Statische Code-Analyse mit SonarQube

Josha von Gizycki, TRIOLOGY GmbH

„Software-Qualität" ist nach wie vor ein Dauerthema, denn sowohl Kunden als auch Entwickler wissen, welche Folgen fehlerhafte Software haben kann. Umso wichtiger ist es, den Quellcode regelmäßig einer entsprechenden Untersuchung zu unterziehen.

Tools zur statischen Code-Analyse können einen wichtigen Beitrag liefern. Die Analyse kann dazu beitragen, Fehler, Sicherheitslücken oder einen ungenügenden Aufbau ausfindig zu machen, ohne dass dazu eine Ausführung notwendig ist. Der Artikel zeigt, wie eine statische Code-Analyse funktioniert, welche Vorteile das Tool SonarQube bietet und warum eine umfassende Interpretation notwendig ist.

Was macht statische Code-Analyse?

Statische Code-Analyse nimmt kompilierten Code oder Quelltext, wendet Metriken darauf an und generiert Zahlen. Damit dies nicht von Hand erfolgen muss, gibt es eine Vielzahl unterschiedlicher Tools, auf die zurückgegriffen werden kann. Die bekanntesten in der Java-Welt sind „PMD", „Checkstyle" und „FindBugs":

- „PMD" sucht nach ineffizientem Code. Darunter fallen beispielsweise leere Codeblöcke, ungenutzte Variablen oder verschwenderisches Nutzen von „Strings" oder „StringBuffern".
- „Checkstyle" prüft den Programmierstil und arbeitet dabei genau wie „PMD" auf Quelltext-Ebene. Dadurch kann die Einhaltung von Programmier-Richtlinien oder Formattern erzwungen werden.
- „FindBugs" arbeitet als einziges der drei genannten Tools auf Bytecode-Ebene. Es sucht dabei nach harten Fehlern wie Problemen in Klassen-Hierarchien, fehlerhaften Array-Behandlungen, unmöglichen Typen-Umwandlungen oder nicht in Paaren überschriebenen „equals"- und „hashcode"-Methoden.

Diese Werkzeuge nutzen entsprechend ihrer Natur und Einsatzgebiete mal mehr, mal weniger Metriken, um ihre jeweiligen Statistiken zu erzeugen. Der Name „Metrik" trägt jedoch wenig Bedeutung von dem in sich, was eine Metrik ausmacht. So hat der Name seinen Ursprung im Lateinischen „ars metrica", was mit „Lehre von den Maßen" übersetzt wird. Fragt man jedoch das Institute of Electrical and Electronics Engineers, erhält man folgende Antwort: „Software quality metric: A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality." „Eine Software-Qualitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet, der als Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit interpretierbar ist." – IEEE-Standard 1061, 1998.

In diesem Fall wird eine Metrik als eine Funktion verstanden, die für beliebige Eingaben Zahlen erzeugt. Diese sind dabei so beschaffen, dass sie untereinander vergleichbar sind, solange sie von derselben Funktion erzeugt wurden. Auf diese Weise können Rückschlüsse auf die Eingabe mit Hinblick auf die Funktion gezogen werden.

Ein Beispiel dafür ist die McCabe-Metrik, auch „zyklomatische Komplexität" genannt. Diese sehr grundlegende Metrik berechnet die Anzahl der unterschiedlichen Pfade durch ein Stück Code. Die For-

```
String nameOfDayInWeek(int nr) {
    switch(nr) {
        case 1: return "Monday";
        case 2: return "Tuesday";
        case 3: return "Wednesday";
        case 4: return "Thursday";
        case 5: return "Friday";
        case 6: return "Saturday";
        case 7: return "Sunday";
    }
    return "";
}
```

Listing 1

```
String nameOfDayInWeek(int nr) {
    String[] names =  new String[] {
        "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday",
        "Sunday"
    };
    if(nr > 0 && nr <= names.length) {
        return names[nr - 1];
    }
    return "";
}
```

Listing 2

mel ist sehr einfach: Es wird die Anzahl von Kontrollstrukturen wie „if", „while", „case" sowie boolescher Operatoren wie „&&" und „||" hochsummiert und 1 addiert. Diese Information soll anhand eines Beispiels verdeutlicht werden *(siehe Listing 1)*.

Diese sehr einfache Methode gibt den Namen eines Wochentages entsprechend seiner 1-indizierten Position innerhalb der Woche zurück. Ihre zyklomatische Komplexität beträgt acht: 1 plus 7 mal „case". Dies ist ein relativ hoher Wert: Ein Maximalwert von 10 gilt als allgemein akzeptiert und ausreichend erprobt. Um also die Komplexität dieser Methode zu verringern, wird sie refaktorisiert *(siehe Listing 2)*.

Die zyklomatische Komplexität dieser Methode beträgt drei: „1 plus 1 mal if plus 1 mal &&". Durch den unterschiedlichen Ansatz wird die Komplexität verringert, jedoch ist es relativ unstrittig, dass die erste Version schneller verstanden werden kann.

Sollen die genannten Werkzeuge nun zusammen benutzt werden, müssen alle entsprechend konfiguriert und ihre Ergebnisse zusammengeführt werden, damit sich ein gemeinsames Bild ergibt. Außerdem kommt es dabei zwangsweise zu Dopplungen in ausgewerteten Metriken oder anderen Kennzahlen. „PMD" beispielsweise besitzt durch seinen relativ vagen Aufgabenbereich Überschneidungen im Hinblick auf Codestil mit „Checkstyle", während es aber auch genauso wie „FindBugs" auf ungenutzten Code achtet. An solchen und weiteren Stellen kann SonarQube Verbesserungen herbeiführen.

SonarQube

SonarQube besteht im Wesentlichen aus drei grob voneinander abtrennbaren Komponenten: einem Scanner, der Code entgegennimmt und analysiert, einer Datenbank, in der Analyse-Ergebnisse gespeichert werden, und einer Web-Komponente, die die gesammelten Ergebnisse aufbereitet anzeigt. Auf diese Weise kann der Scanner aus beliebigen Quellen aufgerufen werden, beispielsweise von einem Maven-Build, einem CI-Server oder aus IDEs heraus. SonarQube steht unter der LGPL v3 und ist damit Open Source.

In Sachen Analyse und Metriken bedient sich SonarQube dabei unter anderem der bereits genannten Tools. Die Analysen aus „PMD" und „Checkstyle" sind fest integriert, „FindBugs" kann über eine Plug-in-Schnittstelle nachinstalliert werden.

Für die Analyse-Ergebnisse von Metriken werden von SonarQube Richtwerte bereitgestellt. Für jeden Verstoß gegen einen solchen Richtwert entsteht ein Issue. Diese werden nach Kategorie und Schwere sortiert. Für die folgenden Screenshots wurde eine Analyse von Apache Log4j in der Version 1.2.18-SNAPSHOT gefahren.

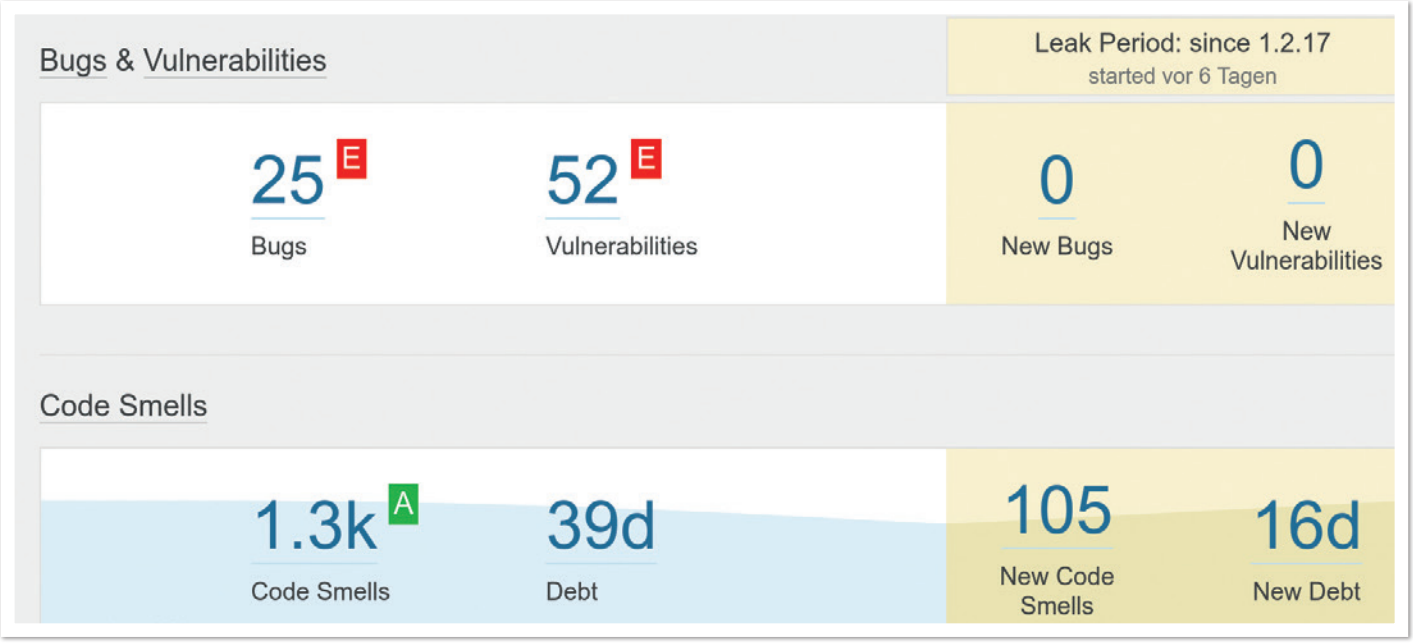

Abbildung 1: Die Kategorien

Kategorien

Bei der Analyse unterteilt SonarQube Metrik-Verstöße, „Issues" genannt, in drei Kategorien *(siehe Abbildung 1):*

- **Code Smells**
  Beispiele sind die zyklomatische Komplexität, als „Deprecated" markierter Code oder unnütze mathematische Funktionen, beispielsweise das Runden von Konstanten. Solche Issues deuten in den meisten Fällen auf tieferliegende Probleme hin. Ist beispielsweise die zyklomatische Komplexität einer Methode zu hoch, deutet das eventuell auf einen Design-Mangel in der Architektur hin.
- **Vulnerability**
  In dieser Kategorie landen Issues, bei denen es um Sicherheit geht. Damit ist nicht nur die Sicherheit in Form von SQL-Injection oder fest einprogrammierten Passwörtern gemeint, sondern auch die innere Sicherheit. Beispielsweise wird „public static fields should be constants" hier eingeordnet.
- **Bug**
  Darüber können klassische Handwerksfehler identifiziert werden, die beispielsweise der Java-Spezifikation widersprechen. So werden hier Issues wie der Vergleich von Klassen über ihren nicht voll qualifizierten Namen, unendliche Schleifen oder das Dereferenzieren von bekannten Null-Variablen verortet.

Schwere

Darüber hinaus werden Issues nach ihrer Schwere unterteilt. Diese reichen vom Entwicklungsstopp bis zum Hinweis am Rande *(siehe Abbildung 2):*

- **Blocker**
  In der schwersten Kategorie befinden sich Issues wie fest hinterlegte Passwörter. Wie der Name schon nahelegt, sollte kein Projekt mit derartigen Issues weiterentwickelt werden, ohne sie zu beseitigen.
- **Critical**
  Issues wie unbehandelte Exceptions oder andere, die Programmabläufe schwer beeinträchtigen können, werden mit „Critical" bewertet.
- **Major**
  Diese Issues gehören bereits in den Bereich „Codestil" oder „Konventionen" wie leere Code-Blöcke ohne erklärenden Kommentar, fehlende „@Override"-Annotationen oder auskommentierter Code.
- **Minor**
  Issues dieser Schwere sind syntaktisch korrekt, ihre Semantik lässt jedoch zu wünschen übrig. Dazu gehören unnötige Typumwandlungen, Duplikationen oder ungenutzte Rückgabewerte.
- **Info**
  Diese Schwere wird Issues zugeordnet, die irgendwann einmal behandelt werden sollten, wie Todo-Kommentare oder das Entfernen von „@Deprecated"-Code.

Automatische Interpretation

SonarQube nutzt ein Verfahren zur groben Bewertung der Code-Basis: die technische Schuld. Jedem Issue wird dabei eine Zeit zugeordnet, die benötigt wird, um ihn zu beheben. Die Summe dieser technischen Schuld wird dann im Verhältnis zum Gesamtaufwand des Projekts gestellt und ergibt das Maintainability Rating: „Main-

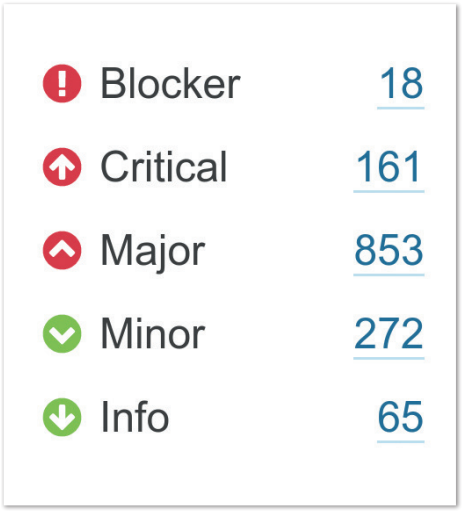

Abbildung 2: Die Schwere

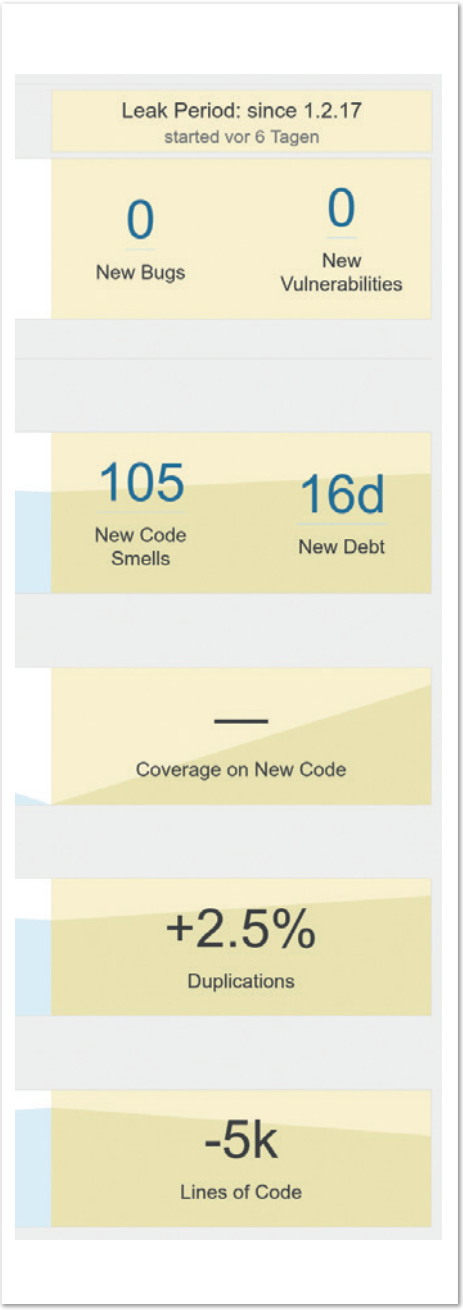

Abbildung 3: Entwicklung Projektgröße zu Duplikationen

tainability Rating = Technical Debt/Development Cost". Je nach Verhältnis dieser beiden Kennziffern fällt demnach das von SonarQube genannte Maintainability Rating aus:

- A: 0 – 0,1
- B: 0,11 – 0,2
- C: 0,21 – 0,5
- D: 0,51 – 1
- E: > 1

Als Mittelwert nimmt SonarQube an, dass pro Codezeile dreißig Minuten Entwicklungszeit benötigt wurden. Dies trifft natürlich lange nicht auf jede Codezeile zu, aber als grober Mittelwert sollte dies für große Projekte über die Zeit zutreffen. Eine Beispielrechnung: Bei einem kleinen Projekt von 2.500 Zeilen produziertem Code und fünfzig Tagen Entwicklungszeit wurde technische Schuld angesammelt. Bei einem typischen Arbeitstag von acht Stunden Länge können sechzehn Zeilen Code geschafft oder 0,0625 Tage pro Zeile benötigt werden. Dies führt zu folgender Rechnung: „50/(0,0625 * 2.500) = 0,32". Laut der obigen Tabelle ergibt das die Note C.

Bei genauerer Betrachtung des Bewertungsmaßstabs wird erkennbar, dass pro Zeiteinheit für die Entwicklung maximal zehn Prozent davon an technischer Schuld generiert werden muss, um schlechter als mit einer A-Note bewertet zu werden. Die Erfahrung lehrt, dass jedes Projekt, das groß genug ist, in der Gesamtbetrachtung genau diese Note erreicht. Das wirkt weniger erstaunlich, wenn man bedenkt, dass große Projekte lange laufen und über genau diesen langen Zeitraum hinweg im Durchschnitt viel durchschnittlich guter Code produziert wurde. Daher ist es interessanter, wenn das Maintainability-Rating nach den bereits bekannten Kategorien berechnet wird: „Bugs", „Vulnerability" und „Code Smells", wie *Abbildung 2* schon zeigt.

Manuelle Interpretation

Im ersten Schritt ergibt sich durch die Analyse eine eher unüberschaubar große Menge von Statistiken und Zahlen. Dabei ist es nicht damit getan, diese Statistiken zu akzeptieren und ihnen schlichtweg zu glauben. Der interessante Teil bei Code-Analyse ist, dieser durch Interpretation Bedeutung zu verleihen. Dabei gilt es, ein paar einfache Regeln zu beachten:

- **Relative Werte schlagen absolute**
  5.000 Issues im Projekt? Diese auf den ersten Blick hohe Zahl relativiert sich, wenn man weiß, dass es nur fünf pro Klasse sind.
- **Änderungen schlagen den Status quo**
  Ganz dem Prinzip des Vorwärtsdenkens entsprechend ist es interessanter, dass 5.000 Issues im letzten Release geschlossen werden konnten und nur 500 hinzukamen, als dass 4.500 noch offen sind. Die positive Entwicklung ist das Wichtige.
- **Metriken verstehen**
  Wenn eine Metrik nicht verstanden wurde, kann sie weder angewandt noch kann ihr Ergebnis interpretiert werden. Beispielsweise ist es schön zu wissen, was zyklomatische Komplexität ist. Wer das allerdings mit Lesbarkeit oder gar Wartbarkeit gleichsetzt, befindet sich auf dem Holzweg.
- **Metriken in Relation zueinander stellen**
  Wie der vorige Punkt schon andeutet, sagt eine Metrik allein wenig aus. Eine hohe zyklomatische Komplexität kann nämlich durchaus bedeuten, dass der betroffene Code schwer zu lesen ist. Allerdings gibt es auch noch Metriken wie die maximale Verschachtelungstiefe oder die Komplexität boolescher Ausdrücke, die da auch noch ein Wörtchen mitzureden haben.

Metriken verstehen

Die beliebteste, zugleich aber auch am häufigsten missbrauchte Metrik ist die der „Lines of Code" (LOC). Dass allerdings LOC nicht gleich LOC ist, zeigt ein kurzer Blick in gängige Nachschlagewerke:

- **Lines of Code (LOC)**
  Alle physisch existierenden Zeilen, also Kommentare, Klammern, Leerzeilen etc.
- **Source Lines of Code (SLOC)**
  Wie LOC, nur ohne Leerzeilen und Kommentare
- **Comment Lines of Code (CLOC)**
  Alle Kommentarzeilen
- **Non-Comment Lines of Code (NCLOC)**
  Alle Zeilen, die keine Leerzeilen, Kommentare, Klammen, Includes etc. sind

Wenn man nun die Größe eines Projekts beziffert, welche Kennziffer sollte herangezogen werden? Oder anders: Wie groß mag der Einfluss von unterschiedlichen Code-Styleguides auf die Anzahl der LOC und auf die Anzahl von NCLOC sein? SonarQube nutzt für seine Angaben über die Anzahl an Codezeilen NCLOC.

Metriken in Relation zueinander stellen

Listing 3 zeigt ein weiteres Mal Beispielcode. Die oben stehende Methode sucht in einer doppelt geschachtelten Liste nach einem Objekt mit einer gegebenen ID. Nach McCabe besitzt sie eine zyklomatische Komplexität von fünf, alles im Rahmen. Allerdings besitzt sie gleichzeitig eine maximale Verschachtelungstiefe von vier. SonarQube definiert ein Maximum von drei, also besitzt diese Methode ihren ersten Issue. Hätten wir nur McCabe betrachtet, wäre sie als grün durchgegangen und im Bericht nicht aufgefallen. Ein Early Return später haben wir das Problem gelöst *(siehe Listing 4)*.

Die Metriken sind zufrieden – wir auch? Es gibt einen Grund, warum sie zur Kategorie „Code Smell" gehören: Hier wurde sehr wahrscheinlich beim Architektur-Design versagt. Warum wurde eine doppelt geschachtelte Liste in die Methode hineingereicht? Arbeitet der Code absichtlich auf dieser geringen Abstraktionsebene? Wurde vielleicht am Domänen-Design vorbei gearbeitet? Warum wurde nicht gleich das passende DTO aus der Datenquelle abgefragt, statt eine Methode mit hoher Laufzeit-Komplexität anzustoßen?

```
DTO search(List<List<DTO>> rawData, int id) {
  if(rawData != null) {
    for(List<DTO> sublist : rawData) {
      for(DTO dto : sublist) {
        if(dto.getId() == id) {
          return dto;
        }
      }
    }
  }
  return null;
}
```

Listing 3

```
DTO search(List<List<DTO>> rawData, int id) {
  if(rawData == null) {
    return null;
  }

  for(List<DTO> sublist : rawData) {
    for(DTO dto : sublist) {
      if(dto.getId() == id) {
        return dto;
      }
    }
  }
  return null;
}
```

Listing 4

Viele sagen auch, dass das DTO-Pattern ein Antipattern ist. Diese Fragen legen den Schluss nahe, dass der eigentliche Fehler nicht in der einzelnen Methode liegt, sondern auf einer höheren Ebene zu suchen ist.

Als weiterer Punkt wurde oben schon die Entwicklung genannt: Änderungen schlagen den Status quo. SonarQube erstellt Historien, die sich auch wunderbar auswerten lassen. Beispielsweise kann man die Entwicklung der Projektgröße mit der Anzahl der Duplikationen vergleichen *(siehe Abbildung 3)*.

Im selben Zeitraum, in dem 5.000 Zeilen Code entfernt wurden, kamen 2,5 Prozent Code-Duplikationen hinzu. Dies kann bedeuten, dass redundanter Code geschaffen oder nicht redundanter Code gelöscht wurde, sodass Duplikationen schwerer wiegen konnten. Wird dann noch in Betracht gezogen, dass das Projekt insgesamt nur 16.000 Zeilen Code besitzt, ist Letzteres wahrscheinlicher.

Erfahrungswerte

Eine ausreichend große Code-Basis vorausgesetzt, erreichen viele Legacy-Projekte ein A-Rating. Wie bereits gesagt, ist das wenig verwunderlich. Die Aufteilung nach Issue-Kategorien hilft an dieser Stelle enorm.

Im Embedded-Bereich gibt es den Grundsatz, dass man pro dreißig Regelverstöße drei kleinere und einen schwerwiegenden Bug erwarten kann. Diese Regel hilft, aus der Anzahl der Issues während der Entwicklung auf die Bugs während des Betriebs zu schließen. Denn meistens fällt es schwer, eine direkte Verbindung zwischen diesen Kennziffern herzustellen: In manchen prognostizierten „NullPointer" laufen Benutzer beispielweise einfach nicht hinein, weil die Applikation gar nicht auf die nötige Weise bedient wird. Allerdings helfen auch hier Mittelwerte wie die oben genannten. Die reinen Zahlenwerte müssen sicherlich fein abgestimmt sein, um auf den jeweiligen Einsatzbereich zu passen, das Muster sollte jedoch stimmen.

Live mit SonarQube zu arbeiten, kann dazu führen, dass Issues entweder sofort behoben werden oder versucht wird, sie gleich im Vorfeld zu vermeiden. Diese durchaus lobenswerte Neigung kann allerdings dazu führen, dass zu komplex gedacht und das eigentliche Ziel aus den Augen verloren wird. Dieses Verhaltensmuster hat einen Namen: „Over Engineering". Empfehlenswerter ist es, gemeinsam zum Sprint-Ende oder an vergleichbaren Terminen zu sichten, wie sich die Codebasis entwickelt.

Fazit

Statische Code-Analyse kann eine gute Grundlage für die Sicherstellung von Code-Qualität sein. Allerdings ist ebenso die Fähigkeit der Interpretation notwendig, damit die Code-Analyse die vollständige Wirkungskraft entfalten kann. So sind Statistiken zwar interessant, jedoch ist eine statische Code-Analyse erst dann wirklich vollständig, wenn ein Mindestmaß an Interpretation hineingeflossen ist.

So liefert Code-Analyse in erster Linie ein Gefühl für die Code-Basis. Erst so können fundierte Aussagen darüber getroffen werden, welche Bereiche des Projekts besonders gefährdet, instabil oder renovierungsbedürftig sind.

Für die Teams können regelmäßige Analysen die Team-Motivation erhöhen. Eine positive Issue-Bilanz am Ende eines Sprints und aufwärtszeigende Historien-Graphen sind gute Treiber für ein Entwickler-Team und Beweis der eigenen Leistung. Darüber hinaus können Analyse-Ergebnisse als Argumentationsgrundlage gegenüber dem Kunden dienen. Mithilfe der Projekt-Historie, die eine Auswahl gut darstellbarer Kennzahlen beinhaltet, kann vor Kunden oder Entscheidern besser über ein eventuell nötiges technisches Release diskutiert werden.

Josha von Gizycki
josha.von.gizycki@triology.de

Josha von Gizycki arbeitet als Software-Entwickler bei der TRIOLOGY GmbH in Braunschweig und hat sich auf die Entwicklung von Enterprise-Applikationen im Java- und Oracle- Umfeld spezialisiert. Sein Schwerpunkt liegt dabei in den Bereichen „Wartung" und „System-Design".