: Java EE Patterns – Rethinking Best Practices, press.adam-bien.com 2012.
[Be03] Beck, Kent: Test-Driven Development by Example, Addison-Wesley 2003.
[BG05] Bloch, Joshua; Gafter, Neal: Java Puzzlers – Traps, Pitfalls, and Corner Cases, Addison-Wesley 2005.
[Bl08] Bloch, Joshua: Effective Java, Addison-Wesley 2008.
[CC14] CruiseControl, http://cruisecontrol.sourceforge.net/ 2014.
[CCD14] Westphal, Ralf, Lieske Thomas: Clean Code Developer Wiki http://clean-code-developer.de 2014.
[CDI09] JSR 330: Dependency Injection for Java, https://jcp.org/en/jsr/detail?id=330 2009.
[Ch14] Checkstyle, http://checkstyle.sourceforge.net/ 2014.
[Co14] Cobertura, http://cobertura.github.io/cobertura/ 2014.
[Cy14] Cypher query language, http://neo4j.com/developer/cypher/ 2014.
[DDN13] Demeyer, Serge, Ducasse, Stephane, Nierstrasz, Oscar: Object-Oriented Reengineering Patterns, http://scg.unibe.ch/download/oorp/ 2013.
[De14] DeepStaticMethodsAction.groovy https://gist.github.com/jensnerche/719ed00347da14cb506c 2014.
[Dö13] Dörnenburg, Erik: Toxicity reloaded, http://erik.doernenburg.com/2013/06/toxicity-reloaded/ 2013.
[Ev04] Evans, Eric: Domain-Driven Design, Addison-Wesley 2004.
[Ev11] Evans, Eric: Four Strategies for Dealing with Legacy Systems, http://dddcommunity.org/library/evans_2011_2/ 2011.
[FB14] FindBugs, http://findbugs.sourceforge.net/ 2014.
[Fe05] Feathers, Michel C.: Working Effectively With Legacy Code, Prentice Hall 2005.
[Fo99] Fowler, Martin; et al.: Refactoring – Improving the Design of Existing Code, Addison-Wesley 1999.
[FP11] Freeman, Steve; Pryce, Nat: Growing Object-Oriented Software, Guided by Tests, Addison-Wesley 2011.
[gr14] gradle: https://www.gradle.org/ 2014.
[He10] Henney, Kevlin (Hrsg.): 97 Things Every Programmer Should Know, O'Reilly 2010.
[HF10] Humble, Jez; Farley, David: Continuous Delivery, Addison-Wesley 2010.
[HT00] Hunt, Andrew; Thomas, David: The Pragmatic Programmer, Addison-Wesley 2000.

[Hu14] Hunger, Michael: Using AsciiArt to analyse your SourceCode with Neo4j and OSS Tools, https://de.slideshare.net/neo4j/software-analytics 2014.
[JCC14] JaCoCo Java Code Coverage Library, http://www.eclemma.org/jacoco/ 2014.
[JD14] JDepend: http://clarkware.com/software/JDepend.html 2014.
[Je14] Jenkins, http://jenkins-ci.org 2014.
[jM14] jMock Framework: http://jmock.org/ 2014.
[JN14] JavaNCSS: http://javancss.codehaus.org/, 2014.
[jP14] IntelliJ IDEA Plugin für jQAssistant: http://plugins.jetbrains.com/plugin/7618?pr=idea 2014.
[jQA14] Buschmais: jQAssistant. http://jqassistant.org/ 2014.
[JU14] JUnit: http://junit.org/ 2014.
[lp14] live-plugin: https://github.com/dkandalov/live-plugin 2014.
[Ma09] Martin, Robert: Clean Code – A Handbook of Agile Software Craftsmanship, Prentice Hall 2009.
[Ma11] Martin, Robert: The Clean Coder: A Code of Conduct for Professional Programmers, Prentice Hall 2011.
[Ma14] Apache Maven: http://maven.apache.org/ 2014.
[Mc76] McCabe, Thomas J.: A Complexity Measure: IEEE Transactions on Software Engineering: 308–320, 1976.
[MC14] jQAssistant Plugins bei Maven Central: http://search.maven.org/#search|ga|1|kontext-e 2014.
[Me07] Meszaros, Gerard: xUnit Test Patterns – Refactoring Test Code, Addison-Wesley 2007.
[Mo14] Mockito Framework: https://code.google.com/p/mockito/ 2014.
[MSC09] Manifesto for Software Craftsmanship. http://manifesto.softwarecraftsmanship.org/ 2009.
[Ne14] Neo4j: http://neo4j.com/ 2014.
[NB14] Neo4j Browser: http://neo4j.com/developer/guide-data- visualization/#_the_neo4j_browser 2014.
[Ni06] Nilsson, Jimmy: Applying Domain-Driven Design and Patterns, Addison-Wesley 2006.
[PU14] PlantUML. http://plantuml.sourceforge.net/ 2014.
[RWE13] Robinson, Ian; Webber, Jim; Eifrem, Emil: Graph Databases, O'Reilly 2013.
[SF12] Sadalage, Pramod J.; Fowler, Martin: NoSQL Distilled – A Brief Guide to the Emerging World of Polyglot Persistence, Addison-Wesley 2012.
[SQ14] SonarQube, http://www.sonarqube.org/ 2014.
[We10] Wettel, Richard: Welcome to CodeCity! http://www.inf.usi.ch/phd/wettel/codecity.html 2010.