Samuel Eickelberg
(samuel.eickelberg@gmail.com)
ist Softwareentwickler, Tester und IT-Berater aus Berlin und bei der ANYBET GmbH als Quality Assurance Engineer tätig. Er studierte Informatik an der Freien Universität Berlin und befasst sich unter anderem mit neuen Teststrategien und mit Codequalität.

Das Java Architecture Testing Framework
Ein Framework für automatisierte, mit Unit-Tests kombinierbare Architekturtests

Automatisierte Tests konzentrieren sich auf die Funktionsweise von Softwarekomponenten und deren Zusammenspiel. Neu ist jedoch das automatisierte Testen von Abhängigkeiten, Entwurfsmustern, Programmierparadigmen und Codemetriken. Hier ist der Begriff des Architekturtests prägend als Oberbegriff für solche Arten von Tests. Dieser Artikel zeigt unterschiedliche Dimensionen von Architekturtests auf und stellt ein Framework sowie Strategien vor, Architekturtests in Softwareprojekte zu integrieren.

Softwarearchitektur

Die Softwarearchitektur ist das Gesamtkonstrukt des entworfenen Softwareprodukts. Martin Fowler schreibt dazu in [Fowl03], dass Architektur ein Begriff sei, „den zahlreiche Leute zu definieren versuchen, ohne letztlich einen Konsens zu erzielen. Es gibt jedoch zwei gemeinsame Elemente: Erstens: Es geht um die Zerlegung eines Systems in Komponenten auf höchster Ebene. Zweitens: Es geht um Entscheidungen, die sich nur schwer ändern lassen." Darüber hinaus befindet er, „dass es nicht nur eine Architektur eines Systems gibt, sondern dass ein System mehrere Architekturen enthalten kann".

Eine Softwarearchitektur weist also Merkmale auf, die für das Zusammenspiel der Softwarekomponenten von Bedeutung sind. Würden solche Architekturmerkmale zerstört, funktioniert die Softwarekom-

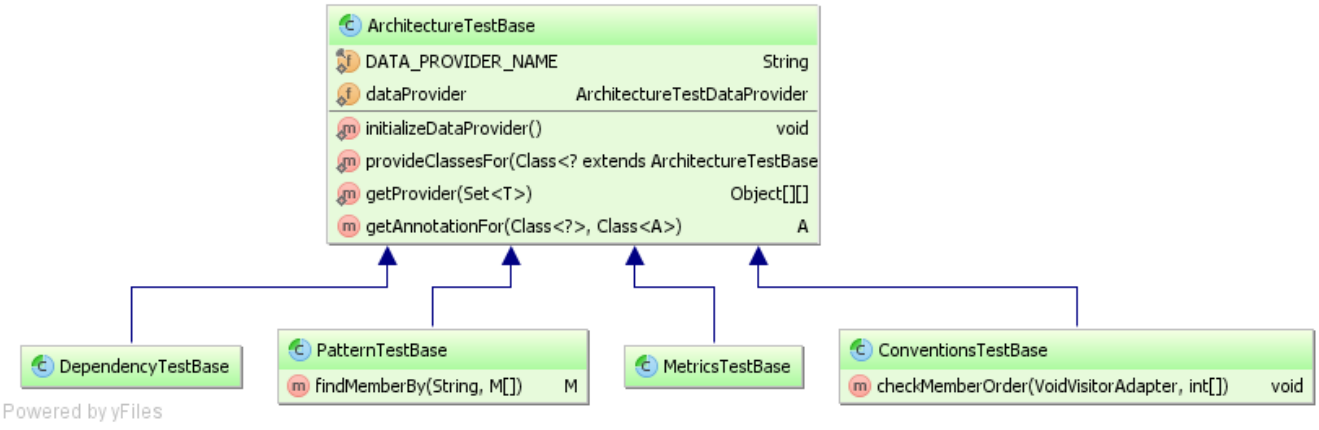

Abb. 1: Basisklassen der Architekturtests

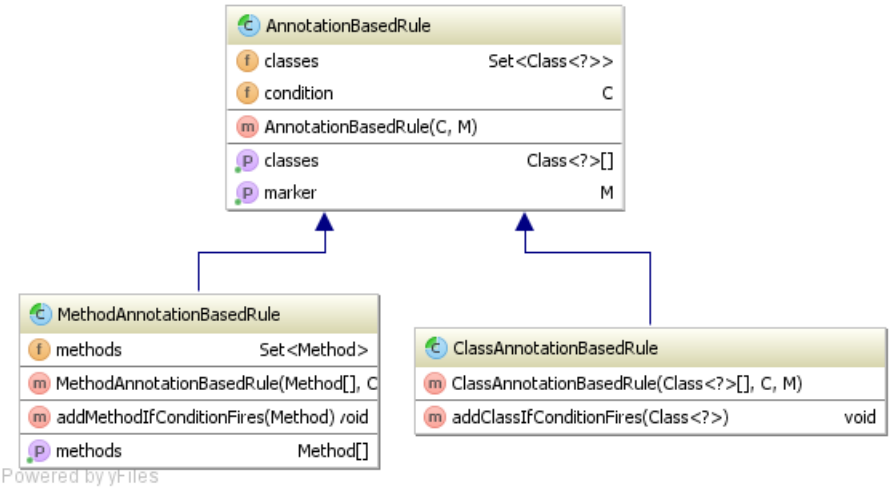

Abb. 2: Basisklassen für die Testregeln

ponente nicht mehr. Es ist daher entscheidend, welche Merkmale von Architekturen für automatisierbare Tests herangezogen werden. Architekturtests sind statische Unit-Tests und kommen ohne Mock-Testdaten und ohne andere Komponenten, wie etwa Drittsysteme oder Datenbanken, aus. Das Reporting der Testergebnisse kann in der Continuous-Integration-Umgebung erfolgen.

Im Framework adressierte Architekturtests

Es gibt unterschiedliche Arten von Codemetriken, die in der Literatur auch Softwaremetriken genannt werden. Sie dienen dazu, bestimmte Qualitätskriterien an geschriebenem Code zu überprüfen. Architekturtests für Codemetriken erfordern für jede zu überprüfende Metrik bestimmte Grenzwerte, gegen die sie die ermittelten Ist-Werte der Metrik vergleichen. So nennt Robert C. Martin zwei Codemetriken, das Afferent Coupling (Ca) und das Efferent Coupling (Ce) [Mart94].

Afferente Kopplung ist die Anzahl der Referenzierungen anderer Komponenten im Projekt in die Komponente hinein. *Efferente Kopplung* ist die Anzahl der Referenzierungen aus der Komponente heraus zu anderen Komponenten im Projekt. Beide Kennzahlen lassen sich trivial durch das Zählen der Import-Statements (in Java) beziehungsweise der Using-Direktiven (in C#) in jeder Klasse des betrachteten Pakets ermitteln. Voraussetzung hierfür ist, dass nicht genutzte Import-Statements vom Code entfernt werden, um die Verlässlichkeit eines solchen Architekturtests sicherzustellen.

Des Weiteren wird die *Zyklomatische Komplexität nach McCabe (MCC)* als Kriterium für die Anzahl möglicher Programmflusszweige herangezogen. Eine hohe McCabe-Zahl deutet auf schwieriger wart- und damit testbaren Code hin, bei dem es sich lohnen kann, ihn in mehrere (Methoden-)Fragmente zu zerlegen [McCa76].

Die *Halstead-Metrik* wurde 1977 von Maurice Howard Halstead vorgestellt und misst die Komplexität von Software basierend auf der Annahme, dass Programmteile aus Operationen und Operanden aufgebaut sind. Gerade für Java-Code lässt sich das Verfahren leicht berechnen und automatisieren. Es liefert unter anderem eine Aussage darüber, wie viele Bugs in geschriebenem Code zu erwarten sind [Hals77].

In der Mathematik wird unter *Methodenreinheit* (engl. method purity) verstanden, dass für die Lösung eines Problems keine fremden Konzepte herangezogen werden sollen. Abgewandelt für die Softwareentwicklung bedeutet dies, dass eine Methode dann als *rein* gilt, wenn sie ihre Umgebung möglichst nicht oder in einem zu definierenden Rahmen minimal beeinflusst. Eine Methode ist also rein, wenn sie:

- ein von ihr instanziiertes Objekt verändert,
- ihren Rückgabewert verändert und
- andere reine Methoden aufruft.

Unrein ist eine Methode, wenn sie:

- ein (statisches) Klassenmitglied verändert,
- Parameter entgegennimmt, die von ihr verändert werden, oder
- andere unreine Methoden aufruft.

Im Framework lautet der Test, der die Methoden einer Klasse auf ihre Reinheit hin untersucht, `MethodPurityTest`.

Tests für Entwurfsmuster achten darauf, dass im Code Merkmale des zu findenden Entwurfsmusters existieren. Es gibt einen interessanten Ansatz, der Entwurfsmuster in bestehendem Code mittels Softwaremetriken und maschinellem Lernen untersucht: Satoru Uchiyama und andere erläutern, dass viele automatisierte Ansätze, die auf statischer Codeanalyse beruhen, Probleme haben, Entwurfsmuster, deren Implementierung ähnlich aussieht, zu unterscheiden [Uchi14]. Darüber hinaus ist es schwierig, Entwurfsmuster in ihrer Anwendungsvielfalt zu unterscheiden. In dem hier vorgestellten Framework werden Tests für die fünf Entwurfsmuster aus [Uchi14] implementiert: *Singleton*, *Adapter*, *Template Method*, *State* und *Strategy*. Für diese Muster existieren im Framework stark vereinfachte Architekturtests, die auf bestimmte Merkmale im Code achten und dabei in konkreten Anwendungsfällen falsche Positive produzieren können.

Architekturtests, die Abhängigkeiten von Komponenten untereinander prüfen, achten darauf, welche anderen Komponenten innerhalb einer bestimmten Komponente wie verwendet werden. Konkret geht es um Verwendung, Vererbung und Implementierung. Die betreffenden Tests heißen dementsprechend `UsesTest`, `ExtendsTest` und `ImplementsTest`. Zu jedem dieser Tests muss in entsprechenden Annotationen, `@MustUse`, `@MustNotUse`, `@MustExtend`, `@MustNotExtend`, `@MustImplement` beziehungsweise `@MustNotImplement`, angegeben werden, welche anderen Klassen erforderlich sind und welche nicht enthalten sein dürfen.

Bei Programmierrichtlinien (Coding Conventions/Guidelines) wird im Wesentlichen angenommen, dass Namenskonventionen bestehen und die Reihenfolge von Sichtbarkeiten (`private`, `protected`, `public`) in Klassen eingehalten wird. Außerdem gilt für lokale Variablen, dass sie nicht mit einem `final`-Modifier versehen werden sollten (das macht den Code unnötig unleserlich und der Compiler optimiert das für die JVM ohnehin von sich aus). Innere Klassen sind oft ein Anzeichen dafür, dass eine Klasse mehr als ihren ihr zugedachten Aufgabenbereich erfüllen soll (siehe *Single Responsibility Principle, SRP*), sodass es besser ist, solche inneren Klassen als eigene Klassen zu definieren und so den Code wartbarer zu halten. Die Tests hierfür sind `CamelCaseTest`,

NoFinalModifierInLocalVariablesTest, NoInnerClassesTest, OrderingTest und OverlyChainedMethodCallsTest.

Das hier vorgestellte "Java Architecture Testing Framework" [Eick15,JATF] ist ein Framework für die Realisierung statischer Codeanalyse innerhalb dynamischer Tests, mit den einhergehenden Vorteilen beider „Welten". Sowohl dynamische Tests als auch statische Codeanalyse haben die Verbesserung des Entwicklungsprozesses zum Ziel, wollen also, dass die gefundenen Defekte und Fehler behoben werden. Dynamische Tests, die Architekturmerkmale am Quellcode statisch prüfen sollen, benötigen Zugriff auf ebendiesen Quellcode. Das unterscheidet Architekturtests von anderen dynamischen Tests grundlegend.

Allerdings darf nicht unerwähnt bleiben, dass weder statische Codeanalyse noch Architekturtests Code Reviews ersetzen wollen oder können. Sie können diese jedoch unterstützen, beispielsweise durch Testausführung vor und nach dem Review zur Untersuchung der Verbesserungen.

Spillner und Linz schreiben, dass statische Codeanalyse bereits vor einem Review hilft, die Menge der zu betrachtenden Aspekte im Review zu verringern [Spil05]: „Grundsätzlich sollte die statische Analyse durchgeführt werden, auch wenn kein Review geplant ist. Jede entdeckte Unstimmigkeit erhöht die Qualität des analysierten Dokuments." Es ist daher angeraten, Architekturtests, die eine statische Analyse durchführen, regelmäßig und so oft wie möglich ausführen zu lassen. „Mit der statischen Analyse lassen sich allerdings nicht alle Fehler und Unstimmigkeiten nachweisen. Es gibt Fehler oder präziser Fehlerzustände, die erst bei Ausführung, also zur Laufzeit des Programms, zur Wirkung kommen und vorher nicht ermittelbar sind." Aus diesem Grund *ergänzen* Architekturtests stets nur andere Tests, vor allem andere dynamische Tests, die die Funktionalität und das Zusammenwirken von Komponenten untersuchen, sie *ersetzen* sie jedoch nicht.

Zusammengefasst bedeutet dies für Architekturtests, dass sie die folgenden Eigenschaften und Ziele haben:

- Architekturtests betreiben statische Codeanalyse; sie sind *Codeprüfer*.
- Architekturtests haben als statische Analysetools auch dieselben Ziele wie die statische Codeanalyse.
- Architekturtests sind implementiert als spezielle dynamische Tests, die den Quellcode benötigen.
- Architekturtests sind Unit-Tests, die jede zu testende Klasse separat untersuchen.
- Architekturtests ersetzen kein Code-Review, sie unterstützen es, genauso wie dies statische Codeanalysewerkzeuge tun.
- Architekturtests sind als Tests, zum Beispiel auf der Grundlage von JUnit, in derselben Sprache (hier Java) geschrieben wie andere dynamische Tests – sie sind daher transparent allen Entwicklern zugänglich und durch sie auch wartbar.

Die Basisklassen, von denen die Architekturtests im Framework erben, entsprechen den vier Arten von Tests. Diese wiederum beerben ArchitectureTestBase (siehe Abbildung 1), welche die gemeinsame Funktionalität für alle Tests bereitstellt. Zu testende Klassen, die den Architekturtests unterzogen werden sollen, werden mit @ArchitectureTest annotiert. Sie erwartet vier Parameter:

- boolean omitMetrics() default false;
- boolean omitConventions() default false;
- Pattern[] patterns() default {undefined};
- Dependency[] dependencies() default {none};

Regeln für die Formulierung von Architekturtests

Die Pflege der Annotationen bei sehr vielen Klassen, die den Architekturtests unterzogen werden sollen, ist für sich genommen bereits ein gewisser Aufwand. Außerdem entstand während der Implementierung die immer offenkundigere Anforderung, Klassen nach bestimmten Regeln automatisch, also unabhängig davon, ob sie annotiert sind oder nicht, Architekturtests zuzuführen. Dies macht den Entwurf und die Implementierung einer Programmierschnittstelle für diese Regeln, im Folgenden *Rule-API* genannt, erforderlich.

Ausgehend von den bestehenden Annotationen besteht zwischen Annotation *A* und Regel *R* eine *1:n-Relation*, das heißt, zu einer Annotation kann es *n* Regeln geben. Innerhalb dieser Regeln gibt es eine Bedingung *C*, die bestimmt, wann die Regel zutreffen soll. Die Regeln werden vor den eigentlichen Architekturtests ausgeführt, also wenn bestimmt wird, welche Klassen welchen Tests zugeführt werden.

Die Regeln haben eine höhere Priorität als die Annotationen, sodass bei eventuellen Kollisionen die Regel gewinnt. Ein Bei-

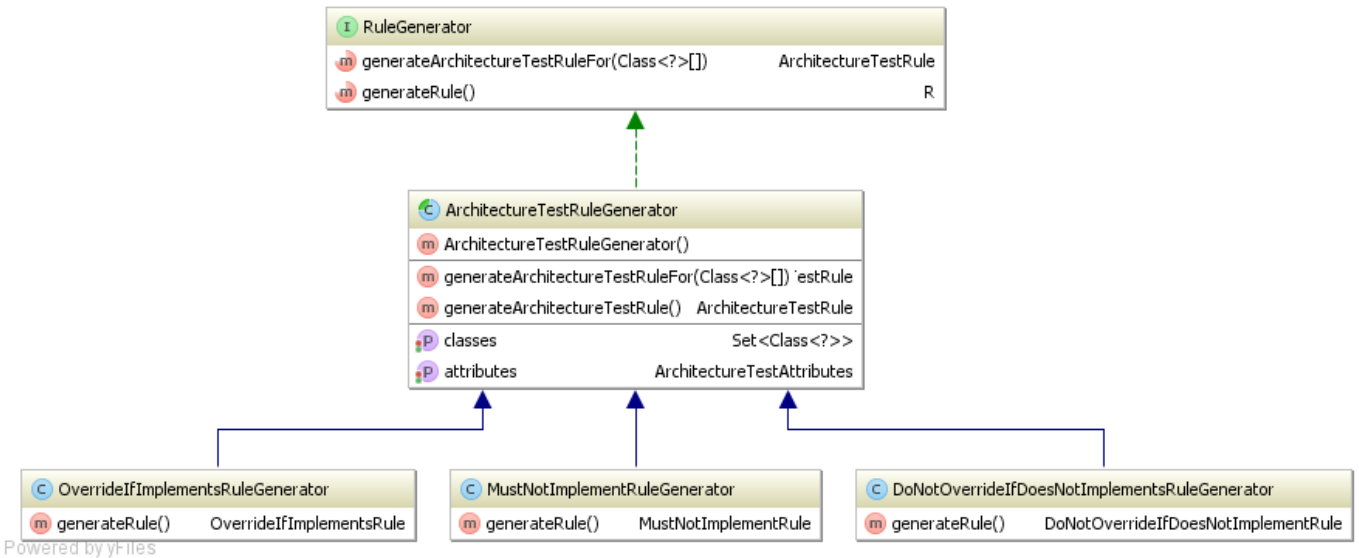

Abb. 3: Regelgenerierer

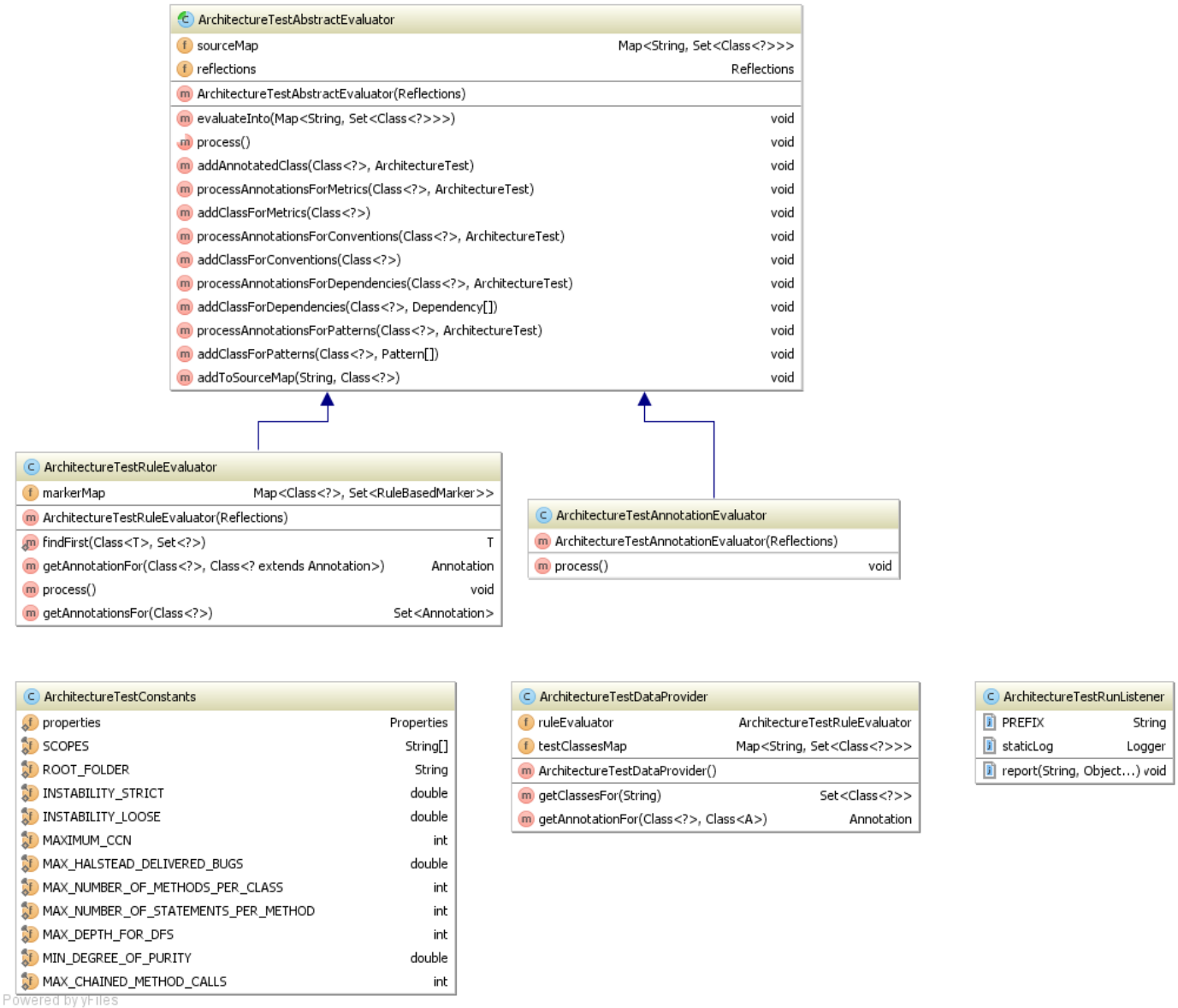

Abb. 4: API-Klassen des Frameworks

spiel: Definiert sei eine Regel, die für alle Klassen, die Cloneable implementieren, vorgibt, dass sie nicht Serializable implementieren dürfen. Eine der Klassen ist jedoch wie folgt annotiert:

```
@ArchitectureTest(dependency =
 {implementsDependency})
@MustImplement(interfaces =
 {Serializable.class}
public class
SomeClassThatImplementsCloneable
   implements Cloneable, Serializable {
 ...
}
```

Die Regel würde also bewirken, dass die @MustImplement-Annotation wirkungslos bleibt und in der Folge der Test fehlschlägt, da die zu testende Klasse im Beispiel Cloneable implementiert, obwohl sie auch Serializable implementiert und dies gemäß Regel nicht darf.

Es gibt Regeln für Methodenannotationen und Regeln für Klassenannotationen (siehe Abbildung 2). Methodenannotationen sind im Framework @MustBePure, @MustThrow und @MustNotThrow. Klassenannotationen sind @ArchitectureTest oder zum Beispiel auch @MustExtend und @MustImplement.

Bedingungen für Regeln bestimmen, wann eine Regel feuert. Beispiele für solche Bedingungen sind AlwaysTrue (wenn eine Regel immer zutrifft), Implements (wenn eine Klasse ein bestimmtes Interface implementiert), Extends (wenn eine Klasse von einer bestimmten Klasse erbt) und ClassNamePatternMatches (wenn ein Klassenname einem bestimmten Muster entspricht). Die logische Verknüpfung mehrerer Regeln erfolgt mittels der von CombinedCondition erbenden Bedingungen And beziehungsweise Or. Es gibt noch mehr Bedingungen im Framework, welches selbst auch durch weitere Bedingungen, Regeln und Tests erweiterbar ist.

Regelgenerierer (siehe Abbildung 3) werden außerhalb des Frameworks geschrieben, sie erben aber von ArchitectureTest-RuleGenerator, damit sie vom ArchitectureTestRuleEvaluator erfasst werden. Sie generieren die eigentlichen Regeln zur Laufzeit.

Die Programmierschnittstelle des Frameworks

Abbildung 4 zeigt die API-Klassen des Frameworks. Zur Einbindung des Frameworks und damit zum Testen des Quellcodes im Projekt muss die *ArchitectureTestConstants*-Instanz sämtliche Werte erhalten, die als Constraints für die Tests

erforderlich sind. Zweitens muss es eine TestSuite innerhalb des Projekts geben, welche die gewünschten Architekturtests aufruft. Die Oberklasse der Architekturtests enthält eine mit @BeforeClass annotierte Methode, in der alle Initialisierungen, Annotations- und Regelauswertungen erfolgen.

Fazit

Der Einsatz des Frameworks kann schleichenden Architekturverlust zwar nicht verhindern, ihn jedoch verlangsamen. Es unterstützt Entwickler dabei, Architektur- und Designentscheidungen einzuhalten durch die regelmäßige Ausführung der Tests (beispielsweise kombiniert mit täglichen Unit-Tests). Dafür sind keine Spezialkenntnisse über statische Codeanalysetools erforderlich; die Offenheit des Frameworks ermöglicht Entwicklern das eigenständige Erweitern der Architekturtests.

Es ist entstanden im Rahmen einer Masterarbeit im Fachgebiet Software Engineering bei der ANYBET GmbH. Die ANYBET GmbH ist ein IT-Dienstleistungsunternehmen und Teil der mybet-Unternehmensgruppe (www.mybet.de). Die mybet SE ist ein Anbieter von Sportwetten und Online-Spielen wie Casino und Poker. Das Framework wird derzeit bei der ANYBET GmbH produktiv eingesetzt und führt in der Continuous-Integration-Umgebung regelmäßig mehrere tausend Tests auf dem Produktionscode aus, Tendenz steigend.

Verfügbar ist das Framework unter der GNU GPLv3 unter [JATF]. Beiträge, Anregungen sowie Mitarbeit daran sind dem Autor herzlich willkommen. ■

Literatur & Links

[Eick15] S. Eickelberg, Ein Framework für automatisierte Architekturtests, Masterarbeit, Institut für Informatik, Freie Universität Berlin, 2015

[Fowl03] M. Fowler, Patterns für Enterprise Application-Architekturen, mitp-Verlag, 2003

[Hals77] M. H. Halstead, Elements of software science, Elsevier, 1977

[JATF] Java Architecture Testing Framework, siehe: https://github.com/Duke2k/jatf

[Mart94] R. C. Martin, OO Design Quality Metrics. An Analysis of Dependencies, Green Oaks, Illinois, USA, 1994

[McCa76] T. McCabe, A Complexity Measure, in: IEEE Transactions on Software Engineering, Band SE-2, Nr. 4, 1976

[Spil05] A. Spillner et al., Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified Tester, dpunkt-Verlag, 2005

[Uchi14] S. Uchiyama et al., Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning, in: Journal of Software Engineering and Applications, Band 7/2014, Seiten 983-998, Scientific Research Publishing, 2014