Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Johannes Zink
BACHELOR THESIS

Erweiterung von jqwik

Eingereicht am 12. Juli 2021

Betreuer:
Prof. Dr. Dirk Riehle, M.B.A.
Dipl.-Inf.-Med. Johannes Link
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Erlangen, 12. Juli 2021

License

_____

Erlangen, 12. Juli 2021

Abstract

Property-Based Testing enables the user to generate test cases automatically. The JUnit 5 based engine jqwik makes Property-Based Testing for the Java programming language easy. However, until now it is not possible to generate test cases for all data types built into the JDK.
This paper will introduce the basics of testing, before creating insights about the most important features of Property-Based Testing, demonstrating them using jqwik. In the next step, the engine jqwik will be extended to allow the generation of email addresses as well as dates and times. Finally, the implemented code will be evaluated in comparison to the requirements set in the paper.

Zusammenfassung

Property-Based Testing ermöglicht dem Anwender die automatische Generierung von Testfällen. Die auf JUnit 5 basierende Engine jqwik macht Property-Based Testing für die Programmiersprache Java einfach, allerdings kann sie im Moment noch nicht für alle im JDK eingebauten Datentypen Testfälle erzeugen.
In dieser Arbeit soll erst auf die Grundlagen des Testens eingegangen werden, anschließend die wichtigsten Eigenschaften des Property-Based Testings verstanden und anhand von jqwik aufgezeigt werden und nachfolgend die Engine jqwik um die Generierung von E-Mail-Adressen sowie Daten und Uhrzeiten erweitert werden. Zum Schluss soll noch eine Bewertung des umgesetzten Codes im Vergleich zu den in der Arbeit gestellten Anforderungen abgegeben werden.

Inhaltsverzeichnis

Begriffsklärung

Build-Werkzeug: Automatisiert alle notwendigen Schritte, die zum Erstellen einer lauffähigen Software durchgeführt werden müssen (Macke, 2017)

C++: Programmiersprache

Engine: „Programmteile […], die grundlegende Aufgaben im Rahmen eines Anwendungsprogramms erfüllen“ (Computerlexikon.com, 1998)

FAQ: Frequently Asked Questions: Eine Sammlung, die häufig gestellte Fragen beantwortet (Peters, 2015)

Haskell: Programmiersprache

IDE: Integrated Development Environment (Integrierte Entwicklungsumgebung): „Stellt Programmierern eine Sammlung der wichtigsten Werkzeuge zur Softwareentwicklung unter einer Oberfläche zur Verfügung“ (Luber & Augsten, 2017)

Invariante: „Zusicherung […], die immer erfüllt bleibt“ (Zachmann, 2010)

ISO: „Norm[en] der International Organization of Standardization” (Tüv Nord, 2016)

Java: Programmiersprache

JAR: Java Archive: Besteht aus mehreren Java-Dateien und Metadaten die komprimiert und zusammengefasst werden (Ionos, 2020, 7. Mai)

Kfz: Kraftfahrzeug (Kraftfahrt-Bundesamt, o. D.)

PHP: Hypertext Preprocessor: Programmiersprache (PHP, o. D.)

Proprietäre Software: „Kommerzielle Software[,] Shareware [und] Freeware [ohne] Einblick in [den] Quellcode“ (Picot & Fiedler, 2008)

SmallTalk: Programmiersprache

SUnit: Ein Framework für Tests in SmallTalk (GNU, o. D.)

1 Grundlagen

Jeder, der Software entwickelt, kennt es: Man will eine möglichst fehlerfreie und gut funktionierende Software schreiben. Doch wie findet man heraus, ob die entwickelte Software auch den zuvor gestellten Anforderungen entspricht? Ganz einfach: Indem man sie testet. Mittlerweile ist das Testen aus dem Softwareentwicklungsprozess gar nicht mehr wegzudenken und selbst im bekannten V-Modell ist das Testen von Software fest verankert. (Ionos, 2020, 28. September) Nur so kann man möglichst viele Fehler in der Programmierung finden und diese dann beheben. Dabei gibt es ein großes Spektrum an Möglichkeiten, um Software zu testen. Das Einfachste ist dabei, Testdaten per Hand einzugeben. Es gibt jedoch auch weit effizientere Maßnahmen. So können Tests auch automatisiert durchgeführt werden. Dies ist vor allem sinnvoll, wenn sich Tests ständig wiederholen oder man an einem größeren Projekt arbeitet. (Orlov, 2019)

1.1 Der Softwareentwicklungsprozess

Hinter der Entwicklung einer Software steckt ein aufwendiger Prozess, der aus mehreren Phasen besteht. Dabei kann man verschiedene Entwicklungsmodelle anwenden, sodass nach und nach eine Software entsteht. (simplifier, 2017) Im Folgenden sollen die angewandten Phasen und Entwicklungsmodelle erklärt und anschließend auf den Entwicklungsprozess bei Open-Source-Software eingegangen werden.

1.1.1 Tätigkeiten

In dieser Arbeit wurden bei der Entwicklung der Software die folgenden Tätigkeiten ausgeführt, die im Anschluss erklärt werden: Zuerst wurde eine Anforderungsanalyse durchgeführt, anschließend wurde das Design für die zu entstehende Software erstellt, woraufhin die Software implementiert und schließlich getestet wurde.

1.1.1.1 Anforderungsanalyse

Die Anforderungsanalyse ist ein wichtiger Bestandteil der Softwareentwicklung. Das Ziel dabei ist es, aus den Bedürfnissen des Auftraggebers die Softwareanforderungen abzuleiten. (Avci & Wagner, 2005) Man unterscheidet dabei zwischen funktionalen und nicht-funktionalen Anforderungen. Während funktionale Anforderungen das beschreiben, was das Programm leisten soll, bestimmen nicht-funktionale Anforderungen beispielsweise technische Bedingungen wie etwa die Laufzeit der Anwendung oder ihre Wartbarkeit. (Disterer & Rose, 2009) So soll sichergestellt werden, dass die Anwendung nach Fertigstellung auch wie vom Auftraggeber gewünscht funktioniert.

1.1.1.2 Design

Ein essenzieller Schritt in der Softwareentwicklung ist das Erstellen eines geeigneten Designs. (Arab, Bourhnane & Kafou, 2018) Das Ganze kann in zwei Phasen unterteilt werden: Die Erstellung einer Architektur, sowie das Kreieren des technischen Designs. Während die Architektur die grundlegende Struktur des Programms festlegt, zeigt das technische Design, wie die Architektur genau umgesetzt werden soll. (doubleSlash, o. D.) Um ein gutes Design zu erreichen sind in den einzelnen Phasen oft mehrere Entwicklungsschritte nötig. Eine Möglichkeit das Design darzustellen ist z. B. mittels sogenannter UML-Diagramme, die Klassen, ihre Attribute und ihre Methoden sowie ihre Beziehungen zueinander in einem Diagramm darstellen. Je genauer das Design erstellt wird, desto einfacher ist es am Ende den passenden Code zu

ergänzen. Der Zeitaufwand, den man in diesen Schritt steckt, lohnt sich also. (Arab, Bourhnane & Kafou, 2018)

1.1.1.3 Implementierung

Bei der Implementierung setzt man das zuvor Geplante um. Es entsteht dabei der zugehörige Quellcode. Umso ausführlicher der Plan aus den vorherigen Schritten ist, desto einfacher ist es in diesem Schritt den zugehörigen Code zu schreiben. (chrissikraus & Augsten, 2020)

1.1.1.4 Test

Zum Schluss wird die entstandene Software getestet. Das Ziel dabei ist es, möglichst viele der vorhandenen Fehler zu finden und anschließend zu beheben. Dadurch steigt die Chance, dass das Programm auch in den nicht getesteten Fällen korrekt funktioniert. Aufgrund der Vielzahl der möglichen Eingabedaten ist es meist nicht möglich, ein Programm vollständig zu testen. Um die Korrektheit der Software zu beweisen, muss das zu erwartende Ergebnis des Programms vorher definiert sein, damit ein Vergleich gemacht werden kann. Man findet dabei nur die Auswirkungen des Fehlers, jedoch nicht die genaue Ursache selbst. Vor dem Testen entscheidet man sich zudem für eines von mehreren möglichen Testverfahren. (Glinz & Fritz, 2006)

1.1.2 Angewandte Softwareentwicklungsmodelle

Es gibt viele verschiedene Ansätze, wie man Software entwickeln kann. In dieser Arbeit wurden die iterative sowie die inkrementelle Entwicklung angewandt, die nachfolgend erklärt werden sollen.

1.1.2.1 Iterative Entwicklung

Bei der iterativen Entwicklung fängt man damit an, einen Bruchteil des Problems zu lösen und diese Lösung immer weiter zu ergänzen oder zu verfeinern bis schließlich das gesamte System implementiert ist. Dabei kann in einem Schritt sogar die gesamte Konstruktion geändert werden. Jede der einzelnen Entwicklungsphasen besteht dabei aus der Auswahl der Aufgaben für den nächsten Schritt, welche danach in einem Design entworfen und anschließend implementiert werden. Dieser Prozess wird so lange wiederholt, bis es keine übriggebliebenen Aufgaben mehr gibt. (Basil & Turner, 1975) Ein wesentlicher Vorteil dieser Methode ist, dass das Projekt nicht in einem großen Schritt, sondern in mehreren kleinen Schritten entwickelt wird. So kann der Abnehmer der Software immer wieder prüfen, ob sich das Projekt in die richtige Richtung entwickelt. Auch die Qualität des Programms kann so mehrfach kontrolliert werden. Fehler können so frühzeitig erkannt und behoben werden, wodurch das Risiko, dass das Projekt aufgrund von Mängeln scheitert, sinkt. Da die Anforderungen ständig überprüft werden, steigt zudem die Qualität der entstehenden Software. (form4, o. D.)

1.1.2.2 Inkrementelle Entwicklung

Wenn eine Software inkrementell entwickelt wird, dann bedeutet das, dass die Software nach und nach in Teilen entwickelt wird. Dabei hat jeder dieser Teile eine eigene vollständige Funktionalität. Es wird dabei davon ausgegangen, dass der bereits geschriebene Code nicht mehr überarbeitet werden muss. Egal wie groß das Softwareprojekt ist, wird dabei die Arbeit an einem Inkrement immer erst fertiggestellt, bevor ein neues Inkrement begonnen wird. Dies beinhaltet sowohl das Schreiben von Code als auch das anschließende Testen.

Der Unterschied zur iterativen Entwicklung ist, dass bei der inkrementellen Entwicklung jeder Teil perfektioniert wird, bevor ein neuer Teil begonnen wird, während bei der iterativen Entwicklung das Gesamtprojekt entsteht und dieses als Ganzes nach und nach verbessert wird. (Cohn, o. D.)

1.1.2.3 Kombination der inkrementellen und iterativen Entwicklung

Beide Softwareentwicklungsmodelle in Kombination sind meist besser als eines dieser Verfahren allein anzuwenden. (Cohn, o. D.) Es gibt dabei mehrere Meilensteine. Bei jedem Meilenstein ist der insgesamt entstandene Code lauffähig und getestet und enthält mehr Funktionen als der Code im Meilenstein zuvor. Es ist hierbei auch noch möglich, dass Fehler frühzeitig erkannt und behoben werden. (Brichzin, 2015) In der Praxis hat sich die Kombination dieser Methoden als äußerst effizient erwiesen. Dabei wird das Projekt beginnend mit den nützlichsten Teilen schrittweise umgesetzt. (Schmidt & Schönwald, 2005)

1.1.3 Entwicklungsprozess bei Open-Source-Software

Ein Unterschied beim Entwickeln von Open-Source-Software im Vergleich zu proprietärer Software ist, dass die Open-Source-Software für eine breite Masse an Leuten öffentlich verfügbar ist. Dadurch unterscheidet sich auch die Art, wie der Code verfügbar gemacht und genutzt wird. Jedes Projekt hat seine eigenen Regeln. Es gibt zwei entscheidende Faktoren, die den Entwicklungsprozess leiten: Einerseits die Beitragenden zu einem Projekt und auf der anderen Seite die bisherige Entwicklungsgeschichte. Dabei haben Open-Source Projekte die gleichen Schwierigkeiten wie Projekte im Entwicklungsprozess von proprietärer Software. So muss entschieden werden, wie die Zusammenarbeit stattfinden soll, welche technischen Aspekte das Projekt beinhalten soll, wie dokumentiert wird, wie getestet und veröffentlicht wird, sowie die Entscheidung, wie der Code gewartet werden soll. (Mauerer & Jaeger, 2013)

Als beste Praktik hat sich dabei erwiesen, dass die Kommunikation in der Zusammenarbeit schriftlich über das Internet erfolgt. Dabei werden z. B. Chats oder Foren verwendet. Um die Anonymität der einzelnen Mitwirkenden zu verhindern, organisieren viele Projekte regelmäßige Treffen und halten z. B. Videokonferenzen ab. So können sich die Beitragenden kennenlernen und die Zusammenarbeit verbessern. Es gibt im Open-Source Entwicklungsprozess drei wichtige Rollen: Die Entwickler, die den Code schreiben, die Maintainer, die entscheiden, ob eine Codeänderung akzeptiert wird, sowie die Reviewer, die die Qualität des Codes prüfen. Wenn ein Entwickler eine Änderung am Code vornimmt, ist es wichtig, dass er den Anderen genau zu verstehen gibt, wofür diese Veränderung gut ist. Oft ist bei Open-Source Projekten auch der Programmierstil, also ob z. B. die Klammern mit Leerzeichen oder in einer neuen Zeile gesetzt werden, fest vorgegeben, damit die Übersicht über den Quellcode erhalten bleibt. Im Gegensatz zu proprietären Softwareprojekten findet man im Open-Source Entwicklungsprozess nicht unzählige Dokumente, welche die zu erledigende Arbeit von Beginn an genauestens dokumentieren. Pläne und Entscheidungen, die gemacht werden müssen, werden über die schriftliche Kommunikation abgearbeitet. Beim Veröffentlichen des Codes gibt es mehrere Varianten: Man kann den Code z. B. für alle verfügbar machen, wenn bestimmte neue Funktionen verfügbar sind, oder man veröffentlicht diesen immer zu bestimmten Veröffentlichungsterminen. Die Testphase hingegen unterscheidet sich nicht so stark von proprietären Softwareprojekten. Auch in Open-Source Projekten wird meistens auf Unit-Tests (siehe 1.2.2) zurückgegriffen. (Mauerer & Jaeger, 2013)

1.2 Rolle automatisierter Tests im Softwareentwicklungsprozess

Durch Automatisierung der Tests ist es möglich, den Aufwand für das Testen zu reduzieren und gleichzeitig Kosten einzusparen. Auch wenn man den Betreuungsaufwand dieser Tests nicht

unterschätzen darf, sind diese in den meisten Fällen dennoch sinnvoll. Wenn man die Automatisierung richtig einsetzt, kann die Effizienz des Softwareentwicklungsprozesses deutlich gesteigert werden. (Ebanhesaten & Stenhorst, 2011) So können sich wiederholende Tests schnell ausgeführt, die Testressourcen effektiver genutzt, menschliche Fehler reduziert, die Testabdeckung erhöht, sowie eine zeitliche Flexibilität bei der Ausführung geschaffen werden. Jedoch gibt es auch einige Nachteile. Da es mehrerer Schritte bedarf, um Tests zu automatisieren, entstehen hohe Initialkosten sowie ein enormer zeitlicher Aufwand. Des Weiteren benötigt man Spezialisten, um die Tests zu schreiben, es steigt die technische Komplexität und zudem lassen sich nicht alle Tests automatisieren. Daher ist es nötig, dass vorher genau überlegt wird, ob die Testautomatisierung für ein bestimmtes Projekt überhaupt sinnvoll ist. (Semenova, o. D.)

1.2.1 Entwicklertests im Vergleich zu externer Qualitätsanalyse

Während Entwicklertests vom Entwickler während oder sogar vor der Programmierung der einzelnen Teile selbst geschrieben und anschließend durchgeführt werden, wird bei der externen Qualitätsanalyse das Programm als Ganzes auf seine vorher festgelegten Spezifikationen durch unabhängige Tester getestet. (Panitz, 2010)

Wenn der Entwickler selbst sein eigenes Programm testet, ist das Testobjekt bekannt. Es ist also keine Zeit zum Einarbeiten nötig. (Jürjens, 2012) In der Regel werden Fehler beim Entwicklertest sehr früh gefunden und es ist dadurch wesentlich kostengünstiger diese Fehler zu beheben. Es ist zudem auch wesentlich leichter die Ursache des Fehlers im zugehörigen Quellcode zu finden. Es gibt sogar Tests, die nur vom Entwickler selbst durchgeführt werden können. Auch spart es Zeit, wenn der Entwickler selbst testet und sich nicht noch ein unabhängiges Test-Team darum kümmern muss. (Verifysoft, 2012) Jedoch hat es den Nachteil, dass der Entwickler seine eigenen Fehler evtl. gar nicht erkennt, da er davon ausgeht, dass es richtig ist und es sich nicht um einen Fehler handelt. Zudem sind beim Testen andere Kenntnisse gefordert als beim Entwickeln. Ein unabhängiger Tester ist im Normalfall auf das Testen spezialisiert und kann so in der Regel bessere Tests schreiben. (Jürjens, 2012) Ein Problem ist auch, dass die Entwickler häufig unter Zeitdruck leiden. Wenn sie Zeit einsparen müssen, geschieht dies häufig zu Lasten des zu testenden Objekts und die Entwickler testen zu wenig. (Hinrichs, 2011)

Durch das externe Testen können also Fehler gefunden werden, an die der Entwickler gar nicht gedacht hatte. Zudem bringt ein unabhängiges Test-Team das entsprechende Wissen über Tests mit, welches man als Tester haben sollte. Je mehr Erfahrung der Tester hat, umso einfacher kann er spezialisierte Werkzeuge anwenden. Jedoch muss sich der unabhängige Tester zuerst das Wissen über das zu testende Objekt aneignen, was zusätzliche Zeit kostet. (Jürjens, 2012)

Letzten Endes lässt sich sagen, dass beide Varianten sowohl ihre Vor- als auch ihre Nachteile haben. In der Praxis ist es daher meistens sinnvoll, sowohl Entwicklertests als auch Tests einer externen Qualitätsanalyse durchzuführen. (Verifysoft, 2012) Nur so kann man möglichst viele Fehler finden.

1.2.2 Teststufen

In einem Softwareprojekt werden typischerweise die folgenden Arten von Tests vorgenommen: Zuerst werden Komponententests durchgeführt, danach Integrationstests, anschließend Systemtests, woraufhin Systemintegrationstests folgen. Zum Schluss kommt der Abnahmetest. (Friske, 2013)

Während des Komponententests werden die kleinsten testbaren Teile einer Software getestet: die einzelnen Module. In der Regel testet dabei der Entwickler selbst. Die einzelnen Module werden von den anderen Modulen getrennt überprüft. Da dieser Test in einer sehr frühen Phase stattfindet, sind Korrekturen in der Regel günstiger als in den anderen Stufen. Es ist zudem viel einfacher die Ursache für einen Fehler zu finden, da nur eine kleine Einheit getestet wird. Für

das Testen können auch Test-Rahmenwerke zum Einsatz kommen. (Verifysoft, 2017) Außerdem bilden die Komponententests, welche auch Unit-Tests genannt werden, die wichtigste Grundlage der sogenannten Testpyramide, die aus den einzelnen Testphasen besteht. Unit-Tests sind nämlich schnell ausführbar und haben keinen großen Wartungsaufwand. Es ist sinnvoll, so viele Komponententests wie möglich zu schreiben und einzubauen. Die Anzahl dieser Tests sollte zudem weitaus größer sein als die der Tests aus späteren Stufen, welche meist komplexer sind und eine längere Ausführungszeit in Anspruch nehmen. (Müller, 2015) Ein weiterer Vorteil besteht darin, dass man mit Unit-Tests sofort den gerade erstellten Code-Teil überprüfen kann. Dadurch können auch Code-Stücke getestet werden, die noch nicht fertig sind. (Heimlich, 2018)

Beim anschließenden Integrationstest wird das Zusammenwirken der einzelnen Komponenten geprüft. Dabei wird z. B. die Kommunikation voneinander abhängiger Komponenten getestet. (Elberzhager & Naab, 2015) Es sollen dabei möglichst früh Fehler entdeckt werden, die beim Zusammenspiel der einzelnen Teile entstehen. Zudem wird getestet, ob die einzelnen Module des Systems wie geplant funktionieren. (SIMPLYTEST, o. D.)

Durch den Systemtest wird das gesamte System daraufhin überprüft, ob die zuvor gestellten Anforderungen korrekt umgesetzt wurden. (Friske & Schlingloff, 2005) Dabei findet in der Regel kein Zugriff auf den Quellcode statt. Zudem sind diese Tests nicht von der inneren Struktur des gesamten Systems abhängig. (Aubert, 2019)

Bei Durchführung des Systemintegrationstests wird getestet, ob das Zusammenwirken mehrerer einzelner Systeme als ein Ganzes funktioniert. Es muss dabei die Kommunikation untereinander, sowie die Zusammenarbeit getestet werden. Meist sind die einzelnen Systeme voneinander unabhängig entwickelt worden. (Ekssir-Monfared, 2010)

Zum Schluss findet der Abnahmetest statt. Dabei prüft der Auftraggeber, ob alle vertraglich festgehaltenen Anforderungen auch tatsächlich umgesetzt wurden. Zudem kontrolliert dieser, ob die vorher festgehaltenen Abnahmekriterien erfüllt sind. Der letzte Test findet dabei in der Systemumgebung des Auftraggebers statt. (Franz, 2007)

1.3 Das Test-Rahmenwerk JUnit

Zunächst ist es wichtig zu verstehen, was Test-Rahmenwerke, auch bekannt als Test-Frameworks, sind: Im Entwicklungsprozess kann es passieren, dass Tests nach Austausch einer Komponente oder nach Ergänzung der Programmfunktionalität wiederholt werden müssen, um festzustellen, ob die Veränderung den restlichen Programmiercode beeinflusst hat. Da es viel Zeit kostet, die gleichen Tests immer wieder durchzuführen, kann es sinnvoll sein, die Tests zu automatisieren. Durch den geringeren Zeitaufwand ist es so möglich Kosten zu sparen. Und hier kommen Test-Rahmenwerke ins Spiel: Durch diese können Tests automatisiert durchgeführt und ausgewertet werden. (Kalenborn, Will, Thimm, Raab & Fregin, 2006)

JUnit, eines dieser Test-Rahmenwerke, wurde von Kent Beck und Erich Gamma für die Programmiersprache Java entwickelt. (Ullenboom, 2010) Das Konzept von JUnit basiert auf dem für SmallTalk entwickelten SUnit. (Jan-Dirk, 2020) JUnit ermöglicht dem Nutzer unter anderem Unit-Tests (Komponententests) durchzuführen. Dabei wird die Korrektheit von Klassen oder sogar von ganzen Komponenten geprüft. Dies geschieht durch das Aufrufen einzelner Methoden des zu testenden Objekts und anschließendem Kontrollieren des Rückgabewertes, wobei begutachtet wird, ob die Methode sich wie erwartet verhält. (Kalenborn, Will, Thimm, Raab & Fregin, 2006) Mittlerweile ist JUnit als Test-Rahmenwerke nicht mehr wegzudenken und sogar in vielen IDEs standardmäßig vorhanden. (Ullenboom, 2010) Es dient zudem als Vorlage für viele weitere Frameworks, die Komponententests in anderen Programmiersprachen wie etwa C++ oder PHP ermöglichen. (Jan-Dirk, 2020) Doch nicht nur Komponententests können mit

JUnit ausgeführt werden: Es eignet sich für Tests auf den verschiedensten Ebenen. So können beispielsweise auch Integrationstest mittels JUnit umgesetzt werden. (Knapp & Yilmaz, 2016)

Im Jahr 2015 wurde begonnen an der neuen Version JUnit 5 zu arbeiten. Dabei wurde eine komplett neue Basis für den Code geschaffen. Mit JUnit 5 ist es erstmals möglich, dass Testfälle erst dynamisch zur Laufzeit erzeugt werden, die Parameter vorher also nicht genau definiert werden. Die Struktur war bis JUnit 4 monolithisch aus nur einer JAR-Datei, die den Projekten angefügt wurde, aufgebaut. Es bestand also alles aus einer einzelnen Datei. Die Architektur von JUnit 5 soll es möglich machen, dass Entwickler Tests schreiben können und die IDEs oder Build-Werkzeuge diese ausführen. Sie teilt daher beides in ein Test-Tool und in eine Plattform auf. Des Weiteren gibt es einen Adapter, der es ermöglicht, JUnit 4 mittels JUnit 5 auszuführen. (Voß, 2017) Ein weiterer Vorteil von JUnit 5 ist es, dass erstmals Methoden ein sogenannter Displayname gegeben werden kann. Dieser wird dann statt dem eigentlichen Methodennamen in der Ausgabe angezeigt und kann die Übersichtlichkeit deutlich verbessern. Auch kann man einfacher Tags und Annotationen erstellen. Ebenso ist es jetzt möglich, mit einer besonderen Annotation Unterklassen zu erstellen. Zudem können erstmals Lambda-Funktionen verwendet werden. (Merdes, 2018)

Diese Lambda-Funktionen waren eine der Hauptideen hinter der neuen Version von JUnit. So lief das Projekt unter dem Titel „JUnit Lambda“. Man wollte zudem das Ausführen von Tests vom Erstellen des Tests trennen. Der Name der neuen JUnit Engine lautet „JUnit Jupiter“. (Merdes, 2018) Es sollte erstmals möglich gemacht werden, dass sich andere Testrahmenwerke in die Engine einklinken können. (Redaktion JAXenter, 2017)

Beim Erstellen einer Test-Engine gibt es vier wichtige Schritte: Zuerst muss die Compile-Dependencies hinzugefügt werden, als nächstes ein Interface der Test-Engine umgesetzt werden, danach muss die Engine registriert und anschließend die Tests in der IDE gestartet werden. (JAX TV, 2017)

Ein Test mit JUnit 5 und somit auch der Jupiter-Engine kann beispielsweise so aussehen:

```
@Test
void testeAtomicIntegerAddAndGetMethode(){
     AtomicInteger atomicInteger = new AtomicInteger(5);
     int value = atomicInteger.addAndGet(10);
     assertEquals(value, 15);
}
```

Hierbei wird in einem Test ein `AtomicInteger` mit dem Wert `5` erzeugt und anschließend mit `10` addiert. Zum Schluss wird überprüft, ob das Ergebnis der Addition mit `15` übereinstimmt. Es handelt sich hier also um einen Testfall mit vorher festgelegten Werten.

2 Property-Based Testing

Im vorherigen Kapitel wurde das beispielbasierte Testen gezeigt. Allgemein lässt sich das Testen in drei Schritte unterteilen: In das Erstellen gültiger Eingabeparameter, Ausführen des Quellcodes mit diesen Parametern und Erfassen des Ausgabeparameters, sowie in den Vergleich, ob das Paar aus Eingabeparametern und dem Rückgabeparameter mit dem erwarteten Verhalten des Programms übereinstimmt. (Papadakis & Sagonas, 2011) Wendet man dies jedoch auf Beispieltests an, entstehen einige Nachteile. (siehe 2.1) Diese werden mittels Property-Based Testing versucht zu beheben. (siehe 2.2)

2.1 Einschränkung beispielbasierter Tests

Beim beispielbasierten Testen kommen einige Nachteile auf. So entstehen oft Redundanzen bei den Testfällen, da es mehrere Tests gibt, die sich nur in den Parametern unterscheiden, die sie der zu testenden Methode übergeben und von ihr zurückbekommen. Zudem kann es passieren, dass die Entwickler vergessen bestimmte Testfälle, insbesondere Grenzfälle, zu testen. Doch gerade diese Fälle sorgen in den Programmen oft für Fehler. (Macke, 2018) Des Weiteren wird für die Überprüfung im Normalfall kein Code verwendet. So wird das Ergebnis, mit dem der Rückgabewert verglichen wird, im Kopf des Programmierers erzeugt und ist somit anfälliger für Fehler. (Bush, 2018) Dies kann wertvolle Ressourcen wie etwa Zeit beanspruchen, da der Fehler dann erst im zu überprüfenden Code statt im Test gesucht wird. Ein weiteres Problem ist auch, dass beispielbasierte Tests eine allgemeine Aussage treffen wollen, dies aber in der Regel nicht können, da sie nur ein konkretes Szenario verwenden, man von diesem aber nicht immer auf eine generell gültige Aussage schließen kann. (Maciver, o. D.)

Das folgende Beispielszenario eines beispielbasierten Tests soll dieses Problem verdeutlichen. Dafür wurde eine Klasse `MitarbeiterListe` erstellt, die bis zu drei Mitarbeiter einer Firma verwaltet:

```
public class MitarbeiterListe {
     protected List<String> liste;
     public MitarbeiterListe(){ liste = new ArrayList<>(); }
     public boolean neuerMitarbeiter(String name){
          if(anzahlMitarbeiter() == 3){ return false; }
          return liste.add(name);
     }
     public void neueMitarbeiter(List<String> mListe){
          for(String name : mListe){ neuerMitarbeiter(name); }
     }
     public String mitarbeiter(int index){ return liste.get(index); }
     public int anzahlMitarbeiter(){ return liste.size(); }
     public boolean istMitarbeiter(String name){ return liste.contains(name); }
     public Stream<String> stream(){ return liste.stream(); }
```

```
    @Override
    public boolean equals(Object objekt){
        if(this == objekt){ return true; }
        if(objekt == null || getClass() != objekt.getClass()){ return false; }
        MitarbeiterListe mitarbeiter = (MitarbeiterListe) objekt;
        return Objects.equals(liste, mitarbeiter.liste);
    }
    @Override
    public int hashCode(){ return Objects.hash(liste);  }
}
```

Für diese Klasse wird nun ein geeigneter beispielbasierter Test geschrieben, der überprüfen soll, ob auch bis zu drei Mitarbeiter hinzugefügt werden können:

```
@Test
void testeMitarbeiterListeMitAlexKimJulian(){
    MitarbeiterListe mitarbeiter = new MitarbeiterListe();
    mitarbeiter.neuerMitarbeiter("Alex");
    mitarbeiter.neuerMitarbeiter("Kim");
    mitarbeiter.neuerMitarbeiter("Julian");
    assertTrue(mitarbeiter.istMitarbeiter("Alex"));
    assertTrue(mitarbeiter.istMitarbeiter("Kim"));
    assertTrue(mitarbeiter.istMitarbeiter("Julian"));
}
```

In diesem für beispielbasierte Tests klassischen Beispiel wird eine Liste erzeugt, die Mitarbeiter verwalten soll. Dieser Liste werden drei Namen der Mitarbeiter hinzugefügt und anschließend wird überprüft, ob die drei Mitarbeiter auch wirklich erfolgreich in die Liste eingefügt wurden. (Maciver, o. D.)

Doch gibt es dabei ein Problem: Anhand dieses Tests kann die ursprünglich zu überprüfende Aussage (bis zu drei Mitarbeiter können der Liste hinzugefügt werden) weder bestätigt noch widerlegt werden. Es kann lediglich gesagt werden, dass dieser Test besteht und der Liste die Namen Alex, Kim und Julian in genau dieser Reihenfolge erfolgreich angefügt werden können. Eine allgemeingültige Aussage kann also nicht getroffen werden. Auch ist unklar, was passiert, wenn man die Namen selbst oder ihre Reihenfolge ändert. Würde der Test hingegen fehlschlagen, könnte die Aussage, dass drei Mitarbeiter hinzugefügt werden können, widerlegt werden. Allerdings schlägt er nicht fehlt. (Maciver, o. D.)

Und genau das ist das Problem bei beispielbasierten Tests: Es wurde versucht eine allgemeingültige Aussage zu beweisen, dabei wurde lediglich das Szenario „Alex, Kim und Julian können in genau dieser Reihenfolge der Liste hinzugefügt werden“ bewiesen. (Maciver, o. D.) Um jedoch die generelle Aussage zu belegen, kommt das Property-Based Testing ins Spiel, das in den nachfolgenden Kapiteln erläutert werden soll.

2.2 Einführung in Property-Based Testing

Eines der wohl wichtigsten Merkmale von Property-Based Testing (PBT) ist, dass die Software nicht anhand von mehreren Beispielen getestet wird, sondern anhand von Properties (deutsch: Eigenschaften), die sie besitzt. Für das Testen wird also eine gewisse Anzahl an Properties definiert, die das Programm besitzen soll. Wichtig ist dabei, dass jede Property ein Test ist, jedoch

nicht jeder Test auch automatisch eine Property. Properties werden nachfolgend auch als Test bezeichnet. Die Ursprünge dieses Verfahrens liegen beim PBT-Tool QuickCheck für Haskell, jedoch gibt es mittlerweile viele weitere solche Tools für die verschiedensten Programmiersprachen. (Hughes, 2020) Im Gegensatz zu beispielbasierten Tests werden konkrete Szenarien verallgemeinert dargestellt. Es wird sich auf die wesentlichen Aspekte konzentriert. Tests sind in der Folge meist sauberer geschrieben, spezifizieren die Software genauer und decken Fehler auf, die anhand weniger Beispiele unentdeckt geblieben wären. (Maciver, o. D.)

Beim PBT muss der Tester also die generische Struktur der Eingabeparameter, sowie eine gewisse Anzahl an Properties, die für jede gültige Eingabe gelten sollen, angeben. Ein PBT-Werkzeug, welches diese Informationen erhält, erzeugt automatisch zufällige Eingabeparameter, übergibt sie dem Programm und überprüft dessen Ausführung, indem es verifiziert, ob das Ergebnis der Ausführung auch den Erwartungen (Invarianten) der Property entspricht. Sollte dies nicht der Fall sein, schlägt der Testfall fehl.

Eine Property ist dabei vereinfacht gesagt wie folgt definiert: Für alle Elemente, für die ggfs. eine bestimmte Vorbedingung erfüllt ist, muss eine bestimmte Eigenschaft wahr sein. (Dubien, 2018) Ein einfaches Beispiel wäre z. B. das Folgende: Für alle Zahlen a und b, bei denen a ungerade und b gerade ist, ist das Ergebnis der Addition ungerade.

Um den Fehler dann möglichst genau zu orten, kommt das sogenannte „Shrinking“ zum Einsatz. Dabei wird der Eingabeparameter, der den Fehler verursacht, möglichst weit vereinfacht, um einen möglichst kleinen Testfall zu erstellen. (Papadakis & Sagonas, 2011) Dieses Verfahren wird in Kapitel 2.6 genauer erläutert.

Ein Problem das PBT mit sich bringt ist, dass Entwickler anfangs oft Schwierigkeiten haben entsprechende Properties für die Tests zu finden. Auch tun sie sich zu Beginn schwer, die in den Anleitungen aufgeführten Tests zu generalisieren und auf andere Probleme anzuwenden. (Hughes, 2020) Dagegen hilft vor allem üben. Wie man außerdem geeignete Properties finden kann, wird in Kapitel 2.4 näher erläutert.

Der Unterschied zwischen PBT und beispielbasiertem Testen ist also, dass beim PBT der Entwickler die Testfälle nicht selbst festlegen muss, sondern diese automatisch generiert werden. Dies hilft auch dabei Fehler zu finden, an die der Programmierer unter Umständen gar nicht gedacht hätte. (Günther, o. D.) Um die Vorteile beider Varianten zu vereinen, kann es sinnvoll sein, beide Techniken zusammen einzusetzen. (Baganz, 2017)

2.3 Einführung in jqwik

Um Property-Based Testing für Java zu ermöglichen, wurde die auf JUnit 5 basierende Test-Engine jqwik entwickelt. Ein sehr erwähnenswerter Aspekt ist, dass jqwik kein eigenständiges Test-Rahmenwerk ist, sondern sich in das Framework JUnit 5 einklinkt. (Link, 2018, 29. März) Mittels jqwik können so automatisiert Testfälle erzeugt werden.

Das grundlegende Konzept dabei ist es, dass man bestimmte Properties anlegt, die erfüllt werden sollen. Dabei kennzeichnet man eine Testmethode mit der Annotation `@Property`. Die Parameter, die zufällig erzeugt werden sollen, kennzeichnet man mit `@ForAll`. Es ist wichtig, dass eine Methode entweder `true` im Erfolgsfall oder ansonsten `false` zurückgibt oder dass die Methode nichts zurückgibt (`void`) und stattdessen `Assertions` verwendet. Standardmäßig führt jqwik pro Methode 1.000 Versuche durch. Sollten weniger Werte möglich sein, wird die Anzahl der Versuche entsprechend reduziert. Zusätzlich kann man mittels `@Provide` Methoden bereitstellen, die Generatoren (siehe 2.5) zurückgeben, welche zufällige Werte generieren (jqwik, o. D. a)

Um den Nutzern zu zeigen in welcher Version und in welchem Status sich die einzelnen Teile von jqwik befinden, verwendet jqwik sogenannte API-Annotationen. Dabei gibt es die folgenden Status: Stable (Funktionen werden für Hauptversion nicht rückwärtskompatibel geändert), Maintained (Funktionen werden für Unterversion nicht rückwärtskompatibel geändert), Experimental (neue, experimentelle Funktionen), Deprecated (wird evtl. bald entfernt) und Internal (darf nur von jqwik selbst verwendet werden, nicht vom Anwender). (jqwik, o. D. a)

Es wird nun die Problematik aus Kapitel 2.1 erneut aufgegriffen und die dort getroffene Aussage mittels PBT in jqwik geprüft. Dazu wurde folgender Test geschrieben:

```
@Property
void testeMitarbeiterListeMitBisZuDreiNamen(@ForAll List<String> listeMitarbeiter){
     MitarbeiterListe mitarbeiter = new MitarbeiterListe();
     for(String name : listeMitarbeiter){
          mitarbeiter.neuerMitarbeiter(name);
     }
     assertThat(mitarbeiter.anzahlMitarbeiter()).isLessThanOrEqualTo(3);
}
```

Bei diesem Test, der 1.000-mal ausgeführt wird, wird als Eingabeparameter jeweils eine Liste übergeben, die zufällige `Strings` unterschiedlichster Längen enthalten, welche die Namen der Mitarbeiter repräsentieren. Darunter befinden sich auch leere `Strings`. Zusätzlich sind die übergebenen Listen auch verschieden lang. Sie können auch leer sein. Es werden jedoch auch Listen erzeugt, die mehr als drei Einträge haben. In der Methode selbst wird als Erstes eine Liste angelegt, die die Mitarbeiter verwalten soll. Dieser Mitarbeiterliste werden nach und nach alle Mitarbeiter aus der Eingabeliste hinzugefügt. Zum Schluss wird die Eigenschaft geprüft, ob die Mitarbeiterliste maximal drei Mitarbeiter enthält. Dies könnte man alternativ auch als booleschen Ausdruck formulieren.

Mithilfe dieses Tests ist man der Aussage „bis zu drei Mitarbeiter können hinzugefügt werden“ nun schon näher gekommen. Es ist nun naheliegend, dass die Liste maximal drei Namen aufnimmt. Jedoch ist noch unklar, ob überhaupt Namen aufgenommen werden. Dafür wurde der folgende Test erstellt:

```
@Property
void testeMitarbeiterListeNamenWerdenHinzugefuegt(
     @ForAll @Size(max = 3) List<String> listeMitarbeiter
){
     MitarbeiterListe mitarbeiter = new MitarbeiterListe();
     for(String name : listeMitarbeiter){
          mitarbeiter.neuerMitarbeiter(name);
     }
     for(String name : listeMitarbeiter){
          assertThat(mitarbeiter.istMitarbeiter(name)).isTrue();
     }
}
```

Auch bei diesem Test wird eine Liste einer zufälligen Länge mit zufälligen `Strings` als Eingabeparameter übergeben. Die `@Size` Annotation stellt sicher, dass die Liste diesmal maximal drei Elemente beinhaltet. Wieder wird zu Beginn eine Liste erstellt, die die Mitarbeiter verwalten

soll. Auch dieser Mitarbeiterliste werden alle `Strings` aus der Eingabeliste hinzugefügt. Anschließend wird für jedes Element aus der Eingabeliste überprüft, ob es auch in der Mitarbeiterliste vorhanden ist.

Mittels des zweiten Tests kann man nun die Aussage treffen, dass die Mitarbeiterliste wahrscheinlich die Eigenschaft hat, dass sie zwischen null und drei Mitarbeiter erfolgreich aufnehmen kann. Zusammen mit der Aussage aus dem ersten Test „die Liste nimmt maximal drei Namen auf“ ist man nun der ursprünglich zu überprüfenden Aussage „bis zu drei Mitarbeiter können hinzugefügt werden“ erfolgreich näher gekommen. Diese Eigenschaft gilt wahrscheinlich.

Statt einer festen Menge an Mitarbeiternamen wurden nun also zufällig generierte Namen verwendet. Auch die Anzahl der Mitarbeiter variierte. Es wurde gezeigt, dass die Mitarbeiterliste für verschiedene Namen wahrscheinlich funktioniert. Dies zu belegen ist die ursprüngliche Absicht, die hinter den Tests steckt. (Maciver, o. D.) Der Grund, wieso man diese Aussage jedoch nicht endgültig belegen kann, ist, dass die Testdaten zufällig erzeugt werden. (jqwik, o. D. a) Dadurch kann es passieren, dass Werte, für die die Tests fehlschlagen, erst irgendwann später erzeugt werden. Sollte ein Testfall aber fehlschlagen, so kann man die Aussage definitiv widerlegen.

Unklar ist derzeit was passiert, wenn ein Name doppelt übergeben wird. Auch weiß man nicht wie sich die Liste verhält, wenn mehr als drei Namen übergeben werden und ob sie dann z. B. ältere Namen löscht, anstatt den neuen Namen zu verwerfen. Hierfür müsste man weitere Properties schreiben.

2.4 Das Finden passender Properties

Wie eigenschaftsbasiertes Testen funktioniert, wurde in Kapitel 2.3 gezeigt. Doch wie findet man geeignete Tests? Dazu wird die Klasse `MitarbeiterListe` um eine `sortiert` Methode erweitert (Hughes, 2020):

```
public MitarbeiterListe sortiert(){
     Collections.sort(liste);
     return this;
}
```

Diese Methode sortiert die Mitarbeiter in der Mitarbeiterliste alphabetisch nach ihrem Namen. Es ist nun leicht, einen beispielbasierten Test zu finden (Hughes, 2020):

```
@Test
void testeSortiereMethodeKimAlexJulian(){
     MitarbeiterListe mitarbeiter = new MitarbeiterListe();
     mitarbeiter.neuerMitarbeiter("Kim");
     mitarbeiter.neuerMitarbeiter("Alex");
     mitarbeiter.neuerMitarbeiter("Julian");
     MitarbeiterListe sortiert = new MitarbeiterListe();
     sortiert.neuerMitarbeiter("Alex");
     sortiert.neuerMitarbeiter("Julian");
     sortiert.neuerMitarbeiter("Kim");
     assertEquals(mitarbeiter.sortiert(), sortiert);
}
```

Bei diesem Test werden zwei Mitarbeiterlisten erstellt. Eine Liste enthält die Namen in beliebiger Reihenfolge, die andere in sortierter Reihenfolge. Zum Schluss wird die erste Mitarbeiterliste mittels der `sortiert` Funktion umsortiert und geprüft, ob beide Listen jetzt identisch sind.

Die Form dieses Tests entspricht der Form, die für Tests überall verwendet wird: Es wird eine zu testende Funktion auf bekannte Werte angewandt und anschließend mit dem zu erwartenden Ergebnis verglichen. (Hughes, 2020) Jedoch steht man auch hier vor dem gleichen Problem: Statt der Aussage „Die Methode zum Sortieren der Mitarbeiterliste funktioniert“ hat man lediglich bewiesen, dass diese Behauptung für eine Liste mit den Mitarbeitern „Kim“, „Alex“ und „Julian“ in genau dieser Reihenfolge gilt. Dies ist aber wieder keine allgemeingültige Aussage, sondern gilt abermals nur für ein konkretes Szenario.

Wenn man das Ganze jetzt generell gültig machen will, steht man vor folgendem Problem:

```
@Property
void testeSortiertMethode(
     @ForAll @Size(max = 3) List<String> listeMitarbeiter
){
     //Wandle Liste in Mitarbeiterliste um
     assertThat(mitarbeiter.sortiert()).isEqualTo(???);
}
```

Der Vorteil dieses Tests liegt darin, dass man beliebige Mitarbeiterlisten testen kann. Jedoch stellt sich die Frage, was das erwartete Ergebnis ist. Es ist nicht möglich, die sortierte Liste im Voraus zu erstellen, da diesmal die Mitarbeiter in der Liste unbekannt sind. Die erste Lösung, die einem in den Sinn kommt, ist die Folgende:

```
assertThat(mitarbeiter.sortiert()).isEqualTo(erwarteteListe(listeMitarbeiter));
```

Doch ist die Funktion `erwarteteListe`, die die unveränderte Ursprungsliste als Eingabe bekommt, nicht einfacher zu schreiben, als die Methode `sortiert` aus der Klasse `MitarbeiterListe` selbst – Es handelt sich dabei um die exakt gleiche Funktion. Auch wenn das der erste Ansatz ist, der einem in den Sinn kommt, ist er nicht zu empfehlen. Die Kosten steigen, doch der erwartete Mehrwert bleibt aus: Fehler, die im Originalcode gemacht wurden, werden wahrscheinlich auch im zu testenden Code wiederholt. (Hughes, 2020)

Nun stellt sich die Frage, wie man den Code dann testen kann. Ganz einfach: Statt eines zu erwartenden Ergebnisses sucht man Properties, die auf die zu testende Methode zutreffen müssen (Hughes, 2020): So darf sich die Anzahl der Mitarbeiter in der Liste nicht verändern. Auch müssen alle Mitarbeiter erhalten bleiben. Jeder Name ist zudem lexikalisch größer als der vorherige Name. Dies wird mit den folgenden Tests überprüft:

```
@Property
void mitarbeiterSortiertGenausoGrossWieMitarbeiter(
     @ForAll @Size(max = 3) List<String> listeMitarbeiter
){
     MitarbeiterListe mitarbeiter = new MitarbeiterListe();
     MitarbeiterListe vergleichsListe = new MitarbeiterListe();
     for(String name : listeMitarbeiter){
          mitarbeiter.neuerMitarbeiter(name);
          vergleichsListe.neuerMitarbeiter(name);
     }
     assertEquals(
          vergleichsListe.anzahlMitarbeiter(),
          mitarbeiter.sortiert().anzahlMitarbeiter()
     );
}
@Property
void mitarbeiterSortiertMitarbeiterBleibenErhalten(
     @ForAll @Size(max = 3) List<String> listeMitarbeiter
){
     MitarbeiterListe mitarbeiter = new MitarbeiterListe();
     MitarbeiterListe vergleichsListe = new MitarbeiterListe();
     for(String name : listeMitarbeiter){
          mitarbeiter.neuerMitarbeiter(name);
          vergleichsListe.neuerMitarbeiter(name);
     }
     mitarbeiter.sortiert();
     for(int i = 0; i < listeMitarbeiter.size(); i++){
          assertTrue(mitarbeiter.istMitarbeiter(vergleichsListe.mitarbeiter(i)));
     }
}
```

```
@Property
void mitarbeiterSortiertNamenAufsteigend(
     @ForAll @Size(max = 3) List<String> listeMitarbeiter
){
     MitarbeiterListe mitarbeiter = new MitarbeiterListe();
     for(String name : listeMitarbeiter){
          mitarbeiter.neuerMitarbeiter(name);
     }
     mitarbeiter.sortiert();
     for(int i = 1; i < listeMitarbeiter.size(); i++){
          assertThat(
               mitarbeiter.mitarbeiter(i).compareTo(mitarbeiter.mitarbeiter(i-1))
          ).isGreaterThanOrEqualTo(0);
     }
}
```

Bei allen drei Tests wird dabei eine Mitarbeiterliste angelegt, die die Mitarbeiter verwalten soll. Wenn nötig, wird eine zweite solche Liste erzeugt, um die sortierte Liste nachher mit einer Ursprungsliste vergleichen zu können. Diesen Listen werden anschließend die einzelnen Mitarbeiter hinzugefügt. Zum Schluss werden die oben genannten Aussagen auf ihre Gültigkeit geprüft, indem mit der ursprünglichen Liste verglichen wird.

Die Tests verlaufen erfolgreich. Doch gibt es noch ein weiteres Problem: Manchmal treffen Properties nämlich auch zu, wenn man eine Funktion fehlerhaft schreibt. Ursprünglich erstellte beispielbasierte Tests hingegen können den Fehler meist erkennen und schlagen dann fehl. (Link, o. D.) Und genau das ist der Grund, wieso man sich nicht nur auf Property-Based Testing verlassen sollte, sondern es am besten in Kombination mit beispielbasiertem Testen verwendet. (Baganz, 2017) Beide Verfahren in Kombination können die ursprüngliche Aussage dann bestätigen. Mittels der Annotation `@Example` kann man in jqwik beispielbasierte Tests einbauen. Dies funktioniert äquivalent zu `@Test` in JUnit. Der einzige Unterschied dabei: jqwik erwartet nicht nur `void` Methoden, sondern kann zusätzlich auch mit `boolean` Werten als Rückgabe umgehen. (Link, o. D.)

Hilfreich ist es auf der Suche nach Properties zu prüfen, ob die Properties aus der Spezifikation des Problems abgeleitet werden können. Man kann zudem prüfen, ob keine Exceptions auftreten, alle Rückgabewerte gültig sind, die Laufzeit akzeptabel ist und bei mehrmaligem Aufrufen der Funktion mit gleichem Eingabeparameter das gleiche Ergebnis auftritt. Hat die Methode eine inverse Funktion, so ist es möglich zu prüfen, ob bei Aufruf der Methode und anschließend der inversen Funktion der ursprüngliche Wert herauskommt. Ruft man eine idempotente Funktion mehrfach hintereinander auf, darf sich das Ergebnis nicht verändern. (z. B. mehrmaliger Aufruf von `sortiert`) Dies gilt auch, wenn man kommutative Methoden in unterschiedlicher Reihenfolge aufruft. (Link, 2018, 16. Juli) Diese und weitere Möglichkeiten, wie man geeignete Properties, findet werden in den Blogs von Link (2018, 16. Juli) und Blog.ssanj.net (2016) genauer erklärt.

2.5 Grundlegendes über Generatoren

In manchen Fällen sind die vorhandenen Generatoren für Werte nicht ausreichend. In diesem Fall muss man sich selbst einen geeigneten Generator bauen. (Hughes, 2020) Dies ist häufig der Fall, wenn es um Objekte aus der Anwendungsdomäne, wie z. B. selbst erstellte Klassen

geht. Diese sind in den verschiedenen PBT-Tools unterschiedlich vertreten. (Link, o. D.) Die folgenden Abschnitte konzentrieren sich daher ausschließlich auf Generatoren in jqwik.

2.5.1 Generatoren in jqwik

Zunächst ist zu klären, wie Generatoren in jqwik vertreten sind. Diese werden durch das Interface `Arbitrary<T>` repräsentiert. Das `T` steht dabei für den Typ, den das Objekt hat, für das ein Generator erstellt wird. Sobald die zufälligen Werte gebraucht werden, erzeugt die Klasse ein Objekt des Typs `RandomGenerator<T>`, der die Werte generiert. Gleichzeitig kann `Arbitrary<T>` auch verwendet werden, um Generatoren z. B. mittels einer `map` Funktion zu verändern. (jqwik, o. D. a)

Ein Generator für `Strings` implementiert beispielsweise das Interface `Arbitrary<String>`. Dieser Generator stellt dann zufällig erstellte `Strings` bereit. (jqwik, o. D. a)

Im Falle von Zeichenketten gibt es zusätzlich ein weiteres Interface: `StringArbitrary`. Dieses Interface erweitert `Arbitrary<String>` um weitere Methoden. (jqwik, o. D. a) So können z. B. mit den zusätzlichen Methoden `ofLength`, `ofMinLength` und `ofMaxLength` die Längen und mit den Methoden `all`, `alpha`, `ascii`, `numeric`, `whitespace`, `withCharRange` und `withChars` die erzeugten Zeichen der Zeichenketten eingeschränkt werden. (jqwik, o. D. b) Dies wäre ohne das erweiternde Interface `StringArbitrary` nicht so leicht möglich und es müssten z. B. entsprechende Filter selbst gebaut werden.

Des Weiteren ist es in jqwik möglich benutzerdefinierte Generatoren zu erstellen. So können vom Anwender Generatoren für ganze Typen neu erstellt, oder mittels zur Verfügung gestellter Methoden ein oder mehrere Generatoren zu einem neuen Generator verknüpft werden. (jqwik, o. D. a)

Ein einfaches Beispiel wie Generatoren selbst erstellt werden können stellt das Interface `StringArbitrary` dar. Dies ist ein eigenes Interface, erbt jedoch die grundsätzlichen Methoden des Interfaces `Arbitrary<String>` und erweitert diese. In einer dazugehörigen Klasse `DefaultStringArbitrary`, welche erstellt werden muss, um die Methoden des Interfaces dann zu implementieren, werden die funktionalen Eigenschaften für ein `StringArbitrary` umgesetzt. Es wird dazu eine Möglichkeit benötigt, sich ein Objekt der Klasse `DefaultStringArbitrary` als `StringArbitrary` zurückgeben zu lassen. (hier: `Arbitraries.strings()`). Mittels der `@Provide` Annotation können anschließend Objekte des Typs `StringArbitrary` zur Verfügung gestellt werden:

```
@Provide
StringArbitrary erstelleStringsMit10Zeichen(){
     StringArbitrary stringArbitrary = Arbitraries.strings();
     stringArbitrary = stringArbitrary.ofLength(10);
     return stringArbitrary;
}
```

Hier wird ein Objekt des Typs `StringArbitrary` erstellt. Im Gegensatz zu Objekten der Klasse `Arbitrary<String>` kann man zusätzlich die Länge der einzelnen Strings auf genau zehn Zeichen festlegen, was der Vorteil ist, wenn man eigene Interfaces wie etwa `StringArbitrary` mit entsprechenden Methoden erstellt. Um den bereitgestellten Generator zu verwenden, muss man in der dazugehörigen Property der `@ForAll` Annotation den Namen der Methode als Parameter übergeben:

```
@Property
void testeStrings(@ForAll("erstelleStringsMit10Zeichen") String s){
     //mache etwas
}
```

Die Methode `testeStrings` erhält nun `Strings` der Länge zehn, die vom `StringArbitrary`-Generator erzeugt werden.

2.5.2 Mapping

Mittels der Methode `map` können die erzeugten Werte aus einem bereits existierenden Generator verwendet werden, um einen neuen Generator mit anderen Objekten zu bauen. (jqwik, o. D. a)

```
Arbitrary<String> zahlInAnführungszeichen(){
     IntegerArbitrary integerArbitrary = Arbitraries.integers().between(1, 1000);
     return integerArbitrary.map(wert -> "\"" + wert + "\"");
}
```

In diesem Beispiel wird zuerst ein Generator für Ganzzahlen erzeugt. Seine generierten Werte werden anschließend in Anführungszeichen gegeben, er wird somit zu einem Generator des Typs `Arbitrary<String>` umgewandelt, der `Strings` erzeugt.

2.5.3 Flat-Mapping

In manchen Fällen ist die Methode `map` jedoch nicht ausreichend. So gibt es die Methode `flatMap`, die genutzt wird, wenn der Nutzer aus generierten Werten eines Generators einen neuen Generator erstellen will. Der Unterschied zwischen den beiden Methoden ist, dass bei `map` der Typ des Generators umgewandelt wird, während bei `flatMap` für jeden erzeugten Wert ein vollkommen neuer Generator kreiert wird. (jqwik, o. D. a)

```
Arbitrary<List<String>> zahlAlsStringLaengeFuerListe(){
     IntegerArbitrary integerArbitrary = Arbitraries.integers().between(1, 10);
     return integerArbitrary.flatMap(value -> {
          Arbitrary<String> stringArbitrary = Arbitraries.strings()
                                                  .ofLength(value);
          return stringArbitrary.list();
     });
}
```

In diesem Beispiel wird ein Generator für Ganzzahlen erstellt. Seine generierten Werte werden einem neu erzeugten Generator für `Strings` übergeben, der den Wert des ursprünglichen Generators als Länge der Zeichenketten nutzt. Die generierten Zeichenketten werden zusammen in Listen eingefügt. Zurückgegeben wird dann ein Generator für Listen. Die Länge der Zeichenketten unterscheidet sich in den verschiedenen erzeugten Listen, innerhalb einer Liste sind sie aber jeweils gleich groß.

2.5.4 Kombinieren mehrerer Generatoren

Eine weitere Möglichkeit Generatoren zu erstellen ist, mehrere Generatoren mittels der Methode `Combinators.combine` zu einem neuen Generator zu kombinieren. (jqwik, o. D. a)

```
Arbitrary<Integer> multipliziereZweiZahlen(){
     IntegerArbitrary integerArbitrary1 = Arbitraries.integers().between(0, 100);
     IntegerArbitrary integerArbitrary2 = Arbitraries.integers().between(10, 30);
     return Combinators.combine(integerArbitrary1, integerArbitrary2)
                         .as((int1, int2) -> int1 * int2);
}
```

Bei diesem Beispiel werden erst zwei Generatoren für jeweils eine ganze Zahl mit unterschiedlichem Wertebereich erzeugt, deren Werte anschließend zu einer Zahl (Multiplikation aus den beiden anderen Zahlen) kombiniert werden. Es wird ein neuer Generator des Typs `Arbitrary<Integer>` zurückgegeben.

2.5.5 Filtern unerwünschter Werte

Da Generatoren manchmal mehr Werte erzeugen als man möchte, gibt es zusätzlich die Möglichkeit, dass die erzeugten Werte mit der Methode `filter` gefiltert werden. (jqwik, o. D. a)

```
Arbitrary<Integer> filterNurGeradeZahlen(){
     Arbitrary<Integer> integerArbitrary = Arbitraries.integers();
     return integerArbitrary.filter(wert -> wert % 2 == 0);
}
```

In diesem Beispiel wird erst ein Generator für Ganzzahlen erzeugt. Anschließend wird ein neuer Generator erstellt und zurückgegeben, der die Werte des alten Generators gefiltert enthält, sodass der neue Generator nur noch gerade Zahlen erzeugen kann.

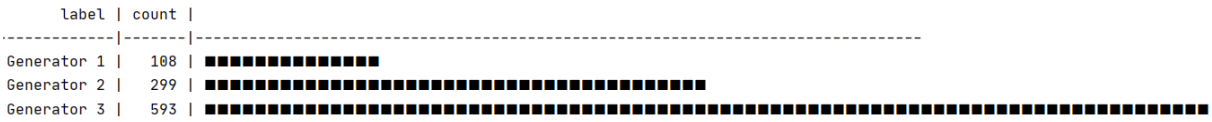
2.5.6 Zufällige Auswahl aus mehreren Generatoren

Eine weitere Möglichkeit ist, einen Generator zu erzeugen, der zufällig zwischen zwei oder mehr Generatoren desselben Typs entscheidet. Dafür wird für jeden generierten Wert einer der übergebenen Generatoren ausgewählt. (jqwik, o. D. a)

```
Arbitrary<Integer> zufaelligeWerte(){
     Arbitrary<Integer> generator1 = Arbitraries.integers().between(2000, 2500);
     Arbitrary<Integer> generator2 = Arbitraries.integers().between(3000, 3500);
     Arbitrary<Integer> generator3 = Arbitraries.integers().between(4000, 4500);
     int haeufigkeitGenerator1 = 1;
     int haeufigkeitGenerator2 = 3;
     int haeufigkeitGenerator3 = 6;
     return Arbitraries.frequencyOf(
          Tuple.of(haeufigkeitGenerator1, generator1),
          Tuple.of(haeufigkeitGenerator2, generator2),
          Tuple.of(haeufigkeitGenerator3, generator3)
     );
}
```

Diese Methode erzeugt drei Generatoren für Ganzzahlen sowie je einen Wert, der ihre Häufigkeit beschreibt. Anschließend wird ein Generator zurückgegeben, der in zehn Prozent der Fälle einen Wert von Generator eins, in 30 Prozent der Fälle einen Wert von Generator zwei und im Rest der Fälle einen Wert von Generator drei zurückgibt. Dies wird auch durch folgendes mittels

jqwik erstelltes Histogramm sichtbar, das allerdings eine leichte zufallsbedingte Abweichung zeigt:



2.5.7 Fazit der vorhandenen Möglichkeiten

Durch die dargestellten Varianten ist es dem Nutzer also leicht möglich selbst Generatoren zu erstellen oder bereits vorhandene Generatoren nach eigenem Wunsch zu verändern und daraus neue Generatoren zu kreieren.

Die hier genannten Möglichkeiten sind jedoch nur ein kleiner Teilausschnitt. In der Praxis gibt es noch viel mehr, was umgesetzt werden kann. Es ist z. B. auch möglich sich Listen, Arrays oder rekursive Generatoren zu bauen. (jqwik, o. D. a)

2.6 Shrinking

Ein Mechanismus, der Property-Based Testing auszeichnet, ist das sogenannte „Shrinking“ (deutsch: Schrumpfen). (Hughes, 2020) Wie dieses genau funktioniert, soll mit Hilfe des folgenden Beispiels erläutert werden: Es wurde eine Klasse `Prozentsatz` erstellt, welche Zahlen von `0` bis `100` als Prozentsatz verwaltet. Sie enthält eine Methode `wert`, die den verwalteten Prozentsatz zurückgibt. Diese Methode schlägt für Zahlen von `30` bis `70` fehl. Es handelt sich hierbei allerdings um einen Fehler und der Entwickler weiß nicht, dass die Methode fehlschlägt:

```
public class Prozentsatz {
	private int wert;
	public Prozentsatz(int wert){
		this.wert = Math.max(0, Math.min(100, wert));
	}
	public int wert(){
		if(wert >= 30 && wert <= 70){
			throw new RuntimeException("Irgendetwas ging schief");
		}
		return wert;
	}
}
```

Die Methode `wert` soll nun auf Korrektheit überprüft werden. Dazu wurde folgender Test erstellt, der ganze Zahlen von `0` bis `100` generiert:

```
@Property(tries = 10)
void testeProzentsatz(@ForAll @IntRange(min = 0, max = 100) int wert){
	Prozentsatz prozentsatz = new Prozentsatz(wert);
	assertThat(prozentsatz.wert()).isEqualTo(wert);
}
```

Dieser wird nun zehnmal mittels jqwik ausgeführt und liefert z. B. folgendes Ergebnis:

```
Original Sample: wert: 69

Shrunk Sample (3 steps): wert: 30

tries = 1
```

Daraus wird ersichtlich, dass jqwik die Methode `testeProzentsatz` beim ersten Versuch mit dem Eingabeparameter `wert = 69` aufgerufen hat. Der Test ist fehlgeschlagen. Jqwik hat diesen Parameter nun so weit vereinfacht, bis der kleinste fehlschlagende Wert, in diesem Fall `30`, erreicht wurde. Es wurden dabei drei Schritte benötigt. (jqwik, o. D. a)

Anhand dieses Beispiels lässt sich gut erkennen, was das Shrinking ist und wie sinnvoll es für das Testen ist. Man findet dadurch den kleinsten Wert, für den ein Programm fehlschlägt, und kann dadurch die Ursache des Fehlers besser erkennen. Jedoch gibt es auch Fälle, bei denen es keinen kleineren Wert gibt. In diesem Fall zeigt jqwik nur den fehlgeschlagenen Wert an. (jqwik, o. D. a)

2.7 Bedingungen beim Testen

Es kann nötig sein, einen Testfall an bestimmte Bedingungen zu knüpfen. Dies können z. B. Vor- oder Nachbedingungen sein, die vor der Ausführung der zu testenden Methode respektive danach gelten müssen. (Hughes, 2020)

2.7.1 Vorbedingungen

In vielen Fällen ist es notwendig, Tests mit geeigneten Vorbedingungen zu versehen. Das sind Bedingungen, die gültig sein müssen, damit ein Test durchgeführt werden kann. (Hughes, 2020) Mittels der `Assume.that` Funktion (Assumption) kann dabei festgelegt werden, unter welcher Bedingung der Test ausgeführt wird. (jqwik, o. D. a)

Das Ganze soll mit dem folgenden Beispiel verdeutlicht werden:

```
@Provide
Arbitrary<MitarbeiterListe> mitarbeiterListenBis2(){
     Arbitrary<String> strings = Arbitraries.strings().ofMinLength(1).alpha();
     Arbitrary<List<String>> listen = strings.list().ofMaxSize(2).uniqueElements();
     return listen.map(this::addAll);
}
MitarbeiterListe addAll(List<String> listeMitarbeiter){
     MitarbeiterListe mitarbeiter = new MitarbeiterListe();
     mitarbeiter.neueMitarbeiter(listeMitarbeiter);
     return mitarbeiter;
}
```

Es wird eine Funktion zur Verfügung gestellt, die zufällige Mitarbeiterlisten generiert. Dabei werden nur `Strings` erstellt, die mindestens ein Zeichen enthalten und aus Buchstaben bestehen. Die `Strings` repräsentieren die Namen der Mitarbeiter und werden anschließend in eine Liste mit der maximalen Länge zwei eingefügt. Anschließend werden die Listen mithilfe der Methode `addAll` in Listen des Typs `MitarbeiterListe` umgewandelt und zurückgegeben.

Nun soll überprüft werden, ob das Einfügen in die Mitarbeiterliste funktioniert:

```
@Property
void mitarbeiterListeNameHinzufuegen(
     @ForAll("mitarbeiterListenBis2") MitarbeiterListe mitarbeiter,
     @ForAll @AlphaChars @StringLength(min = 1) String name
){
     Assume.that(!mitarbeiter.istMitarbeiter(name));
     mitarbeiter.neuerMitarbeiter(name);
}
```

Hierbei werden Mitarbeiterlisten der maximalen Länge zwei sowie ein Mitarbeitername, der mindestens ein Zeichen enthält und aus Buchstaben besteht, generiert. Wenn der Name noch nicht in der Liste vorhanden ist, wird er hinzugefügt, ansonsten wird der Testfall nicht ausgeführt.

Dabei gibt es jedoch folgendes Problem: Noch wird nicht garantiert, dass die automatisch erzeugten Mitarbeiterlisten auch wirklich gültig sind. Die Vorbedingung, die in diesem Fall erfüllt sein muss, ist, dass gültige Mitarbeiterlisten der maximalen Länge zwei erzeugt werden, die jeden Namen, welcher mindestens ein Zeichen lang sein muss, nur einmal enthalten. Zudem dürfen die Namen nur aus Buchstaben bestehen. Dazu wird die folgende Funktion geschrieben:

```
boolean istGueltig(MitarbeiterListe mitarbeiter, int maxLaenge){
     return mitarbeiter != null
          && mitarbeiter.anzahlMitarbeiter() <= maxLaenge
          && mitarbeiter.stream().allMatch(name -> name != null && name.length()>0)
          && namenNurEinmal(mitarbeiter)
          && nurBuchstaben(mitarbeiter);
}
boolean namenNurEinmal(MitarbeiterListe mitarbeiter){
     List<String> namen = new LinkedList<>();
     AtomicBoolean nurEinmal = new AtomicBoolean(true);
     for(int i = 0; i < mitarbeiter.anzahlMitarbeiter(); i++){
          String name = mitarbeiter.mitarbeiter(i);
          if(namen.contains(name)){
               nurEinmal.set(false);
          }
          namen.add(name);
     }
     return nurEinmal.get();
}
```

```
boolean nurBuchstaben(MitarbeiterListe mitarbeiter){
     AtomicBoolean nurBuchstaben = new AtomicBoolean(true);
     for(int i = 0; i < mitarbeiter.anzahlMitarbeiter(); i++){
          String name = mitarbeiter.mitarbeiter(i);
          for(char buchstabe : name.toLowerCase().toCharArray()){
               if(buchstabe < 'a' || buchstabe > 'z'){
                    nurBuchstaben.set(false);
               }
          }
     }
     return nurBuchstaben.get();
}
```

Die Methode `istGueltig` gibt dabei zurück, ob die Vorbedingung erfüllt ist. Um die Methode auch für Listen mit mehr Namen verwenden zu können, gibt es den Parameter `maxLaenge`, der für diese Bedingung den Wert `2` übergeben bekommt. Die Hilfsmethoden `namenNurEinmal` und `nurBuchstaben` helfen, indem sie eine Mitarbeiterliste dahingehend prüfen, ob jeder Name nur einmal vorkommt bzw. alle Namen nur aus Buchstaben bestehen. Diese Vorbedingung wird nun in den zuvor geschriebenen Test eingebaut:

```
@Property
void mitarbeiterListeNameHinzufuegen(
     @ForAll("mitarbeiterListenBis2") MitarbeiterListe mitarbeiter,
     @ForAll @AlphaChars @StringLength(min = 1) String name
){
     Assume.that(istGueltig(mitarbeiter, 2));
     Assume.that(!mitarbeiter.istMitarbeiter(name));
     mitarbeiter.neuerMitarbeiter(name);
}
```

Die Tests werden nur dann ausgeführt, wenn die Vorbedingung erfüllt ist. Allerdings hat dies Nachteile: Wenn man viele verschiedene Tests durchführt, die die gleiche Vorbedingung haben, so muss immer wieder überprüft werden, ob diese auch erfüllt ist. Gerade wenn die Mitarbeiterliste auf z. B. 2.000 Namen erweitert wird, würde dies die Testzeit unnötig in die Länge ziehen. (Hughes, 2020) Auch wenn der Generator später verändert wird, könnten plötzlich alle Tests mit dieser Vorbedingung viele Ausführungen aufgrund der Assumption verwerfen und somit Laufzeit verschwenden und ein ungenaueres Ergebnis liefern. (Link, o. D.)

Deswegen ist es sinnvoll, die Vorbedingung in einen gesonderten Test auszulagern:

```
@Property
void generatorGueltig(
     @ForAll("mitarbeiterListenBis2") MitarbeiterListe mitarbeiter
){
     assertThat(istGueltig(mitarbeiter, 2)).isTrue();
}
```

Dieser Test prüft nun den Generator für die Mitarbeiterlisten darauf, ob die einzelnen Listen die Vorbedingung für die Tests erfüllen und ob er somit gültig ist. Dadurch kann man die Assumptions, die die Bedingung überprüfen, in den einzelnen Tests verwerfen und somit wertvolle Ausführungszeit sparen. (Hughes, 2020)

Anschließend ist es einfach eine Property zu definieren, um zu testen, ob die Liste nach dem Einfügen weiterhin gültig ist (Hughes, 2020):

```
@Property
void mitarbeiterListeNameHinzufuegen(
     @ForAll("mitarbeiterListenBis2") MitarbeiterListe mitarbeiter,
     @ForAll @AlphaChars @StringLength(min = 1) String name
){
     Assume.that(!mitarbeiter.istMitarbeiter(name));
     mitarbeiter.neuerMitarbeiter(name);
     assertThat(istGueltig(mitarbeiter, 3)).isTrue();
}
```

In diesem Fall muss man die erlaubte Größe der Liste als Eingabeparameter auf drei setzen, da – wenn alles funktioniert hat – ein neuer Eintrag hinzugekommen ist.

Man muss jedoch auch aufpassen, dass die Assumptions nicht zu stark sind:

```
@Property
void zweiGleicheZahlen(@ForAll int zahl1, @ForAll int zahl2){
     Assume.that(zahl1 == zahl2);
     assertThat(zahl1).isEqualTo(zahl2);
}
```

Folgender Test sieht richtig aus und funktioniert im Grunde genommen, dennoch schlägt er höchstwahrscheinlich mit der Meldung fehl, dass von 1.000 Versuchen z. B. nur neun vollständig ausgeführt wurden und somit 991-mal die Assumption fehlerhaft war. In der Standardeinstellung lässt jqwik Tests fehlschlagen, wenn das Verhältnis zwischen generierten und akzeptierten Parametern nicht größer als fünf ist. Zwar kann prinzipiell vom Nutzer ein anderes Verhältnis festgelegt werden, doch ist dies in der Regel nicht sinnvoll. Der Zweck, der nämlich dahintersteckt, ist, dass jqwik sicherstellen will, dass die Tests oft genug ausgeführt werden und somit aussagekräftig bleiben. Manchmal ist es möglich die Assumption mit etwas Kreativität zu umgehen. (jqwik, o. D. a) Im oben gezeigten Beispiel wäre das auf folgende Weise möglich:

```
@Property
void zweiGleicheZahlen(@ForAll int zahl1){
     int zahl2 = zahl1;
     assertThat(zahl1).isEqualTo(zahl2);
}
```

In diesem Beispiel entfällt die Assumption komplett, in anderen Fällen sind schwächere Assumptions nötig.

2.7.2 Nachbedingungen

In Kapitel 2.7.1 wurde ein Test erstellt, der einen Namen zu einer Mitarbeiterliste hinzufügt. Doch noch weiß man nicht, ob der Name wirklich erfolgreich hinzugefügt wurde. Dafür gibt es

die Nachbedingungen. Dazu zählen alle Bedingungen, die nach einem Aufruf der zu testenden Methode gültig sein müssen. (Hughes, 2020)

Für den im vorherigen Kapitel angelegten Testfall wäre eine Nachbedingung, dass der eingefügte Name in der Mitarbeiterliste auch wirklich vorhanden ist:

```
@Property
void mitarbeiterListeNameHinzufuegen(
     @ForAll("mitarbeiterListenBis2") MitarbeiterListe mitarbeiter,
     @ForAll @AlphaChars @StringLength(min = 1) String name
){
     Assume.that(!mitarbeiter.istMitarbeiter(name));
     mitarbeiter.neuerMitarbeiter(name);
     assertThat(istGueltig(mitarbeiter, 3)).isTrue();
     assertThat(mitarbeiter.istMitarbeiter(name)).isTrue();
}
```

Allerdings weiß man noch nicht, was die Methode `add` wirklich macht. Sie könnte z. B. alle anderen Namen aus der Liste entfernen oder die Namen verändern. Nachbedingungen sind also nicht leicht zu schreiben. Man müsste zusätzlich sicherstellen, dass die Methode die ursprüngliche Liste – bis auf das Einfügen des neuen Namens – unverändert lässt. Dies kann gerade bei langen Listen viel Laufzeit kosten. (Hughes, 2020)

2.8 Statistiken

Oft ist es nötig sicherzustellen, dass ein PBT-Tool die Werte erzeugt, die erwartet werden. Meistens will man dabei wissen, wie häufig ein bestimmter Wert tatsächlich generiert wird. (jqwik, o. D. a)

Jqwik bietet dafür die Möglichkeit Statistiken zu erstellen. Dies funktioniert wie folgt:

```
@Property
void testeMitarbeiterlistenGroesse(
     @ForAll("mitarbeiterListenBis2") MitarbeiterListe mitarbeiter
){
     Statistics.collect(mitarbeiter.anzahlMitarbeiter());
     Statistics.coverage(coverage -> {
          coverage.check(0).percentage(p -> p > 20);
          coverage.check(1).percentage(p -> p > 20);
          coverage.check(2).percentage(p -> p > 20);
     });
}
```

Hier wird zunächst für den in einem vorherigen Kapitel erstellten Generator eine Statistik angelegt, die die jeweilige Größe der Mitarbeiterliste sammelt. Wenn alle Daten gesammelt sind, wird im Beispiel für die Längen null, eins und zwei überprüft, ob sie jeweils in mehr als 20 Prozent der Fälle vorkommen. Da die Längen der Listen zufällig sind und somit Schwankungen auftreten können, ist es wichtig, dass man keine genaue Grenze testet und statt 33 Prozent etwas Spielraum nach unten gibt. Erfüllt der Test die Bedingung nicht, schlägt er fehl. (jqwik, o. D. a)

Lässt man den Test laufen, erhält man beispielsweise folgende Ausgabe:

```
(1000) statistics =
1 (389) : 39 %
0 (310) : 31 %
2 (301) : 30 %
```

Diese gibt an, dass von 1.000 gesammelten Längen in 39 Prozent der Fälle die Länge eins war, in 31 Prozent der Fälle war die Mitarbeiterliste leer und in 30 Prozent der Fälle enthielt die Liste zwei Namen. In den Klammern steht die jeweils genaue Anzahl wie oft welcher Wert aufgetreten ist.

Statt der `percentage` Funktion kann man auch die Methode `count` verwenden:

```
coverage.check(0).count(c -> c > 200);
```

Diese Zeile würde testen, ob der Wert `0` über 200-mal vorkommt und ist somit bei 1.000 übergebenen Listen identisch mit der Zeile, die die Prozentzahl `20` testet. (jqwik, o. D. a)

3 Erweiterung von jqwik um die Generierung von E-Mail-Adressen

In vielen Anwendungen müssen sich Nutzer mit ihrer E-Mail-Adresse registrieren. Jedes Programm, das E-Mail-Adressen annimmt, sollte daher auch mit E-Mail-Adressen in allen Varianten getestet werden. Etwas, dass auch die Berliner Online-Kfz-Zulassungsstelle auf bittere Art lernen musste: Es wurden dort fälschlicherweise E-Mail-Adressen der Form `mail@[domain]` durch eine Software abgelehnt, die Angriffe erkennen sollte. In einem FAQ-Eintrag wies man dann sogar darauf hin, dass nur E-Mail-Adressen von bekannten Anbietern verwendet werden können. (Böck, 2020) Ein Problem, das durch ordentliches Testen hätte vermieden werden können.

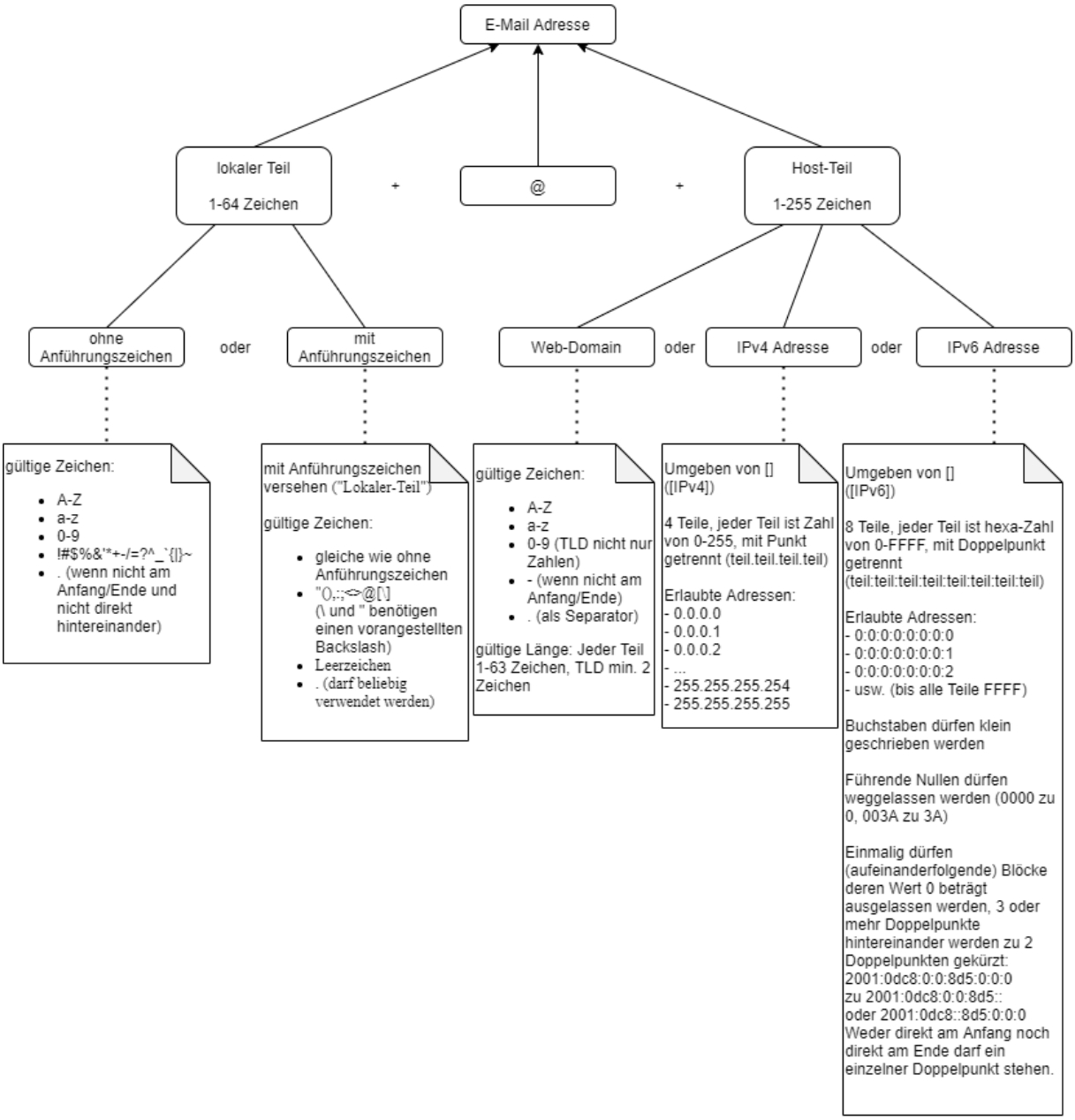
Dennoch gibt es aktuell noch keine Möglichkeit, mittels jqwik E-Mail-Adressen zu generieren und sie dem Programm als Testfall zu übergeben. Deswegen soll jqwik in dieser Arbeit um Generatoren für E-Mail-Adressen erweitert werden.

Um die Methode zur Generierung von E-Mail-Adressen zu erstellen, wurde in dieser Arbeit die iterative Entwicklung verwendet. Die Entwicklung besteht daher aus mehreren Phasen. Um jedoch ein besseres Verständnis zu gewährleisten, wurden in der schriftlichen Arbeit mehrere Phasen zusammengefasst. In Wirklichkeit war der Ablauf also etwas anders als hier dargestellt. Dies gilt auch für das nachfolgende Kapitel 4.

3.1 Anforderungsanalyse

Die von Johannes Link gestellte Anforderung ist es, dass Verwendern von jqwik eine einfache Methode bereitgestellt werden soll, die gültige E-Mail-Adressen erzeugen kann. (Link, 2020, 20. Oktober) Anschließend soll eine Annotation erstellt werden, mit deren Verwendung E-Mail-Adressen ohne eine zusätzlich bereitgestellte Methode generiert werden können. (Link, 2020, 11. November)

Dafür ist zunächst zu klären, was gültige E-Mail-Adressen nach der RFC 3696 Norm sind:



(Quellen der Abbildung: Klensin (2004), Elektronik Kompendium (2020) & Ionos (o. D.))

Der Grundlegende Aufbau einer E-Mail-Adresse ist ein lokaler Teil und ein Host-Teil, welche durch das @-Zeichen voneinander getrennt sind. (Klensin, 2004)

Im lokalen Teil, welcher zwischen einem und 64 Zeichen lang sein muss, sind nur die Zeichen `A` bis `Z`, `a` bis `z`, `0` bis `9`, sowie `!#$%&'*+-/=?^_`{|}~` erlaubt. Zusätzlich dürfen Punkte eingefügt werden, wenn sie weder am Anfang, am Ende noch direkt hintereinanderstehen. (Klensin, 2004)

Es besteht zusätzlich die Möglichkeit, den lokalen Teil mit Anführungszeichen (`"Lokaler-Teil"`) zu versehen. Dadurch sind zusätzlich die Zeichen `"(),:;<>@[\]` sowie das Leerzeichen erlaubt. Auch Punkte dürfen dann beliebig verwendet werden. Jedoch gilt es zu beachten, dass vor jedem Backslash und Anführungszeichen ein weiterer Backslash direkt davorstehen muss. (Klensin, 2004)

Der Host-Teil darf entweder aus einer normalen Web-Domain oder einer IP-Adresse bestehen und darf zwischen einem und 255 Zeichen fassen. Eine IP-Adresse muss von eckigen Klammern umgeben sein. (Klensin, 2004)

Eine Web-Domain darf dabei aus mehreren Teilen bestehen, wobei jeder Teil zwischen einem und 63 Zeichen lang sein muss. Für jqwik soll die Web-Domain eine Top-Level-Domain, also mindestens einen Punkt zwischen den Zeichen, beinhalten. Der letzte Teil (die Top-Level-Domain) muss mindestens zwei Zeichen lang sein. Dabei darf eine Web-Domain aus den Zeichen `A` bis `Z`, `a` bis `z`, `0` bis `9`, sowie einem Punkt als Separator der einzelnen Teile bestehen. Die Top-Level-Domain darf jedoch nicht nur aus Zahlen bestehen. Zusätzlich dürfen Bindestriche verwendet werden, wenn sie weder das erste noch das letzte Zeichen sind. (Klensin, 2004)

Befindet sich im Host-Teil eine IP-Adresse, so kann diese entweder eine IPv4-Adresse oder eine IPv6-Adresse sein. (Klensin, 2004)

Eine IPv4-Adresse setzt sich aus vier Teilen zusammen, die mit einem Punkt voneinander getrennt sind. Jeder Teil besteht aus einer Zahl von `0` bis `255`. So sind z. B. `0.0.3.201` und `255.0.133.227` gültige IPv4-Adressen. (Elektronik Kompendium, 2020)

Die IPv6-Adresse besteht aus acht Teilen, welche durch einen Doppelpunkt separiert sind. Dabei ist jeder Teil eine Hexadezimalzahl von `0` bis `FFFF`. Die Buchstaben dürfen auch kleingeschrieben werden. Um die Adresse möglichst kurzzuhalten, dürfen führende Nullen weggelassen werden. So kann z. B. aus `0000` eine einfache `0` und aus `003A` ein `3A` werden. Einmalig dürfen in der Adresse auch (aufeinanderfolgende) Nullblöcke weggelassen werden. Drei oder mehr Doppelpunkte in Folge werden zu zwei Doppelpunkten gekürzt. So kann die Adresse `2001:0dc8:0:0:8d5:0:0:0` zu `2001:0dc8:0:0:8d5::` oder `2001:0dc8::8d5:0:0:0` gekürzt werden. Es darf jedoch weder direkt am Anfang noch direkt am Ende der Adresse ein einzelner Doppelpunkt stehen. (Ionos, o. D.) Beispielsweise sind zwei weitere gültige IPv6-Adressen `F77d:aeF:99fF:e1A7:5f:079D:B9:F0bd` und `c3:4eE:9Fe9:c:a05C::`.

3.2 Phase 1: Erstellung eines E-Mail Arbitraries

Ziel der ersten Phase war es, einen Generator `EmailArbitrary` zu erstellen und seine Funktionalität zu implementieren. Der Generator soll am Ende dieser Phase gültige E-Mail-Adressen erzeugen können.

3.2.1 Entwicklung der Architektur und des Designs

Der Grundlegende Schritt dieser Phase war, die Klasse `Arbitraries` um eine Methode `emails` so zu erweitern, dass ein Objekt des Typs `EmailArbitrary` zurückgegeben wird. Dazu soll eine Klasse `DefaultEmailArbitrary` erstellt werden, die die Implementierung des `EmailArbitrary` übernimmt. Das `EmailArbitrary` soll zunächst keine Methoden enthalten, diese werden in einem späteren Schritt ergänzt. (siehe Anhang A)

Es sollen dadurch auf die folgende Weise E-Mail-Adressen generiert werden können:

```
@Provide
EmailArbitrary gueltigeEmailAdressen(){
     return Arbitraries.emails();
}
```

Hierbei wird eine Methode als Hilfe für die Tests kreiert, die gültige E-Mail-Adressen durch einen Aufruf der `emails` Methode bereitstellt.

3.2.2 Erweiterung des Designs mit entsprechenden Tests

Um die funktionalen Eigenschaften der Methode `emails` sicherzustellen, wurde eine Klasse `ArbitrariesEmailsTests` erzeugt. (siehe Anhang B) Diese Testklasse prüft mittels Property-Based Testing, ob die Methode korrekt funktioniert. Um die Übersicht zu gewährleisten, enthält die Klasse `ArbitrariesEmailsTests` mehrere innere Klassen, die sich dadurch unterscheiden, welche Art von Tests sie durchführen und die im Folgenden je mittels eines Beispieltests erklärt werden sollen.

Die wichtigste innere Klasse ist die Klasse `AllGeneratedEmailAddressesAreValid`. Sie kontrolliert, ob die erzeugten E-Mail-Adressen auch wirklich gültige Adressen sind. Eine dieser Testmethoden überprüft z. B. die korrekte Verwendung von Punkten vor dem `@`-Zeichen:

```
@Property
void validUseOfDotBeforeAt(@ForAll("emails") String email) {
     String localPart = getLocalPartOfEmail(email);
     Assume.that(!isQuoted(localPart));
     assertThat(localPart).doesNotContain("..");
     assertThat(localPart.charAt(0)).isNotEqualTo('.');
     assertThat(localPart.charAt(localPart.length() - 1)).isNotEqualTo('.');
}
```

Hier wird zunächst der lokale Teil vom Rest der E-Mail-Adresse separiert. Für den Fall, dass dieser ohne Anführungszeichen ist, wird geprüft, dass er keine zwei Punkte direkt hintereinander enthält und dass weder das erste noch das letzte Zeichen ein Punkt ist. Die weiteren Test-Methoden sind:

`containsAt`: Testet, ob ein `@`-Zeichen vorhanden ist

`validLengthBeforeAt`: Prüft die gültige Länge vor dem `@`-Zeichen

`validLengthAfterAt`: Überprüft die gültige Länge nach dem `@`-Zeichen

`validCharsBeforeAtUnquoted`: Kontrolliert, falls Anführungszeichen enthalten sind, ob nur gültige Zeichen vor dem `@`-Zeichen verwendet werden

`validCharsBeforeAtQuoted`: Testet, falls keine Anführungszeichen vorhanden sind, ob nur gültige Zeichen vor dem `@`-Zeichen verwendet werden

`validUseOfQuotedBackslashAndQuotationMarks`: Prüft bei Verwendung von Anführungszeichen, ob jeder Backslash und jedes Anführungszeichen (außer das erste und letzte) einen vorangestellten Backslash besitzen

`validCharsAfterAt`: Stellt sicher, dass nach dem `@`-Zeichen nur gültige Zeichen verwendet werden

`validUseOfHyphenAndDotAfterAt`: Überprüft die korrekte Verwendung von Bindestrichen und Punkten nach dem `@`-Zeichen

`validMaxDomainLengthAfterAt`: Prüft, ob jeder Teil einer Web-Domain die erlaubte Länge besitzt

`tldNotAllNumeric`: Testet, ob die Top-Level-Domain nicht nur aus Zahlen besteht

`validIPAddressAfterAt`: Verifiziert die Korrektheit von IP-Adressen

Eine weitere innere Klasse stellt die Klasse `CheckAllVariantsAreCovered` dar. Diese testet, ob alle möglichen Varianten auch wirklich vorkommen. Die folgende Methode prüft z. B., ob sowohl lokale Teile mit Anführungszeichen als auch ohne mit einer bestimmten Wahrscheinlichkeit vorkommen:

```
@Property
void quotedAndUnquotedUsernamesAreGenerated(@ForAll("emails") String email) {
     String localPart = getLocalPartOfEmail(email);
     Statistics.label("Quoted usernames")
          .collect(isQuoted(localPart))
          .coverage(coverage -> {
               coverage.check(true).percentage(p -> p > 35);
               coverage.check(false).percentage(p -> p > 35);
          });
}
```

Dabei wird zuerst der lokale Teil vom Rest der E-Mail-Adresse getrennt und anschließend wird in die Statistik „Quoted usernames“ aufgenommen, ob der lokale Teil in Anführungszeichen steht oder nicht. Zum Schluss wird kontrolliert, ob beide Varianten zu jeweils mindestens 35 Prozent vorkommen. Ein Test auf 50 Prozent ist nicht möglich, da die beiden Teile zufällig erzeugt und somit nicht immer gleichverteilt sind. Zusätzlich enthält die Testklasse die folgenden weiteren Tests:

`domainsAndIPAddressesAreGenerated`: Prüft, dass sowohl Web-Domains als auch IP-Adressen generiert werden

`IPv4AndIPv6AreGenerated`: Kontrolliert, ob sowohl IPv4- als auch IPv6-Adressen erzeugt werden

`domainHostsWithTwoAndMorePartsAreGenerated`: Überprüft, ob Web-Domains mit zwei oder mehr Teilen auch wirklich generiert werden

Die innere Klasse `ShrinkingTests` soll überprüfen, ob das Schrumpfen korrekt funktioniert. Sie besteht aus mehreren Testmethoden:

`defaultShrinking`: Prüft, ob im Standardfall zu `A@a.aa` geschrumpft wird

`domainShrinking`: Überprüft, ob Web-Domains zu `A@a.aa` geschrumpft werden

`domainShrinkingWithTLD`: Kontrolliert, ob Web-Domains mit Top-Level-Domain zu `A@a.aa` geschrumpft werden

`ipv4Shrinking`: Verifiziert, ob IPv4-Adressen zu `A@[0.0.0.0]` geschrumpft werden

`ipv6Shrinking`: Testet, ob der Host-Teil von IPv6-Adressen zu `::` geschrumpft wird

3.2.3 Implementierung in jqwik

Nebenbei wurde die Funktionalität in der Klasse `DefaultEmailArbitrary` umgesetzt. (siehe Anhang B) Die Methode `arbitrary` lässt dabei einen gültigen lokalen Teil, der in Anführungszeichen gestellt oder nicht mit Anführungszeichen versehen sein kann, sowie einen Host-Teil, der entweder eine IP-Adresse oder eine Web-Domain ist, generieren und setzt diese durch ein `@`-Zeichen getrennt zusammen.

So werden beispielsweise mit folgender Methode die IPv6-Adressen erzeugt:

```
private Arbitrary<String> domainIPv6() {
     Arbitrary<String> addressPart =
          Arbitraries.strings()
               .numeric().withChars('a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C',
                         'D', 'E', 'F')
               .ofMaxLength(4);
     Arbitrary<String> address =
          Combinators.combine(addressPart, addressPart, addressPart, addressPart,
                    addressPart, addressPart, addressPart, addressPart)
               .as((a, b, c, d, e, f, g, h) -> "[" + a + ":" + b + ":" + c + ":" +
                         d + ":" + e + ":" + f + ":" + g + ":" + h + "]");
     address = address.map(v -> removeThreeOrMoreColons(v));
     address = address.filter(v -> validUseOfColonInIPv6Address(v.substring(1,
               v.length() - 1)));
     return address;
}
```

Dabei wird zuerst ein `Arbitrary<String>` erzeugt, der nur aus Zahlen sowie den Zeichen `abcdefABCDEF` besteht und die maximale Zeichenlänge vier besitzt. Dieser Generator erzeugt einen Teil der IPv6-Adresse. Mittels eines `Combinators` wird der Generator nun so zu einem neuen Generator kombiniert, dass eine möglicherweise noch ungültige achtteilige IPv6-Adresse entsteht, die von eckigen Klammern umgeben ist. Da die Adresse aktuell noch drei oder mehr Doppelpunkte in Folge haben kann, werden diese Doppelpunkte mithilfe der `map` und der implementierten `removeThreeOrMoreColons` Methode zu zwei Doppelpunkten gekürzt. Anschließend werden noch ungültige IP-Adressen, die Doppelpunkte falsch benutzen, mittels der `filter` und der dafür erstellten `validUseOfColonInIPv6Address` Methode herausgefiltert. Zum Schluss wird ein Generator für gültige IPv6-Adresse zurückgegeben.

3.3 Phase 2: Erstellung einer E-Mail-Annotation

Im Anschluss ist eine Annotation erstellt worden, mittels der E-Mail-Adressen ohne zusätzliche Hilfsmethoden generiert werden können. Zusätzlich soll es möglich sein, dass nur E-Mail-Adressen mit bestimmten Eigenschaften generiert werden.

3.3.1 Entwicklung der Architektur und des Designs

Das Ziel dieser Phase war es, eine `@Email` Annotation zu erstellen, die einer Property gültige E-Mail-Adressen bereitstellt. Dabei soll die Annotation auch durch Parameter eingeschränkt werden können, sodass beispielsweise nur IPv6-Adressen erzeugt werden. Zusätzlich soll auch das `EmailArbitrary` Interface um solche Einschränkungsmethoden erweitert werden. (siehe Anhang A)

Die Annotation soll z. B. auf die folgende Art und Weise verwendet werden können:

```
@Property
void testeEmailAdressen(
     @ForAll @Email(ipv6Addresses = false, ipv4Addresses = false) String email){
     //mache etwas
}
```

So werden der Methode `testeEmailAdressen` als Parameter gültige E-Mail-Adressen übergeben, die in diesem Fall keine IP-Adressen enthalten.

Des Weiteren soll beispielsweise die folgende Einschränkung möglich sein:

```
@Provide
EmailArbitrary generiereEmailAdressen(){
     return Arbitraries.emails().quotedLocalParts();
}
```

Hierbei wird ein Generator zurückgegeben, der nur E-Mail-Adressen in Anführungszeichen erzeugt.

3.3.2 Erweiterung des Designs mit entsprechenden Tests

Die Klasse `ArbitrariesEmailsTests` ist um die innere Klasse `CheckEmailArbitraryMethods` erweitert worden, um die Methoden des Interfaces `EmailArbitrary` zu testen. Zusätzlich wurde eine Testklasse `EmailProperties` erstellt, um die Annotation zu überprüfen. (siehe Anhang B) Dadurch sollen die neuen funktionellen Eigenschaften dieses Schrittes sichergestellt werden.

Die Klasse `CheckEmailArbitraryMethods` bekommt dabei mehrere Testmethoden. Eine dieser Methoden überprüft z. B., ob der Host-Teil nur IPv4-Adressen enthält:

```
@Property
void onlyIPv4AddressesAreGenerated(@ForAll("onlyIPv4Addresses") String email) {
     String domain = getDomainOfEmail(email);
     assertThat(isIPAddress(domain)).isTrue();
     assertThat(domain).contains(".");
}
@Provide
private EmailArbitrary onlyIPv4Addresses() {
     return Arbitraries.emails().ipv4Addresses();
}
```

Die Methode `onlyIPv4Addresses()` generiert nur E-Mail-Adressen die IPv4-Adressen im Host-Teil haben und soll dahingehend geprüft werden. Die Methode `onlyIPv4AddressesAreGenerated` erhält die zu testenden Adressen als Parameter. Sie trennt zuerst den Host-Teil vom Rest der Adresse und überprüft, ob dieser eine IP-Adresse ist. Wenn dies der Fall ist, wird anschließend getestet, ob die IP-Adresse einen Punkt enthält. Ist dies der Fall, handelt es sich um eine gültige IPv4-Adresse, andernfalls schlägt der Test fehl. Weiter enthält die Klasse die folgenden Tests:

`onlyIPAddressesAreGenerated`: Prüft, ob das Erzeugen ausschließlich von IP-Adressen im Host-Teil funktioniert

`onlyIPv6AddressesAreGenerated`: Überprüft, ob das Generieren allein von IPv6-Adressen im Host-Teil funktioniert

`onlyDomainsAreGenerated`: Testet, ob das Erstellen lediglich von Web-Domains im Host-Teil funktioniert

`onlyQuotedLocalPartsAreGenerated`: Prüft, ob das Erzeugen allein von in Anführungszeichen gestellten lokalen Teilen funktioniert

`onlyUnquotedLocalPartsAreGenerated`: Testet, ob das Generieren ausschließlich von nicht in Anführungszeichen gestellten lokalen Teilen funktioniert

Auch die Klasse `EmailProperties` enthält zwei innere Klassen. Die innere Klasse `checkAnnotationProperties` überprüft dabei, ob die Parameter der `@Email` Annotation korrekt funktionieren. So testet folgende Methode, ob lokale Teile ohne Anführungszeichen generiert werden:

```
@Property
void onlyUnquotedLocalPartsAreGenerated(@ForAll @Email(quotedLocalPart = false)
          String email) {
     String localPart = getLocalPartOfEmail(email);
     assertThat(isQuoted(localPart)).isFalse();
}
```

Die Annotation in der Methode erhält dabei die Anweisung, dass keine mit Anführungszeichen versehenen lokalen Teile generiert werden sollen. In der Methode selbst wird der lokale Teil vom Rest getrennt und daraufhin geprüft, ob er ohne Anführungszeichen ist. Ist er mit Anführungszeichen versehen, schlägt der Test fehl. Die weiteren Methoden dieser Klasse sind:

`onlyIPAddressesAreGenerated`: Testet, ob der Host-Teil nur IP-Adressen enthält

`onlyIPv4AddressesAreGenerated`: Überprüft, ob der Host-Teil nur IPv4-Adressen beinhaltet

`onlyIPv6AddressesAreGenerated`: Prüft, ob der Host-Teil nur aus IPv6-Adressen besteht

`onlyDomainsAreGenerated`: Testet, ob der Host-Teil nur Web-Domains beinhaltet

`onlyQuotedLocalPartsAreGenerated`: Prüft, ob der lokale Teil mit Anführungszeichen versehen ist

Zusätzlich hat die Klasse `EmailProperties` die innere Klasse `CheckAllVariantsAreCovered`, die überprüft, ob die Annotation ohne Parameter auch alle möglichen Varianten abdeckt. Bei der folgenden Methode wird z. B. überprüft, ob im Host-Teil sowohl Web-Domains als auch IP-Adressen generiert werden:

```
@Property
void domainsAndIPAddressesAreGenerated(@ForAll @Email String email) {
     String domain = getDomainOfEmail(email);
     Statistics.label("Domains")
          .collect(isIPAddress(domain))
          .coverage(coverage -> {
               coverage.check(true).percentage(p -> p > 35);
               coverage.check(false).percentage(p -> p > 35);
          });
}
```

Dabei wird zunächst der Host-Teil vom Rest der Adresse getrennt und in die Statistik „Domains“ aufgenommen, ob dieser eine IP-Adresse oder eine Web-Domain ist. Anschließend wird überprüft, ob beide Varianten zu je mindestens 35 Prozent generiert werden. Auch hier ist wegen der zufälligen Verteilung kein Test auf 50 Prozent möglich. Weitere Methoden dieser Klasse sind:

`quotedAndUnquotedUsernamesAreGenerated`: Kontrolliert, ob sowohl in Anführungszeichen gestellte als auch lokale Teile ohne Anführungszeichen erzeugt werden

`IPv4AndIPv6AreGenerated`: Testet, ob IPv4- und IPv6-Adressen generiert werden

3.3.3 Implementierung in jqwik

Daneben wurde das vorher designte umgesetzt. (siehe Anhang B) Dafür wurde die Klasse `DefaultEmailArbitrary` erweitert und sowohl eine `@Email` Annotation als auch ein `EmailArbitraryProvider` erstellt, welcher später genauer erklärt wird.

Die Klasse `DefaultEmailArbitrary` implementiert nun die Methoden des `EmailArbitrary` Interfaces. So kann sie steuern, welche Teile generiert werden sollen und welche nicht:

```
@Override
public EmailArbitrary quotedLocalParts() {
     DefaultEmailArbitrary clone = typedClone();
     clone.allowQuotedLocalPart = true;
     return clone;
}
```

Wenn nun z. B. die Methode `quotedLocalParts` aufgerufen wird, wird zuerst eine Kopie des aktuellen `EmailArbitrary` erzeugt. Diesem Klon wird mitgeteilt, dass lokale Teile mit Anführungszeichen erzeugt werden sollen. Anschließend wird der Klon zurückgegeben. Dies funktioniert äquivalent auch für die anderen Methoden des `EmailArbitrary`.

Des Weiteren wurden in der Klasse `DefaultEmailArbitrary` die Methoden so angepasst, dass evtl. bestimmte Teile nicht erzeugt werden. Dies funktioniert beispielsweise wie folgt:

```
private Arbitrary<String> localPart(){
     if(!allowUnquotedLocalPart && !allowQuotedLocalPart){
          allowUnquotedLocalPart = true;
          allowQuotedLocalPart = true;
     }
     Arbitrary<String> unquoted = localPartUnquoted();
     Arbitrary<String> quoted = localPartQuoted();
     int frequencyUnquoted = allowUnquotedLocalPart ? 1 : 0;
     int frequencyQuoted = allowQuotedLocalPart ? 1 : 0;
     return Arbitraries.frequencyOf(
          Tuple.of(frequencyUnquoted, unquoted),
          Tuple.of(frequencyQuoted, quoted)
     );
}
```

Diese Methode erzeugt gültige lokale Teile. Dabei wird zuerst überprüft, ob das Erzeugen ohne Anführungszeichen sowie in Anführungszeichen gestellter lokaler Teile `false` ist. Dies ist der

Fall, wenn weder die Methode `quotedLocalParts` noch `unquotedLocalParts` aufgerufen wurde. In diesem Fall werden beide Werte auf `true` gesetzt, damit etwas generiert wird. Anschließend wird ein Generator für nicht in Anführungszeichen gestellte und einer für in Anführungszeichen gestellte lokale Teile erzeugt. Danach wird seine Häufigkeit festgelegt: Er bekommt den Wert eins, wenn er erzeugt werden soll und den Wert null, wenn er es nicht soll. Zum Schluss wird ein Generator zurückgegeben, der die beiden anderen Generatoren entsprechend ihrer Häufigkeit verwendet. Dadurch ist es möglich, dass z. B. nur von Anführungszeichen umgebene lokale Teile generiert werden.

Die Annotation `@Email` wird erstellt und legt die Parameter fest, die ihr übergeben werden können.

Der `EmailArbitraryProvider` (siehe Anhang A) sorgt dafür, dass, wenn die Annotation bei einem `String` verwendet wird, auch entsprechend E-Mail-Adressen generiert werden. Er erzeugt ein Objekt des Typs `EmailArbitrary` und wendet, je nachdem welche Parameter der Annotation übergeben wurden, die entsprechenden Methoden des `EmailArbitrary` an. Das Ganze basiert auf dem Konzept des Service Provider Interfaces, das dafür sorgt, dass das entsprechende Modul zur Laufzeit gefunden und zur Verfügung gestellt wird. (Epple, o. D.) Dieser Service Provider kommt in Kombination mit einem Service Loader zum Einsatz, welcher die Implementierungen erkennt und anschließend lädt und somit den Grundbaustein des Service Provider Interfaces darstellt. (baeldung, 2020)

3.4 Bewertung gegenüber den Anforderungen

Das entstandene Programm erfüllt die beiden Anforderungen aus der Anforderungsanalyse. Mit der Methode `emails` sowie mit der Annotation `@Email` können nun auf einfache Art und Weise alle gültigen E-Mail-Adressen vom Nutzer generiert werden. Ein Kritikpunkt des Programms in seiner jetzigen Fassung ist jedoch, dass es sehr viele Randfälle gibt, die von jqwik bevorzugt generiert werden und dadurch die gleichmäßige Verteilung der verschiedenen Varianten sowie von zufällig erzeugten E-Mail-Adressen beeinflusst wird. Dies soll in einer späteren Version von jqwik überarbeitet werden, ist jedoch nicht mehr Bestandteil der Arbeit.

4 Erweiterung von jqwik um die Generierung von Daten und Zeiten

Das Jahr 1999 versetzte viele Menschen in Panik. Sie befürchteten all ihr Geld auf der Bank zu verlieren und dass ihre elektronischen Geräte den Geist aufgeben könnten. Viele Experten warnten sogar davor, dass es zu Kernschmelzen in Atomkraftwerken kommen könnte oder alle Telekommunikationsnetze der Welt zusammenbrechen. Es wurde sich auch davor gefürchtet, dass innerhalb der Bevölkerung große Unruhen aufkommen. Und das alles nur wegen dem Jahreswechsel zum Jahr 2000, in dem der sogenannte „Y2K"-Bug befürchtet wurde. Dieser trat dann tatsächlich auch ein, jedoch weit weniger schlimm als erwartet: Uhren funktionierten nicht mehr, Neugeborene bekamen als Geburtsjahr 1900 eingetragen, Automaten streikten, Mahnungen wurden wegen einer Überfälligkeit von 100 Jahren verschickt und weitere kleinere Komplikationen traten auf. Das alles führte weltweit zu Schäden in Milliardenhöhe. Und auch im Februar 2000 kam es erneut zu Fehlern, da viele Maschinen den 29. Februar 2000 übersprangen, der, im Gegensatz zu dem im Jahr 1900, durchaus existierte. Dennoch waren die Folgen weitaus geringer als befürchtet. (DER SPIEGEL, 2007)

Bald steht der Menschheit ein ähnliches Problem bevor: Oft werden Daten in Sekunden, die seit dem 1. Januar 1970 vergangen sind, dargestellt. Dies geschieht häufig als vorzeichenbehaftete 32 Bit-Zahl. Im Jahr 2038 wird der 32-Bit-Raum jedoch vollständig ausgenutzt sein und es wird zu einem Überlauf kommen. Aus der größtmöglichen positiven wird die kleinstmögliche negative 32-Bit-Zahl. Oder anders gesagt: Nach dem 19. Januar 2038 um 03:14:07 Uhr kommt der 13. Dezember 1901. Noch ist genügend Zeit sich darauf vorzubereiten und neuere Systeme z. B. auf 64-Bit-Zahlen umzustellen. Doch besonders bei älterer Hardware kann dies dennoch zu Problemen führen. (Parthesius, 2020)

Ob Y2K- oder Y2K38-Bug: Beides liese sich verhindern, indem man ausreichend im Voraus testet. Jqwik bietet jedoch bislang keine Möglichkeit Datumsobjekte zu generieren. Genauso ist es nicht möglich, dass automatisch Uhrzeiten erzeugt werden. In dieser Arbeit soll jqwik dahingehend erweitert werden, dass sowohl Generatoren für Daten und Zeiten, sowie die Kombination aus beidem erstellt werden kann.

4.1 Anforderungsanalyse

Die für dieses Problem gestellte Anforderung von Johannes Link ist es, dass die Nutzer von jqwik eine einfache Möglichkeit bekommen sollen, Daten und Uhrzeiten zu erzeugen. Dies müssen Nutzer bislang aus primitiven Datentypen selbst machen, um z. B. aus mehreren Zahlengeneratoren eine Uhrzeit zu basteln. Jqwik soll dafür um das neue Modul `time` erweitert werden. Dieses soll alle im JDK vorhandenen Typen von Zeiten und Daten abdecken. Dazu zählen auch Zeitspannen, etc. Zusätzlich sollen sinnvolle Methoden bereitgestellt werden, um die generierten Werte einzuschränken. Dies soll zusätzlich über Annotationen möglich gemacht werden. (Link, 2020, 9. Dezember).

Zunächst sind hierfür grundsätzliche Dinge zu regeln: In der traditionell verwendeten Zeitrechnung existiert das Jahr null nicht. Nach dem 31. Dezember 1 v. Chr. folgte der 1. Januar 1 n. Chr. (Was War Wann?, o. D.) Deswegen soll jqwik das Jahr null nicht erzeugen. Für das Generieren von Daten wurde zudem beschlossen, dass nur positive Jahreszahlen möglich sein sollen. Dies soll allerdings nicht den Generator für Jahreszahlen betreffen. Dieser soll bei Bedarf auch negative Jahreszahlen kreieren können. Es soll zudem auch ignoriert werden, dass der Gregorianische Kalender erst am 15. Oktober 1582 begann und dabei zehn Kalendertage ausgelassen wurden. (welt, 2010, 12. September) Da beim Testen zudem meistens nur aktuelle oder zeitnahe Daten benötigt werden, soll jqwik im Standardfall nur Jahreszahlen von 1900 bis 2500 generieren.

Auch wenn es kaum bekannt ist, gibt es in der Zeitrechnung neben Schaltjahren auch unregelmäßig auftauchende Schaltsekunden. Dabei existiert z. B. die Uhrzeit 23:59:60. (Bikos & Buckle, o. D.) Da die gängigen Java-Typen für Zeiten dies allerdings nicht unterstützen, sollen Schaltsekunden in jqwik nicht berücksichtigt werden. Die Zeitumstellung soll in dieser Arbeit ebenfalls nicht beachtet werden.

4.2 Entwicklung eines ersten Designs

Um herauszufinden, welche Methoden für das Modul von Bedeutung sind, wurden zunächst zwei Beispielapplikationen mit einzelnen datums- und zeittypischen Eigenschaften überlegt, die diese Apps erfüllen sollen. Anschließend wurden nach einer Recherche weitere Funktionen ergänzt.

Die erste Beispielanwendung ist eine Kalender-Planungs-App (siehe Anhang C), bei welcher der Nutzer Ereignisse eintragen kann, die in der Zukunft liegen. Es soll zudem eine Anzeige geben, wie lange es noch bis zu diesem Ereignis dauert. So könnten z. B. die Serverwartungsarbeiten am 20.10.2021 um 14:30:15 erfasst werden. Dafür ist es wichtig, dass alle Ereignisse in der Zukunft liegen. Es wird also eine Methode benötigt, mit der man ein minimales Datum festlegt. Für den Testfall der Subtraktionsmethode, welche anzeigt wie lange es noch bis zum Ereignis dauert, werden lediglich zwei Datumswerte benötigt, die aber nicht weiter eingeschränkt werden müssen.

Eine zweite Beispielapplikation ist ein Reservierungs-Tool für Restaurants (siehe Anhang D). Reservierungen sollen nur an Daten möglich sein, an denen das Restaurant geöffnet hat. Sowohl am Dienstag als auch am Donnerstag hat das Restaurant Ruhetag. Auch hier müssen alle Reservierungen in der Zukunft liegen, wodurch deutlich wird, dass eine min-Methode auf jeden Fall erforderlich ist. Des Weiteren müssen alle generierten Daten an bestimmten Wochentagen sein. Hierfür ist also eine Funktion nötig, mit der man die Wochentage einschränken kann. Auch hat das Restaurant eine bestimmte Öffnungszeit. Es wird also eine Methode gebraucht, welche die generierten Uhrzeiten einschränkt.

Nach einer Recherche, welche Funktionen noch sinnvoll wären, wurden die durch die Beispielanwendungen zusammengetragenen Methoden um weitere Funktionen ergänzt (siehe Anhang E): So soll es zusätzlich auch möglich sein, in jqwik ein maximales Datum festzulegen. Des Weiteren sollen auch minimale sowie maximale Jahreszahlen, Monate, Tage, Stunden, Minuten und Sekunden zur weiteren Einschränkung festgelegt werden können. Auch soll es möglich sein, Perioden und Zeitintervalle zu erzeugen.

Die hier zusammengetragenen Methoden (siehe Anhang E) stellen allerdings nur einen ersten Entwurf dar und sollen in späteren Phasen bei Bedarf noch erweitert werden.

4.3 Phase 1: Daten

In den folgenden Schritten sind zuerst Generatoren für Daten (ohne Zeiten) und datumsbasierte Klassen implementiert worden. Für eine bessere Verständlichkeit werden die einzelnen Phasen hier ggfs. in einer anderen Reihenfolge dargestellt als sie tatsächlich umgesetzt wurden. Da hierfür die einzelnen Schritte innerhalb einer Phase fast identisch mit denen aus anderen Phasen sind, wurden die Schritte in dieser Arbeit zum Teil gekürzt oder weggelassen. Diese Punkte gelten auch für die nachfolgenden Kapitel.

4.3.1 Phase 1.1: YearArbitrary

Zunächst wurde ein Generator `YearArbitrary` für Jahreszahlen des Typs `Year` umgesetzt. Dieser soll am Ende der Phase gültige Jahreszahlen (also ohne das Jahr `0`) erzeugen können.

4.3.1.1 Entwicklung der Architektur und des Designs

Ziel dieser Phase ist es eine neu erstellte Klasse `Dates` um eine Methode `years` zu ergänzen. Es soll ein Objekt des Typs `YearArbitrary` zurückgegeben werden. Dafür ist eine Klasse `DefaultYearArbitrary` nötig, die die Implementierung des Interfaces `YearArbitrary` übernimmt. (siehe Anhang G)

Zum Schluss soll es wie folgt möglich sein, zufällige Jahreszahlen zu generieren:

```
@Provide
Arbitrary<Year> gueltigeJahreszahlen() {
     return Dates.years();
}
```

Mittels einer Methode `between` soll es möglich sein, die generierten Jahre einzuschränken.

4.3.1.2 Erweiterung des Designs mit entsprechenden Tests

Die funktionellen Eigenschaften dieses Moduls sollen mittels eines neu angelegten Packages `dates.year` mittels PBT überprüft werden. (siehe Anhang F) Da das vierte Kapitel über 1.000 Tests aufgrund der sehr hohen Testabdeckung enthält, wurden die Tests auf mehrere Klassen innerhalb ihres zugehörigen Packages verteilt. Dies gilt auch für alle nachfolgenden Kapitel. Dabei wurde jeweils mindestens eine Klasse für jede Testart erstellt. War die Klasse zu groß, wurde sie in einem eigenen Package in mehrere Klassen unterteilt. Zudem werden für eine bessere Übersicht im gesamten vierten Kapitel nur wenige für ihr Package typische Tests dargestellt. Alles andere würde über den Rahmen dieser Arbeit hinausgehen. Die Tests aus den verschiedenen Packages sind sich zum Teil sehr ähnlich.

So wurde beispielsweise der folgende Test in der Klasse `SimpleArbitrariesTests` umgesetzt:

```
@Property
void yearIsNotZero(@ForAll("yearsAround0") Year year) {
     assertThat(year).isNotEqualTo(Year.of(0));
}
@Provide
Arbitrary<Year> yearsAround0() {
     return Dates.years().between(-10, 10);
}
```

Dabei werden die generierten Jahre, die um das Jahr `0` liegen, daraufhin getestet, ob sie sich vom Jahr `0` unterscheiden und dieses somit nicht erzeugt wird.

Weitere Tests sind:

`YearMethodsTests:between`: Überprüft, ob die `between` Methode korrekt funktioniert

`ShrinkingTests:defaultShrinking`: Verifiziert, dass der Wert zu 1900 (im Standardfall minimal erzeugtes Jahr) geschrumpft wird

`ExhaustiveGenerationTests:between`: Prüft, ob alle von dem Jahr `-5` bis zum Jahr `5` möglichen Jahreszahlen (ohne das Jahr `0`) generiert werden

`EdgeCasesTests:all`: Erprobt die korrekte Generierung der Randfälle

4.3.1.3 Implementierung in jqwik

Gleichzeitig wurde die Umsetzung der Klasse `DefaultYearArbitrary` durchgeführt. (siehe Anhang F) Dabei generiert die Methode `arbitrary` gültige Jahreszahlen, die im Standardfall von 1900 bis 2500 gehen:

```
public class DefaultYearArbitrary extends ArbitraryDecorator<Year> implements YearArbitrary {
//mehr Code
@Override
protected Arbitrary<Year> arbitrary() {
     Year min = yearBetween.getMin() != null ? yearBetween.getMin() : DEFAULT_MIN;
     Year max = yearBetween.getMax() != null ? yearBetween.getMax() : DEFAULT_MAX;
     Arbitrary<Integer> years = Arbitraries.integers()
                                          .between(min.getValue(), max.getValue())
                                          .filter(v -> v != 0);
     return years.map(Year::of);
}
//mehr Code
}
```

Hier werden zunächst das minimal und maximal mögliche Jahr berechnet und anschließend die möglichen Zahlen mittels eines Generators `Arbitrary<Integer>` erzeugt. Danach wird das Jahr 0 herausgefiltert. Zum Schluss wird der Generator mittels der `map` Funktion in den Typ `Arbitrary<Year>` umgewandelt.

4.3.1.4 Design und Implementierung der Default Generation

Es soll möglich sein, Jahreszahlen ohne eine `@Provide` Methode über den Typen Year zu generieren. Hierfür wurde ein `YearArbitraryProvider` designt (siehe Anhang J) und programmiert. (siehe Anhang F) Die Default Generation muss zusätzlich in der Datei `META-INF/services/net.jqwik.api.providers.ArbitraryProvider` registriert werden. (jqwik, o. D. a)

Damit die korrekte Funktionsweise sichergestellt ist, wurde beispielsweise der folgende Test ergänzt:

`DefaultGenerationTests:yearIsNotNull:` Prüft die Gültigkeit der erzeugten Werte

4.3.1.5 Design und Implementierung einer YearRange

Anschließend wurde eine Annotation `YearRange` entworfen (siehe Anhang H und I) und implementiert. (siehe Anhang F) Dadurch soll es möglich sein, eine minimale und maximale Jahreszahl mittels Annotation festzulegen. Die Ranges müssen zudem in der Datei `META-INF/services/net.jqwik.api.configurators.ArbitraryConfigurator` registriert werden.

Sie soll z. B. auf folgende Weise verwendet werden können:

```
@Property
void testeJahreszahlen(@ForAll @YearRange(min = 3000, max = 3500) Year jahr){
     //mache etwas
}
```

Hierbei würden nur Jahre von 3000 bis 3500 generiert werden.

Um die Funktionalität zu gewährleisten, wurde beispielsweise folgender Test hinzugefügt:

`ConstraintTests$Constraints:yearRangeBetweenMinus100And100:` Testet, ob nur die Werte generiert werden, die per Annotation möglich gemacht werden

Zugleich wurde die Implementierung umgesetzt. Dafür wurde ein `YearRangeConfigurator` mit einer entsprechenden inneren Klasse erzeugt, welche die Einschränkungen an das `YearArbitrary` weitergibt.

4.3.2 Phase 1.2: Monate, Wochentage und Tage

In der nächsten Phase wurden einfache Generatoren für Objekte des Typs `Month` und `DayOfWeek`, sowie für Tage (repräsentiert als `Integer`) realisiert. Diese sollen jedoch aufgrund der Einfachheit keine eigenen Interfaces erhalten.

4.3.2.1 Entwicklung der Architektur und des Designs

Zu Beginn wurde die Klasse `Dates` mit den Methoden `months`, `daysOfWeek` und `daysOfMonth` erweitert. Zurückgegeben werden sollen Objekte des Typs `Arbitrary<Month>`, `Arbitrary<DayOfWeek>` bzw. `Arbitrary<Integer>`. (siehe Anhang G)

Möglichgemacht werden soll, dass die Generatoren wie folgt verwendet werden können:

```
@Provide
Arbitrary<Month> gueltigeMonate() {
     return Dates.months();
}
@Provide
Arbitrary<DayOfWeek> gueltigeWochentage() {
     return Dates.daysOfWeek();
}
@Provide
Arbitrary<Integer> gueltigeTage() {
     return Dates.daysOfMonth();
}
```

Eine Einschränkung der Generatoren soll nicht über zusätzlich erstellte Methoden möglich sein. Dies kann der Nutzer sehr leicht mittels geeigneter Filter umsetzen.

4.3.2.2 Erweiterung des Designs mit entsprechenden Tests

Aufgrund der Einfachheit des Moduls ist jeweils nur ein Test nötig. Dieser wird in das für später gebrauchte Package für `LocalDate` untergebracht. Dafür wird das Package `dates.localDates` neu erzeugt. (siehe Anhang F)

Es wird der folgende Test in die Klasse `SimpleArbitrariesTests` hinzugefügt:

```
@Property
void validMonthIsGenerated(@ForAll("months") Month month) {
     assertThat(month).isNotNull();
}
@Provide
Arbitrary<Month> months() {
     return Dates.months();
}
```

Dabei werden Objekte des Typs `Month` erzeugt und anschließend daraufhin geprüft, ob sie gültig sind.

Zusätzlich bekommt die Klasse zwei ähnlich aufgebaute weitere Tests:

`validDayOfWeekIsGenerated:` Prüft, ob erzeugte Wochentage gültig sind

`validDayOfMonthIsGenerated:` Testet die Gültigkeit von generierten Tagen

4.3.2.3 Implementierung in jqwik

Auch die Implementierungen wurden durchgeführt (siehe Anhang F):

```
public static Arbitrary<Month> months() {
     return Arbitraries.of(Month.class);
}
public static Arbitrary<DayOfWeek> daysOfWeek() {
     return Arbitraries.of(DayOfWeek.class);
}
public static Arbitrary<Integer> daysOfMonth() {
     return Arbitraries.integers()
                       .between(1, 31)
                       .edgeCases(edgeCases -> edgeCases.includeOnly(1, 31));
}
```

Hierbei wurde die `Arbitraries.of` Methode zu Hilfe genommen, welche es ermöglicht Enum-Werte zu generieren. Für die Tage wurde der schon vorhandene Generator für Ganzzahlen verwendet und entsprechend modifiziert, so dass nur Werte von `1` bis `31` generiert werden können.

4.3.2.4 Design und Implementierung einer DayOfMonthRange

Im Anschluss wurde eine Annotation `DayOfMonthRange` designt (siehe Anhang H und I) und implementiert. (siehe Anhang F) So soll es möglich werden, generierte Integer-Werte auf gültige Tage einzuschränken und Grenzen für die erzeugten Werte festzulegen.

Dies soll folgendermaßen möglich gemacht werden:

```
@Property
void gueltigeTage(@ForAll @DayOfMonthRange int tag){
     //mache etwas
}
```

```
@Property
void gueltigeTageZwischen15Und20(
     @ForAll @DayOfMonthRange(min = 15, max = 20) int tag
) {
     //mache etwas
}
```

So ist es einerseits möglich sehr leicht Zahlen von 1 bis 31 zu generieren, als auch den Tagen eine entsprechende Einschränkung zu geben.

Um die Funktionalität zu testen, wurde ein neues Package `dates.dayOfMonth` mit der Klasse `ConstraintTests` ergänzt, die z. B. folgenden Test enthält:

`InvalidUseOfConstraints:dayOfMonthRange`: Überprüft, dass bei Verwendung eines nicht korrekten Typs (hier `Random`) `null` zurückgegeben wird

4.3.3 Phase 1.3: LocalDate

Anschließend wurde der Generator `LocalDateArbitrary` für Objekte des Typs `LocalDate` umgesetzt. Dieser soll nur positive Jahre erzeugen können.

4.3.3.1 Entwicklung der Architektur und des Designs

Die Klasse `Dates` ist erneut um eine weitere Methode `dates` ergänzt worden, die einen Generator des Typs `LocalDateArbitrary` zurückgibt. Hierfür wurde ein `DefaultLocalDateArbitrary` erstellt, welches die Implementierung des Interfaces `LocalDateArbitrary` umsetzt. (siehe Anhang G)

Es soll folgenderweise möglich sein Daten zu generieren:

```
@Provide
Arbitrary<LocalDate> dates() {
     return Dates.dates();
}
```

Zudem sollen die Methoden `between`, `atTheEarliest`, `atTheLatest`, `yearBetween`, `monthBetween`, `onlyMonths`, `dayOfMonthBetween` und `onlyDaysOfWeek` die generierten Werte einschränken können.

4.3.3.2 Erweiterung des Designs mit entsprechenden Tests

Um sicherzustellen, dass dieses Modul korrekt funktioniert, wurde das Package `dates.localDates` um weitere Tests erweitert. (siehe Anhang F) So wurde z. B. folgender Test in der Klasse `InvalidConfigurationTests` hinzugefügt:

```
@Property
void monthWithout31DaysButDayOfMonth31(
     @ForAll("monthWithout31Days") @Size(min = 1) Set<Month> months
) {
     assertThatThrownBy(
          () -> Dates.dates()
                   .onlyMonths(months.toArray(new Month[]{}))
                   .dayOfMonthBetween(31, 31)
                   .generator(1)
     ).isInstanceOf(IllegalArgumentException.class);
}
@Provide
Arbitrary<Set<Month>> monthWithout31Days() {
     Arbitrary<Month> months =
               Arbitraries.of(FEBRUARY, APRIL, JUNE, SEPTEMBER, NOVEMBER);
     return months.set();
}
```

Bei diesem Test werden `Sets` von Monaten erzeugt, von denen keiner 31 Tage hat und dem Test als Parameter übergeben. Es wird nun geprüft, ob die Einschränkung von Monaten auf dieses `Set` in Kombination mit der Einschränkung, dass nur der 31. Tag des Monats generiert werden darf, eine `IllegalArgumentException` zur Folge hat. Falls nicht, schlägt der Test fehl.

Weitere Tests sind z. B.:

`SimpleArbitrariesTests:validLocalDateIsGenerated:` Testet, ob gültige Daten erzeugt werden

`DateMethodTests$DayOfMonthsMethods:dayOfMonthBetween:` Überprüft, ob die Einschränkung für Tage bei generierten Daten funktioniert

`EdgeCasesTests:between:` Verifiziert die korrekte Erstellung von Randfällen beim Anwenden der `between` Methode

`ExhaustiveGenerationTests:dayOfMonthBetweenAndBetween`: Prüft, dass bei Verwendung von `dayOfMonthBetween` und `between` nur die erwarteten Werte generiert werden können

`EqualDistributionTests:months:` Überprüft, ob die generierten Monate gleichmäßig verteilt sind

`ShrinkingTests:shrinksToSmallestFailingValue:` Testet den Generator daraufhin, dass bei einem Fehler zum kleinsten fehlschlagenden Wert geschrumpft wird

`ConstraintTests$Constraints:yearRangeBetween500And700:` Verifiziert, ob bei Nutzung der Annotation `YearRange` nur Jahre von `500` bis `700` generiert werden

4.3.3.3 Implementierung in jqwik

Zur gleichen Zeit wurde die Implementierung der Klasse `DefaultLocalDateArbitrary` vorgenommen. (siehe Anhang F) Die Methode `arbitrary` erzeugt dabei einen Generator, der Objekte des Typs `LocalDate` erzeugt und wendet gleichzeitig die entsprechenden Einschränkungen an. So wird in ihr z. B. ein `Combinator` aus Jahren, Monaten und Tagen gebaut, der anschließend die Methode `generateDateFromValues` mit den erzeugten Werten aufruft.

```
private LocalDate generateDateFromValues(Year y, Month m, int d) {
     LocalDate date;
     date = LocalDate.of(y.getValue(), m, d);
     if (date.isBefore(dateMin)
          || date.isAfter(dateMax)
          || !isInAllowedDayOfWeeks(date.getDayOfWeek())
     ) {
          throw new DateTimeException("Invalid date for the input parameters");
     }
     return date;
}
```

Diese Methode versucht aus den generierten Werten ein Datum zu bauen. Schlägt dies fehl (z. B. 31. Februar) oder liegt das erstellte Datum vor dem Mindest- oder nach dem Maximal-Datum oder ist es an einem nicht erlaubten Wochentag, so wird eine `DateTimeException` geworfen, um mitzuteilen, dass das Datum nicht gültig ist. Ansonsten wird das Datum zurückgegeben. Die `Exception` wird in der `arbitrary` Methode ignoriert und übersprungen, es wird dann also kein Wert zu den möglichen Werten hinzugefügt.

Auch wurden der `YearRangeConfigurator` und `DayOfMonthRangeConfigurator` um jeweils eine weitere innere Klasse `ForLocalDate` erweitert.

4.3.3.4 Design und Implementierung der Default Generation

Auch für `LocalDate` Objekte soll eine Default Generation ermöglicht werden. Dazu wurde ein `LocalDateArbitraryProvider` entworfen (siehe Anhang J) und implementiert. (siehe Anhang F)

Um diesen zu Testen wurde beispielsweise folgender Test hinzugefügt:

`DefaultGenerationTests:validLocalDateIsGenerated:` Überprüft, ob gültige Daten erzeugt werden

4.3.3.5 Design und Implementierung einer MonthRange

Danach ist eine Annotation `MonthRange` entworfen (siehe Anhang H und I) und umgesetzt worden. (siehe Anhang F) So wird es ermöglicht, nur bestimmte Monate generieren zu lassen.

Die Verwendung soll z. B. wie folgt aussehen:

```
@Property
void testeMonate(
     @ForAll @MonthRange(min = Month.MARCH, max = Month.JULY) LocalDate datum
) {
     //mache etwas
}
```

Bei diesem Test würden nur Daten mit den Monaten März bis Juli erzeugt werden.

Zudem wurde beispielsweise der folgende Test hinzugefügt:

`ConstraintTests$Constraints:monthRangeBetweenMarchAndJuly:` Testet, ob nur Monate von März bis Juli erzeugt werden.

Zusätzlich wurde die Implementierung umgesetzt. Hierfür wurde ein `MonthRangeConfigurator` mit entsprechender innerer Klasse erzeugt, der die Einschränkung an das `LocalDateArbitrary` weiterleitet.

4.3.3.6 Design und Implementierung einer DayOfWeekRange

Anschließend ist die Annotation `DayOfWeekRange` geplant (siehe Anhang H und I) und eingefügt worden. (siehe Anhang F) Sie macht es möglich, dass generierte Daten nur bestimmte Wochentage enthalten.

Sie kann beispielsweise auf folgende Weise verwendet werden:

```
@Property
void testeWochentage(
     @ForAll
     @DayOfWeekRange(min = DayOfWeek.TUESDAY, max = DayOfWeek.FRIDAY)
     LocalDate datum
) {
     //mache etwas
}
```

Die hier erzeugten Daten befänden sich jetzt nur von Dienstag bis Freitag.

Um das Ganze zu testen, wurde z. B. folgender Test ergänzt:

`ConstraintTests$Constraints:dayOfWeekRangeOnlyMonday:` Prüft, ob bei Anwendung der Annotation mit entsprechender Einschränkung nur der Wochentag Montag für die Daten erzeugt wird

Zur gleichen Zeit wurde die Implementierung durchgeführt. Es wurde ein `DayOfWeekRangeConfigurator` mit entsprechender innerer Klasse `ForLocalDate` erschaffen, der die getroffenen Einschränkungen an das `LocalDateArbitrary` weitergibt.

4.3.3.7 Design und Implementierung einer DateRange

Zum Schluss ist noch die Annotation `DateRange` designt (siehe Anhang H und I) und implementiert worden. (siehe Anhang F) So wird es möglich, Daten mittels ihres ISO-Strings einzuschränken. Dies sieht z. B. wie folgt aus:

```
@Property
void testeDaten(
     @ForAll @DateRange(min = "2013-05-25", max = "2020-08-23") LocalDate datum
) {
     //mache etwas
}
```

Alle erzeugten Daten würden nun innerhalb des Intervalls vom 25.05.2013 bis zum 23.08.2020 liegen.

Da auch hier die Funktionalität getestet werden sollte, wurde z. B. folgender Test in die Klasse `ConstraintTests` hinzugefügt:

`Constraints$InvalidConfigurations:dateRangeThrowsExceptionIllegalString:` Prüft das Werfen einer Exception bei Verwendung eines nicht erlaubten Strings (hier: „foo“)

Auch hier wurde gleichzeitig die Implementierung umgesetzt. Um dies zu realisieren, wurde ein `DateRangeConfigurator` mit passender innerer Klasse programmiert, der die Einschränkungen an das `LocalDateArbitrary` weitergibt.

4.3.4 Phase 1.4: Calendar

`LocalDate` Objekte können bereits zufällig erzeugt werden. In dieser Phase wurde das Ganze für Objekte des Typs `Calendar` möglich gemacht.

Da diese Phase im Grunde genommen identisch zu der von `LocalDate` ist, wird in dieser Arbeit nur die Implementierung (siehe Anhang F) gezeigt:

Die Methode `arbitrary` hat aktuell nur die Aufgabe einen Generator für den Typen `LocalDate` in einen Generator des Typs `Calendar` umzuwandeln. Später soll die Grundlage `LocalDateTime` sein, denn `Calendar` ist eigentlich ein Typ für Datum mit Zeit. Dies ist aber nicht mehr Teil der Arbeit, weshalb der Zeitanteil entsprechend weggekürzt wurde. Das Umwandeln geschieht mittels der `map` Funktion:

```
@Override
protected Arbitrary<Calendar> arbitrary() {
     return dates.map(DefaultCalendarArbitrary::localDateToCalendar);
}
```

Die zusätzlich erstellte Methode `localDateToCalendar` wandelt dabei den Typen des Datums um.

Im gleichen Zuge wurden auch der `YearRangeConfigurator`, `DayOfMonthRangeConfigurator`, `MonthRangeConfigurator`, `DayOfWeekRangeConfigurator` und `DateRangeConfigurator` jeweils um die innere Klasse `ForCalendar` erweitert.

4.3.5 Phase 1.5: Date

Nun sollen Objekte des Typs `Date` zufällig generiert werden können.

Auch diese Phase ist sehr ähnlich zur vorherigen und so wird nur die Implementierung (siehe Anhang F) dargestellt:

Ähnlich wie im vorherigen Kapitel hat die Methode `arbitrary` lediglich die Aufgabe einen bereits existierenden Generator in einen Generator des Typs `Date` umzuwandeln. Diesmal ist der ursprüngliche Generator aber vom Typ `Calendar`. Auch bei `Date` wird der Zeitanteil in dieser Arbeit weggekürzt und später ergänzt. Das Ganze sieht dann so aus:

```
@Override
protected Arbitrary<Date> arbitrary() {
     return calendars.map(Calendar::getTime);
}
```

Zusätzlich wurden auch hier der `YearRangeConfigurator`, `DayOfMonthRangeConfigurator`, `MonthRangeConfigurator`, `DayOfWeekRangeConfigurator` und `DateRangeConfigurator` jeweils um die innere Klasse `ForDate` erweitert.

4.3.6 Phase 1.6: YearMonth

Als nächstes ist der Generator `YearMonthArbitrary` umgesetzt worden, der eine Kombination aus Jahr und Monat für Jahre größer als `0` erzeugt.

4.3.6.1 Implementierung in jqwik

Die Methode `arbitrary` erzeugt dabei einen Generator für Daten, der jeweils nur den ersten Tag des Monats erzeugen kann. Anschließend werden die erzeugten Werte in Objekte des Typs `YearMonth` umgewandelt. (siehe Anhang F) Die folgende Methode hilft die generierten Daten in ihrem Jahr einzuschränken, indem sie das minimal bzw. maximal benötigte Jahr berechnet:

```
private void setYearMinMax() {
     if (yearMin.getValue() == 1900
          && !yearMonthMin.equals(YearMonth.of(Year.MIN_VALUE, Month.JANUARY))
     ) {
          yearMin = Year.of(yearMonthMin.getYear());
     }
     if (yearMax.getValue() == 2500
          && !yearMonthMax.equals(YearMonth.of(Year.MAX_VALUE, Month.DECEMBER))
     ) {
          yearMax = Year.of(yearMonthMax.getYear());
     }
}
```

Dabei wird geprüft, ob sich das minimale/maximale Jahr vom Standardwert nicht unterscheidet, also vom Nutzer nicht verändert wurde. Ist das der Fall und gleichzeitig unterscheidet sich das Jahr von dem im minimal/maximal möglichen Wert, so wir das minimale/maximale Jahr entsprechend angepasst.

Dazu wurden der `YearRangeConfigurator` und `MonthRangeConfigurator` jeweils um die innere Klasse `ForYearMonth` erweitert.

4.3.6.2 Design und Implementierung einer YearMonthRange

Anschließend wurde die Annotation `YearMonthRange` entworfen (siehe Anhang H und I) und umgesetzt. (siehe Anhang F) Dadurch wird es möglich, die erzeugten Werte per Annotation einzuschränken.

Die Annotation soll wie folgt verwendet werden können:

```
@Property
void testeJahrUndMonat(
     @ForAll @YearMonthRange(min = "2013-05", max = "2020-08") YearMonth jahrMonat
) {
     //mache etwas
}
```

Hier würden nur Objekte von Mai 2013 bis August 2020 erzeugt werden. Der String wird im ISO-Format angegeben.

4.3.7 Phase 1.7: MonthDay

Zusätzlich soll es möglich sein, `MonthDay` Objekte zu generieren. Dafür wurde der Generator `MonthDayArbitrary` erstellt.

4.3.7.1 Implementierung in jqwik

Auch hier erstellt die Methode `arbitrary` einen Generator für Daten, die anschließend in den Typ `MonthDay` umgewandelt werden (siehe Anhang F):

```
@Override
protected Arbitrary<MonthDay> arbitrary() {
     DateArbitrary dates = Dates.dates()
                               .atTheEarliest(LocalDate.of(
                                    2020,
                                    monthDayMin.getMonth(),
                                    monthDayMin.getDayOfMonth()
                               ))
                               .atTheLatest(LocalDate.of(
                                    2020,
                                    monthDayMax.getMonth(),
                                    monthDayMax.getDayOfMonth()
                               ))
                               .yearBetween(2020, 2020)
                               .monthBetween(monthMin, monthMax)
                               .onlyMonths(allowedMonths)
                               .dayOfMonthBetween(dayOfMonthMin, dayOfMonthMax);
     Arbitrary<MonthDay> monthDays =
                    dates.map(v -> MonthDay.of(v.getMonth(), v.getDayOfMonth()));
     monthDays = addAllNeededEndOfMonthEdgeCases(monthDays);
     return monthDays;
}
```

Dabei wird ein Generator für Daten erzeugt, der nur Daten zwischen dem minimalen und maximalen `MonthDay` im Jahr 2020 erzeugt. Das verwendete Jahr muss ein Schaltjahr sein, um auch den 29. Februar generieren zu können. Anschließend werden die in den restlichen Einschränkungsmethoden verwendeten Parameter an den Generator weitergegeben. Danach wird der Generator für Daten zu einem Generator für `MonthDays` umgewandelt. Schließlich werden noch die zusätzlich benötigten Randfälle ergänzt und der neue Generator zurückgegeben.

Nebenbei wurden der `DayOfMonthRangeConfigurator` und `MonthRangeConfigurator` jeweils um eine innere Klasse `ForMonthDay` ergänzt.

4.3.7.2 Design und Implementierung einer MonthDayRange

Danach ist die MonthDayRange Annotation designt (siehe Anhang H und I) und implementiert worden. (siehe Anhang F) So wird es den Nutzern ermöglicht, die generierten Werte mittels einer Annotation einzuschränken.

Sie soll auf die folgende Weise verwendet werden können:

```
@Property
void testeMonatUndTag(
     @ForAll @MonthDayRange(min = "--05-25", max = "--08-23") MonthDay monatTag
) {
     //mache etwas
}
```

Dieser Test würde nur Werte, die sich vom 25. Mai bis zum 23. August befinden, generieren. Der String wird dabei im ISO-Format angegeben.

4.3.8 Phase 1.8: PeriodArbitrary

Als letztes soll es noch möglich sein, Objekte des Typs `Period` automatisch erstellen zu lassen. Hierfür wurde ein Generator `PeriodArbitrary` erzeugt.

4.3.8.1 Entwicklung der Architektur und des Designs

Dazu wurde die Klasse `Dates` um eine weitere Methode `periods` erweitert. Es soll ein Objekt des Typs `PeriodArbitrary` zurückgegeben werden. Hierfür wurde ein `DefaultPeriodArbitrary` erstellt, welches die Methoden des Interfaces `PeriodArbitrary` realisiert. (siehe Anhang G)

Man soll anschließend auf folgende Art und Weise `Periods` generieren können:

```
@Provide
Arbitrary<Period> gueltigePerioden() {
     return Dates.periods();
}
```

Die Methode `between` soll die Werte zudem entsprechend einschränken können.

4.3.8.2 Erweiterung des Designs mit entsprechenden Tests

Um auch hier die Funktionsweise des Generators zu prüfen, wurde das Package `dates.period` angelegt. (siehe Anhang F) Dieses enthält beispielsweise folgenden Test:

```
@Property
void periodCanBePositiveAndNegative(@ForAll Period period) {
     Statistics.label("Period is negative")
                .collect(period.isNegative())
                .coverage(coverage -> {
                     coverage.check(true).percentage(p -> p >= 40);
                     coverage.check(false).percentage(p -> p >= 40);
                });
}
```

Dieser Test legt eine Statistik an, ob die Periode negativ ist, und sammelt diese Daten. Anschließend wird geprüft, ob jeweils zu mindestens 40 Prozent nicht negative sowie negative Perioden auftreten. Ist dies nicht der Fall, schlägt der Test fehl.

Weitere Tests sind z. B.:

`SimpleArbitrariesTests:validPeriodIsGenerated:` Kontrolliert die Gültigkeit der erzeugten Perioden

`EdgeCasesTests:defaultEdgeCases:` Prüft die korrekte Erzeugung von Randfällen

`ExhaustiveGenerationTests:between:` Testet, ob die erwarteten Werte zwischen zwei Perioden auch tatsächlich generiert werden

`PeriodMethodsTests:between:` Überprüft, ob die `between` Methode funktioniert

`ShrinkingTests:defaultShrinking:` Verifiziert das korrekte Schrumpfen der Perioden

4.3.8.3 Implementierung in jqwik

Zur gleichen Zeit wurde die Implementierung der Klasse `DefaultPeriodArbitrary` umgesetzt. (siehe Anhang F) Die Methode `arbitrary` erzeugt dabei einen Generator für Zahlen, welche anschließend in eine Periode umgewandelt werden.

4.3.8.4 Design und Implementierung der Default Generation

Es wurde ein `PeriodArbitraryProvider` designt (siehe Anhang J) und umgesetzt (siehe Anhang F), um die Default Generation für Perioden zu ermöglichen. Dieser sieht wie folgt aus:

```
public class PeriodArbitraryProvider implements ArbitraryProvider {
     @Override
     public boolean canProvideFor(TypeUsage targetType) {
          return targetType.isAssignableFrom(Period.class);
     }
     @Override
     public Set<Arbitrary<?>> provideFor(
          TypeUsage targetType,
          SubtypeProvider subtypeProvider
     ) {
          return Collections.singleton(Dates.periods());
     }
}
```

Die erste Methode prüft dabei, ob auf den angegebenen Typen die Default Generation angewendet werden kann und gibt `true` zurück, wenn das Objekt den Typ `Period` hat. Währenddessen stellt die zweite Methode den Generator für Objekten des Typs `Period` zur Verfügung, falls die erste Methode `true` zurückgegeben hat.

Anschließend wurde z. B. folgender Test hinzugefügt:

`DefaultGenerationTests:validPeriodIsGenerated:` Testet, ob die Default Generation funktioniert

4.3.8.5 Design und Implementierung einer PeriodRange

Zum Schluss der ersten Phase ist eine `PeriodRange` entworfen (siehe Anhang H und I) und implementiert worden. (siehe Anhang F) So können die erzeugten Perioden per Annotation eingeschränkt werden.

Die Verwendung sieht beispielsweise wie folgt aus:

```
@Property
void testePerioden(
     @ForAll @PeriodRange(min = "P1Y2M", max = "P1Y5M3D") Period periode
) {
     //mache etwas
}
```

Hier würden nur Perioden innerhalb von einem Jahr und zwei Monaten sowie einem Jahr, fünf Monaten und 3 Tagen erzeugt werden. Der String wird auch hier im ISO-Format angegeben.

Die Funktionsweise wird z. B. mittels folgenden Tests getestet:

`ConstraintTests$Constraints:nonIsoPeriodThrowsException:` Prüft, ob das Verwenden eines nicht erlaubten Strings eine Exception wirft

Zudem wurde gleichzeitig die Implementierung eines `PeriodRangeConfigurator` umgesetzt. Dieser gibt die vom Nutzer festgelegten Einschränkungen an das `PeriodArbitrary` weiter.

4.4 Phase 2: Zeiten

In diesem Kapitel werden Generatoren für Zeiten und zeitbasierte Klassen behandelt. Aufgrund der Ähnlichkeit der Tests mit bereits im vorherigen Kapitel gezeigten Tests, werden Tests nicht immer erwähnt und nur einige wenige Ausgewählte dargestellt. Dies gilt für alle nachfolgenden Kapitel.

Die folgenden Werte sollen – wenn verfügbar – durch die einzelnen Generatoren eingeschränkt werden können: die Uhrzeit, die Stunden, die Minuten, die Sekunden, das Offset, die Zeitdauer, sowie die Genauigkeit der erzeugten Werte.

4.4.1 Phase 2.1: LocalTime

Zunächst wurde der Generator `LocalTimeArbitrary` umgesetzt. Dieser soll am Ende gültige Uhrzeiten ohne Spezifikation der Zeitzone erzeugen können.

4.4.1.1 Entwicklung der Architektur und des Designs

Als Ergebnis gemeinsamer Designüberlegungen wurde eine neu zu erstellende Klasse `Times` mit einer Methode `times` hinzugefügt. Diese gibt ein Objekt des Typs `LocalTimeArbitrary` zurück. Dazu wird eine Klasse `DefaultLocalTimeArbitrary` benötigt, um die Implementierung des Interfaces zu übernehmen. (siehe Anhang K)

Zeiten können nun auf folgende Weise generiert werden:

```
@Provide
Arbitrary<LocalTime> gueltigeZeiten() {
     return Times.times();
}
```

Die Methoden `between`, `atTheEarliest`, `atTheLatest`, `hourBetween`, `minuteBetween`, `secondBetween` und `ofPrecision` sollen die generierten Zeiten zusätzlich einschränken können.

4.4.1.2 Erweiterung des Designs mit entsprechenden Tests

Die Tests in dem neuangelegten Package `times.localTime` sollen die funktionellen Eigenschaften mit Hilfe von PBT prüfen. (siehe Anhang F)

Es wurde z. B. der folgende Test erstellt:

```
@Property
void validLocalTimeIsGenerated(@ForAll("times") LocalTime time) {
     assertThat(time).isNotNull();
}
@Provide
Arbitrary<LocalTime> times() {
     return Times.times();
}
```

Es wird dabei getestet, ob nur gültige Zeiten erstellt werden.

4.4.1.3 Implementierung in jqwik

Parallel wurde die Klasse `DefaultLocalTimeArbitrary` implementiert. (siehe Anhang F) Die Methode `arbitrary` erzeugt einen Generator für Ganzzahlen, aus welchen anschließend die passende Uhrzeit berechnet wird. Die folgende Methode sorgt dafür, dass der Präzisionswert, welcher die Genauigkeit der erzeugten Werte festlegt, automatisch angepasst wird:

```
private void setOfPrecisionImplicitly(
     DefaultLocalTimeArbitrary clone,
     LocalTime time
) {
     if (clone.ofPrecision.isSet()) {
          return;
     }
     ChronoUnit ofPrecision = ofPrecisionFromTime(time);
     if (clone.ofPrecision.isGreatherThan(ofPrecision)) {
          clone.ofPrecision.setProgrammatically(ofPrecision);
     }
}
```

Diese Funktion wird nur vollständig ausgeführt, wenn die Präzision nicht explizit vom Nutzer festgelegt wurde. Sollte die Präzision von der übergebenen Zeit genauer sein als die bisherige Präzision (also z. B. Nanosekunden statt Millisekunden), so wird die neue Präzision übernommen.

4.4.1.4 Design und Implementierung der Default Generation

Anschließend wurde auch hier die Default Generation designt (siehe Anhang N) und umgesetzt. (siehe Anhang F)

4.4.1.5 Design und Implementierung einer TimeRange

Danach wurde eine `TimeRange` Annotation entworfen (siehe Anhang L und M) und eingefügt. (siehe Anhang F)

Diese soll auf die folgende Weise verwendet werden können:

```
@Property
void testeZeiten(
     @ForAll @TimeRange(min = "01:32:21.113943", max = "03:49:32") LocalTime zeit
){
     //mache etwas
}
```

Hierbei würden jetzt nur Uhrzeiten von 01:32:21.113943 bis 03:49:32 Uhr generiert werden. Die automatische Anpassung der Präzision würde den Wert Microsekunden auswählen.

4.4.1.6 Design und Implementierung einer HourRange

Nun wurde eine `HourRange` designt (siehe Anhang L und M) und implementiert. (siehe Anhang F)

Die Verwendung soll wie folgt möglich sein:

```
@Property
void testeZeiten(@ForAll @HourRange(min = 11, max = 13) LocalTime zeit) {
     assertThat(time.getHour()).isBetween(11, 13);
}
```

Bei Verwendung dieser Annotation würden hier nur Stundenwerte von 11 bis 13 erzeugt werden.

4.4.1.7 Design und Implementierung einer MinuteRange

Analog zum vorherigen Kapitel wurde nun eine `MinuteRange` umgesetzt.

4.4.1.8 Design und Implementierung einer SecondRange

Ebenso wurde eine `SecondRange` hinzugefügt.

4.4.1.9 Design und Implementierung von Precision

Zum Schluss wurde eine `Precision` Annotation konzipiert (siehe Anhang L und M) und programmiert. (siehe Anhang F)

Sie soll auf folgende Weise verwendet werden können:

```
@Property
void testeZeiten(@ForAll @Precision(ChronoUnit.MILLIS) LocalTime zeit) {
     //mache etwas
}
```

Die generierten Zeiten hätten nun die Genauigkeit Millisekunden.

4.4.2 Phase 2.2: ZoneOffset

Anschließend ist der `ZoneOffsetArbitrary` Generator hinzugefügt worden. Dieser soll `ZoneOffsets` von `+14:00:00` bis `-12:00:00` in 15 Minuten Intervallen generieren können.

4.4.2.1 Entwicklung der Architektur und des Designs

Die Klasse `Times` wurde um eine weitere Methode `zoneOffsets` erweitert. Es soll dabei ein Generator des Typs `ZoneOffsetArbitrary` zurückgegeben werden. Um die Programmierung dieses Interfaces zu realisieren, wird zusätzlich eine Klasse `DefaultZoneOffsetArbitrary` benötigt. (siehe Anhang K)

Die Generierung von `ZoneOffsets` soll wie folgt möglich gemacht werden:

```
@Provide
Arbitrary<ZoneOffset> gueltigeZoneOffsets() {
     return Times.zoneOffsets();
}
```

Mittels der Methode `between` sollen die generierten Werte zudem eingeschränkt werden können.

4.4.2.2 Erweiterung des Designs mit entsprechenden Tests

Um die funktionellen Eigenschaften mittels PBT zu überprüfen, wurde in das Package `time` ein weiteres Package `zoneOffset` hinzugefügt. (siehe Anhang F)

Unter anderem wurde folgender Test ergänzt:

```
@Property
void betweenSame(@ForAll("offsets") ZoneOffset sameOffset, @ForAll Random random) {
     Arbitrary<ZoneOffset> offsets = Times.zoneOffsets()
                                          .between(sameOffset, sameOffset);
     assertAllGenerated(offsets.generator(1000), random, offset -> {
          assertThat(offset).isEqualTo(sameOffset);
          return true;
     });
}
@Provide
Arbitrary<ZoneOffset> offsets() {
     return Times.zoneOffsets();
}
```

Dabei wird ein zufälliger `ZoneOffset` Wert generiert. Dieser wird anschließend als obere und untere Grenze für den `ZoneOffset`-Generator verwendet. Anschließend wird getestet, ob alle von diesem Generator erzeugten Werte mit dem zufällig generierten Wert identisch sind.

4.4.2.3 Implementierung in jqwik

Gleichzeitig ist die Klasse `DefaultZoneOffsetArbitrary` programmiert worden. (siehe Anhang F) Die `arbitrary` Methode erzeugt dabei Ganzzahlen, welche einem Index für alle möglichen

Offset-Werte entsprechen, die anschließend in einen `ZoneOffset` Wert umgerechnet werden. Das Umrechnen funktioniert mit der folgenden Methode:

```
static private ZoneOffset offsetFromValue(int index) {
     int hour = index / 4;
     index -= hour * 4;
     int minute = index * 15;
     return ZoneOffset.ofHoursMinutes(hour, minute);
}
```

Dabei wird aus einem generierten Indexwert zunächst der Stundenwert abgeleitet und anschließend der Minutenwert. So würde aus der Zahl `9` z. B. der Wert `2` Stunden und `15` Minuten werden.

4.4.2.4 Design und Implementierung der Default Generation

Im Anschluss ist die Default Generation entworfen (siehe Anhang N) und hinzugefügt worden. (siehe Anhang F)

4.4.2.5 Design und Implementierung einer OffsetRange

Anschließend ist eine `OffsetRange` designt (siehe Anhang L und M) und implementiert worden. (siehe Anhang F)

Das Verwenden dieser Annotation soll auf folgende Art und Weise möglich sein:

```
@Property
void testeZoneOffsets(@ForAll @OffsetRange(min = "-09:00:00") ZoneOffset offset) {
     //mache etwas
}
```

Hierbei würden nur `ZoneOffsets` generiert werden, die größer gleich `-09:00:00` sind.

4.4.3 Phase 2.3: Zeitzonen und Zonen-IDs

Nun sind die Generatoren für Objekte des Typs `TimeZone` und `ZoneId` hinzugefügt worden. Es soll für diese Klassen aufgrund der Einfachheit kein eigenes Interface ergänzt werden.

4.4.3.1 Entwicklung der Architektur und des Designs

Die Klasse `Times` wurde um die Methoden `timeZones` und `zoneIds` erweitert. Zurückgegeben werden Generatoren des Typs `Arbitrary<TimeZone>` bzw. `Arbitrary<ZoneId>`. (siehe Anhang K)

Die Verwendung der Generatoren soll wie folgt möglich gemacht werden:

```
@Provide
Arbitrary<TimeZone> gueltigeZeitzonen() {
     return Times.timeZones();
}
```

```
@Provide
Arbitrary<ZoneId> gueltigeZonenIDs() {
     return Times.zoneIds();
}
```

Die generierten Werte können nicht direkt eingeschränkt werden, jedoch kann der Nutzer dies mit Hilfe von geeigneten Filtern umsetzen.

4.4.3.2 Erweiterung des Designs mit entsprechenden Tests

Da die Generatoren einfach gehalten sind, ist jeweils nur ein Test nötig. Dieser wird in dem bereits erstellten Package `times.localTime` untergebracht. (siehe Anhang F)

Der folgende Test wird der Klasse `SimpleArbitrariesTests` hinzugefügt:

```
@Property
void validTimeZoneIsGenerated(@ForAll("timeZones") TimeZone timeZone) {
     assertThat(timeZone).isNotNull();
}
@Provide
Arbitrary<TimeZone> timeZones() {
     return Times.timeZones();
}
```

Dabei werden alle generierten Zeitzonen auf Gültigkeit überprüft. Der ähnlich aufgebaute Test `validZoneIdIsGenerated` prüft das Ganze für Zone-IDs.

4.4.3.3 Implementierung in jqwik

Die Implementierungen wurden nun durchgeführt. (siehe Anhang F)

```
public static Arbitrary<TimeZone> timeZones() {
     return Arbitraries.of(TimeZone.getAvailableIDs()).map(TimeZone::getTimeZone);
}
public static Arbitrary<ZoneId> zoneIds() {
     return Arbitraries.of(ZoneId.getAvailableZoneIds()).map(ZoneId::of);
}
```

Hier wird wieder auf die `Arbitaries.of` Methode zurückgegriffen, um die Werte zufällig zu erzeugen. Da jedoch nur die jeweiligen IDs generiert werden, müssen diese mittels der `map` Methode noch in die jeweils benötigten Typen umgewandelt werden.

4.4.4 Phase 2.4: OffsetTime

In der anschließenden Phase wurde das `OffsetTimeArbitrary` ergänzt.

4.4.4.1 Entwicklung der Architektur und des Designs

Nun ist die Klasse `Times` um die weitere Methode `offsetTimes` erweitert worden. Das Rückgabeobjekt dieser Funktion hat den Typ `OffsetTimeArbitrary`. Es wird zusätzlich eine Klasse

`DefaultOffsetTimeArbitrary` benötigt, die das Interface `OffsetTimeArbitrary` realisiert. (siehe Anhang K)

Auf die folgende Art soll die Generierung von Zeiten mit einem Offset möglich gemacht werden:

```
@Provide
Arbitrary<OffsetTime> gueltigeZeitenMitOffset() {
     return Times.offsetTimes();
}
```

Die Methoden `between`, `atTheEarliest`, `atTheLatest`, `hourBetween`, `minuteBetween`, `secondBetween`, `offsetBetween` und `ofPrecision` sollen es zusätzlich möglich machen, die generierten Werte einzuschränken.

4.4.4.2 Erweiterung des Designs mit entsprechenden Tests

Das Package `times` wurde um ein weiteres Package `offsetTime` ergänzt, um die funktionalen Eigenschaften sicherzustellen. (siehe Anhang F)

4.4.4.3 Implementierung in jqwik

Nebenbei wurde die Klasse `DefaultOffsetTimeArbitrary` implementiert. (siehe Anhang F)

```
@Override
protected Arbitrary<OffsetTime> arbitrary() {
     return Combinators.combine(localTimes, zoneOffsets).as(OffsetTime::of);
}
```

Die Methode `arbitrary` kombiniert zwei Generatoren für die Typen `LocalTime` und `ZoneOffset` und baut daraus einen Generator für `OffsetTime`, der zurückgegeben wird.

Gleichzeitig wurden die `TimeRange`, `HourRange`, `MinuteRange`, `SecondRange`, `OffsetRange` und `Precision` Annotationen für `OffsetTime` verfügbar gemacht.

4.4.4.4 Design und Implementierung der Default Generation

Anschließend ist die Default Generation entworfen (siehe Anhang N) und implementiert (siehe Anhang F) worden.

Der folgende Test stellt sicher, ob alle erzeugten Werte gültig sind:

```
@Property
void validOffsetTimeIsGenerated(@ForAll OffsetTime time) {
     assertThat(time).isNotNull();
}
```

4.4.5 Phase 2.5: Duration

Als letzter Schritt in der zweiten Phase ist ein `DurationArbitrary` Generator umgesetzt worden.

4.4.5.1 Entwicklung der Architektur und des Designs

Die Klasse `Times` ist nun um eine Methode `durations` ergänzt worden. Zurückgegeben wird ein Objekt des Typs `DurationArbitrary`. Die Klasse `DefaultDurationArbitrary` soll dieses Interface umsetzen. (siehe Anhang K)

Die Verwendung soll wie folgt möglich sein:

```
@Provide
Arbitrary<Duration> gueltigeZeitDauern() {
     return Times.durations();
}
```

Mittels der Methoden `between` und `ofPrecision` sollen die generierten Werte eingeschränkt werden können.

4.4.5.2 Erweiterung des Designs mit entsprechenden Tests

Damit die funktionalen Eigenschaften überprüft werden können, wurde das Package `times` um das Package `duration` erweitert. (siehe Anhang F)

So wurde beispielsweise folgender Test hinzugefügt:

```
@Property
void ofPrecision(@ForAll ChronoUnit chronoUnit) {
     Assume.that(!chronoUnit.equals(NANOS));
     Assume.that(!chronoUnit.equals(MICROS));
     Assume.that(!chronoUnit.equals(MILLIS));
     Assume.that(!chronoUnit.equals(SECONDS));
     Assume.that(!chronoUnit.equals(MINUTES));
     Assume.that(!chronoUnit.equals(HOURS));
     assertThatThrownBy(
          () -> Times.durations().ofPrecision(chronoUnit)
     ).isInstanceOf(IllegalArgumentException.class);
}
```

Hierbei werden zufällige Einheiten erzeugt. Wird eine nicht erlaubte Einheit verwendet, so wird geprüft, ob eine passende `Exception` geworfen wird.

4.4.5.3 Implementierung in jqwik

Zur gleichen Zeit wurde die Klasse `DefaultDurationArbitrary` umgesetzt. (siehe Anhang F) Die `arbitrary` Methode generiert dabei Ganzzahlen, die anschließend in eine `Duration` umgewandelt werden. Die folgende Methode berechnet dabei die maximal mögliche `Duration` anhand der gegebenen Präzision:

```
public Duration maxPossibleDuration() {
     switch (precision) {
          case HOURS:
               return Duration.ofSeconds((Long.MAX_VALUE / (60*60)) * (60*60), 0);
          case MINUTES:
               return Duration.ofSeconds((Long.MAX_VALUE / 60) * 60, 0);
          case MILLIS:
               return Duration.ofSeconds(Long.MAX_VALUE, 999_000_000);
          case MICROS:
               return Duration.ofSeconds(Long.MAX_VALUE, 999_999_000);
          case NANOS:
               return Duration.ofSeconds(Long.MAX_VALUE, 999_999_999);
          default:
               return Duration.ofSeconds(Long.MAX_VALUE, 0);
     }
}
```

Durch die Division und anschließende Multiplikation bei den Stunden bzw. Minuten wird dafür gesorgt, dass die Sekunden bzw. auch die Minuten (bei Stunden) auf den Wert 0 gesetzt werden.

Nebenbei wurde der `PrecisionConfigurator` um die innere Klasse `ForDuration` erweitert.

4.4.5.4 Design und Implementierung der Default Generation

Hiernach wurde die Default Generation entworfen (siehe Anhang N) und implementiert. (siehe Anhang F)

4.4.5.5 Design und Implementierung einer DurationRange

Anschließend ist eine `DurationRange` entworfen (siehe Anhang L und M) und implementiert worden. (siehe Anhang F)

Diese soll wie folgt verwendet werden können:

```
@Property
void testeZeitDauern(
     @ForAll
     @DurationRange(min = "PT-3000H-39M-22.123111444S", max = "PT1999H22M11S")
     Duration duration
) {
     //mache etwas
}
```

Die hier generierten `Durations` würden im Intervall von `-3000H-39M-22.123111444S` bis `1999H22M11S` liegen.

4.5 Phase 3: LocalDateTime

Zum Schluss folgte ein Generator für die Kombination aus Zeit und Datum – einer für die Klasse `LocalDateTime`. Dafür ist der Generator `LocalDateTimeArbitrary` umgesetzt worden.

4.5.1 Entwicklung der Architektur und des Designs

Es ist eine Klasse `DateTimes` mit einer Methode `dateTimes` neu erstellt worden. Dabei wird ein Objekt des Typs `LocalDateTimeArbitrary` zurückgegeben. Hierzu wird eine Klasse `DefaultLocalDateTimeArbitrary` benötigt, die die Implementierung des Interfaces übernimmt. (siehe Anhang O)

Die Generierung von Zeit und Datum soll wie folgt ermöglicht werden:

```
@Provide
Arbitrary<LocalDateTime> gueltigesDatumMitZeit() {
     return DateTimes.dateTimes();
}
```

Dabei sollen die bereits aus dem `LocalDateArbitrary` bzw. `LocalTimeArbitrary` bekannten Methoden die generierten Werte einschränken können.

4.5.2 Erweiterung des Designs mit entsprechenden Tests

Anschließend wurde ein neues Package `dateTimes` mit dem Package `localDateTime` angelegt. (siehe Anhang F) Dieses soll die funktionellen Eigenschaften mit Hilfe von PBT testen.

Dieses Package enthält den folgenden Test:

```
@Property
void between(@ForAll LocalDate min, @ForAll LocalDate max, @ForAll Random random) {
     Assume.that(!min.isAfter(max));
     Arbitrary<LocalDateTime> dateTimes = DateTimes.dateTimes()
                                                   .dateBetween(min, max);
     assertAllGenerated(dateTimes.generator(1000, true), random, dateTime -> {
          assertThat(dateTime.toLocalDate()).isAfterOrEqualTo(min);
          assertThat(dateTime.toLocalDate()).isBeforeOrEqualTo(max);
          return true;
     });
}
```

Dieser Test erhält zwei Daten, die er als Minimum bzw. Maximum in einem erstellten `LocalDateTime` Generator verwendet. Anschließend wird dieser Generator daraufhin überprüft, ob alle generierten Daten zwischen dem Mindest- und Höchstdatum liegen.

4.5.3 Implementierung in jqwik

Gleichzeitig wurde die Klasse `DefaultLocalDateTimeArbitrary` umgesetzt. (siehe Anhang F) Die `arbitrary` Methode erzeugt dabei je einen Generator für Daten und Zeiten, welchem sie alle vom Nutzer getroffenen Einschränkungen übergibt. Anschließend kombiniert sie diese zu einem Generator für den Typ `LocalDateTime`:

```
@Override
protected Arbitrary<LocalDateTime> arbitrary() {
      //mehr Code
      LocalDateArbitrary dates = Dates.dates();
      //mehr Code
      dates = setDateParams(dates);
      //mehr Code
      return dates.flatMap(date -> /*mehr Code*/);
}
```

Die folgende Methode gibt einzelne Einschränkungen an den Generator für Daten weiter:

```
private LocalDateArbitrary setDateParams(LocalDateArbitrary dates) {
      dates = dates.onlyMonths(allowedMonths.get().toArray(new Month[]{}));
      if (dayOfMonthBetween.getMin() != null && dayOfMonthBetween.getMax() != null){
            dates = dates.dayOfMonthBetween(
                        dayOfMonthBetween.getMin(), dayOfMonthBetween.getMax()
                    );
      }
      dates = dates.onlyDaysOfWeek(allowedDayOfWeeks.get()
                                                      .toArray(new DayOfWeek[]{})
                                    );
      return dates;
}
```

So werden die erlaubten Monate und die erlaubten Wochentage übergeben, sowie – wenn Grenzen dafür gesetzt sind – die erlaubten Tageswerte.

Dazu wurden einige bereits vorhandene Konfiguratoren um eine Klasse `ForLocalDateTime` erweitert. (siehe Anhang P)

4.5.4 Design und Implementierung der Default Generation

Danach wurde die Default Generation designt (siehe Anhang O) und hinzugefügt. (siehe Anhang F)

4.5.5 Design und Implementierung einer DateTimeRange

Abschließend wurde noch eine `DateTimeRange` Annotation entworfen (siehe Anhang O und P) und implementiert. (siehe Anhang F)

Sie soll wie folgt verwendet werden können:

```
@Property
void testeDatumMitZeit(
     @ForAll
     @DateTimeRange(
          min = "2013-05-25T01:32:21.113943", max = "2020-08-23T01:32:21.113943"
     )
     LocalDateTime dateTime
) {
     //mache etwas
}
```

So würden jetzt nur Daten vom 25.05.2013 01:32:21.113943 Uhr bis zum 23.08.2020 01:32:21.113943 Uhr erzeugt werden.

4.6 Anwendung in der Praxis

Zu Beginn des vierten Kapitels wurde bereits über den Y2K38-Bug geschrieben. Doch kann dieser mittels der neu eingefügten Module in jqwik entdeckt werden? Dazu wird folgender Test geschrieben:

```
@Property
void y2k38(@ForAll @YearRange(min = 1970) LocalDateTime datum){
     int zeitstempel = (int) datum.toEpochSecond(ZoneOffset.UTC);
     LocalDateTime datumAusStempel = LocalDateTime.ofEpochSecond(
                                             zeitstempel, 0, ZoneOffset.UTC
                                        );
     assertThat(datumAusStempel).isEqualTo(datum);
}
```

Dabei werden zufällige Daten mit Zeit ab dem 1.1.1970 00:00:00 Uhr generiert, also dem Beginn der Unix-Zeit. Anschließend werden diese in einen Unix-Zeitstempel – also in Sekunden seit dem 1.1.1970 – umgewandelt. Aus diesem Stempel wird ein Datum mit Uhrzeit gebaut. Zum Schluss wird verglichen, ob das neugebaute Datum mit dem Ursprungsdatum identisch ist.

Führt man den Test nun aus, schlägt er fehl. Jqwik zeigt dabei die folgende Fehlermeldung an:

```
expected: 2038-01-19T03:14:08 (java.time.LocalDateTime)
but was : 1901-12-13T20:45:52 (java.time.LocalDateTime)
```

Dies ist bereits das geschrumpfte Ergebnis. Wie anfangs erwähnt sieht man hier deutlich, dass der letzte funktionierende Wert der 19.01.2038 03:14:07 Uhr ist. Nur eine Sekunde später springt das Jahr auf 1901.

Mittels eines solchen Tests kann das Problem früh erkannt und so rechtzeitig behoben werden. Andernfalls wird es im Jahr 2038 dazu kommen, dass einige Geräte auf 1901 zurückspringen werden.

4.7 Bewertung gegenüber den Anforderungen

Jqwik wurde nun um einige hilfreiche Funktionen erweitert. Im Großen und Ganzen entsprechen die einzelnen Programmteile auch den zuvor gestellten Anforderungen. So können Nutzer von jqwik in Zukunft sehr einfach Daten und Uhrzeiten erzeugen.

Allerdings ist dies weiterhin noch nicht für alle im JDK vorhandenen Typen für Zeiten bzw. Daten möglich. Dies betrifft vor allem sogenannte „Date-Time-Werte“. Das liegt daran, dass in der Planung der Gesamtaufwand für dieses Modul deutlich unterschätzt wurde und eine Umsetzung den Rahmen dieser Arbeit übertroffen hätte. Diese Entscheidung wurde zusammen mit Johannes Link – dem Initiator von jqwik – getroffen.

Auch sind sowohl während der schriftlichen Ausarbeitung als auch noch während der Entwicklung immer wieder Fehler in den einzelnen Programmteilen gefunden worden. Dies ist anhand der vielen Codezeilen des Projekts allerdings nicht verwunderlich und es ist unwahrscheinlich, dass bereits alle Fehler gefunden wurden.

5 Schlussfolgerung

In dieser Arbeit konnte jqwik um einige neue Funktionen erweitert werden. So können Nutzer in Zukunft automatisch E-Mail-Adressen, Daten und Uhrzeiten generieren lassen. Dazu konnten viele Dinge über das Entwickeln und Testen von Software erlernt und verinnerlicht werden.

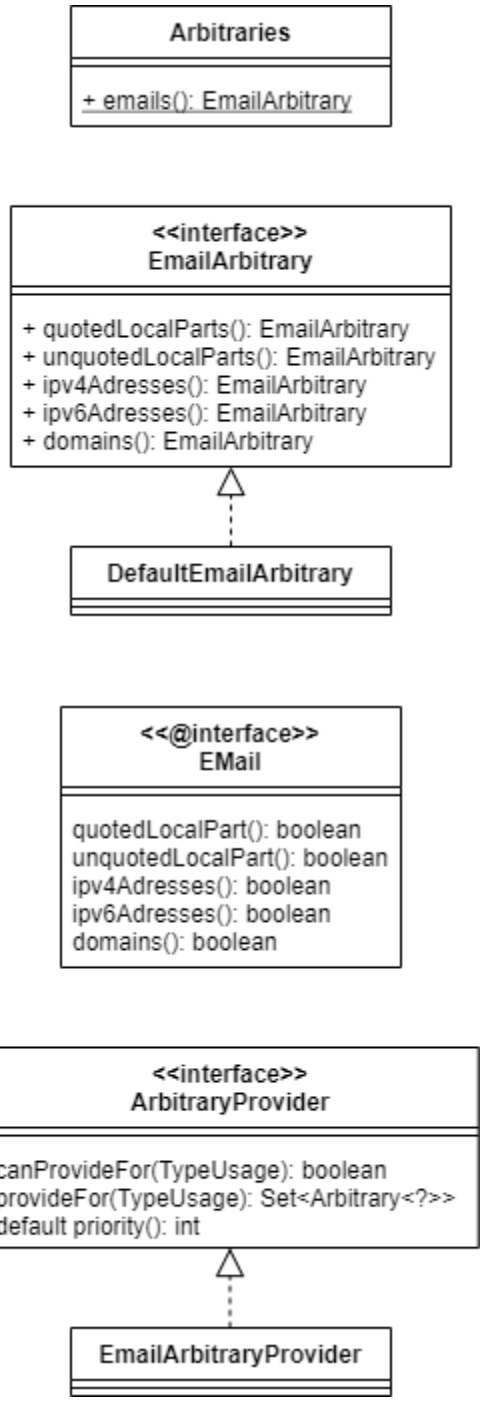
Zu Beginn ist die Einarbeitung in jqwik etwas schwergefallen, da noch keine Erfahrung mit Property-Based Testing vorhanden war. Durch Lesen der jqwik Dokumentation sowie Hilfe von Johannes Link wurde es erleichtert, die Entwicklung durchzuführen. Bei der Erweiterung des Frameworks jqwik sollte Vorwissen in der Programmiersprache Java (oder einer ähnlichen) vorhanden sein, jedoch benötigt man kein Vorwissen zu Property-Based Testing. Zudem sollte man etwas Zeit mitbringen, um sich in jqwik und evtl. PBT einzuarbeiten. Hilfreich ist es auch, zunächst selbst Properties zu schreiben, bevor man mit der Erweiterung beginnt. Sollte weitere Hilfe nötig sein, kann man im zugehörigen Forum von jqwik auf GitHub darum bitten.

Es wurde zudem festgestellt, dass trotz vieler automatisierter Beispieltests und Properties während der Entwicklung bei weitem nicht alle Fehler gefunden werden. Im Umkehrschluss bedeutet das auch, dass selbst eine hohe Testabdeckung kein fehlerfreies Programm garantiert. So bleibt Testen zwar weiterhin ein sehr wichtiger Bestandteil der Entwicklungsphase, jedoch nicht der einzige, um Fehler zu finden. Kleine Fehler werden häufig erst im Laufe der Zeit bei Nutzung oder Weiterentwicklung des Programms gefunden, jedoch nicht immer in den vorherigen Testphasen.

Jqwik bietet nun mehr Datentypen, für welche der Nutzer Property-Based Testing in Java verwenden kann. Allerdings ist dies nur ein Bruchteil der insgesamt im JDK vorhandenen Datentypen, und so muss jqwik auch in Zukunft weiterentwickelt werden. Da jqwik ein Open-Source Projekt ist, kann jeder dazu beitragen und neue Funktionen in einem Pull-Request ergänzen, die dann evtl. zu jqwik hinzugefügt werden. Vielleicht fühlt sich auch der ein oder andere Leser dieser Arbeit angesprochen und dazu motiviert jqwik zu erweitern.

Anhang

Anhang A – Design E-Mail-Adressen Generierung



Anhang B – Quellcode: E-Mail-Adressen Generierung

Der Quellcode sowie der Änderungsverlauf können über die folgenden Pull requests eingesehen werden:

- https://github.com/jlink/jqwik/pull/128
- https://github.com/jlink/jqwik/pull/131
- https://github.com/jlink/jqwik/pull/135

Anhang C – Beispielapplikation 1

Kalender-Planungs-App:
- Nutzer kann Datum und Uhrzeit von Terminen in der Zukunft eintragen
- Anzeige wie lange noch bis zu Termin
- Beispiel: 20.10.2021 14:30:15 Serverwartungsarbeiten

Beispiel-Properties:
- `ForAll LocalDateTime lcd: lcd.after(LocalDateTime.now())`
  Alle Daten müssen in der Zukunft sein, dazu wird eine min-Funktion benötigt.
- `ForAll LocalDateTime lcd, Forall LocalDateTime now:`
  `if(now.before(lcd)): Subtraktionsmethode der Klasse muss lcd-now ergeben`
  `else: Subtraktionsmethode der Klasse muss 0 ergeben`
  Hierfür werden zwei Daten benötigt, die keine Einschränkung haben. Es wird keine weitere Funktion in jqwik benötigt.

Anhang D – Beispielapplikation 2

Reservierungs-App für Restaurant:
- Reservierungen nur zu den Öffnungszeiten, Dienstag und Donnerstag Ruhetag

Beispiel-Properties:
- `ForAll LocalDateTime lcd: lcd.after(LocalDateTime.now())`
  Alle Daten müssen in der Zukunft sein, dazu wird eine min-Funktion benötigt.
- `ForAll LocalDateTime lcd: lcd.getDayOfWeek() is MO/MI/FR/SA/SO`
  Alle Daten müssen an einem bestimmten Wochentag sein, dazu wird eine Einschränkungs-Funktion für Wochentage benötigt.
- `ForAll LocalDateTime lcd: lcd.toLocalTime().isBetween(startTime, endTime)`
  Alle generierten Uhrzeiten müssen während der Öffnungszeit sein, dazu wird eine min/max-Zeit-Funktion benötigt

Anhang E – Sammlung der Methoden

Mindestens die folgenden Methoden sollen im Modul zur Verfügung gestellt werden:

Datumsbezogen:
- Min/Max Datum gesamt
- Min/Max Jahreszahl
- Min/Max Monatswert
- Min/Max Tageszahl
- Nur bestimmte Wochentage
- Perioden

Zeitbezogen:
- Min/Max Uhrzeit
- Min/Max Stundenzahl
- Min/Max Minutenzahl
- Min/Max Sekundenzahl
- Zeitintervalle/Durations

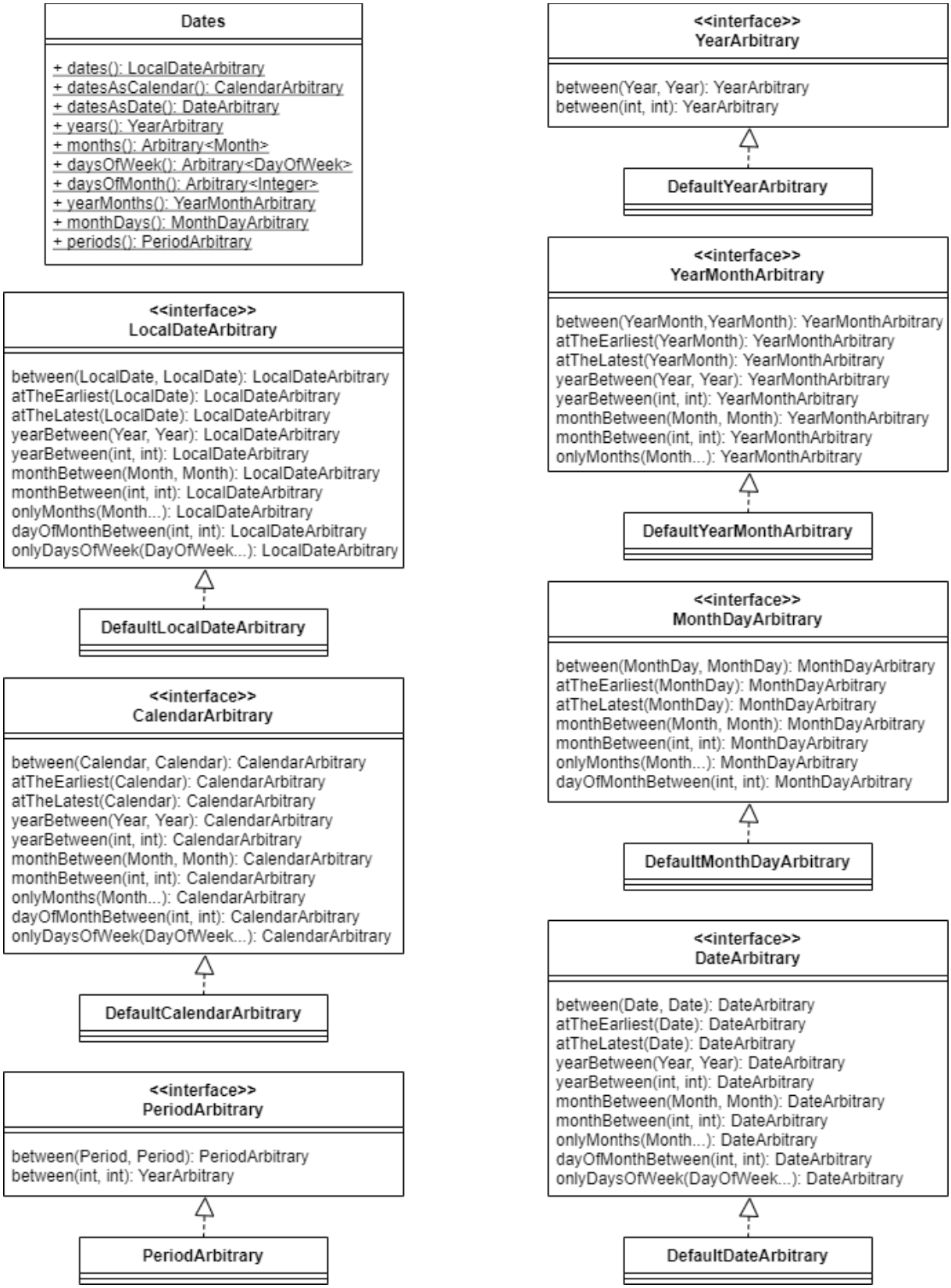
Datums- und Zeitbezogen:
- Kombination der Funktionen aus datums- und zeitbezogen

Anhang F – Quellcode: Generierung von Daten

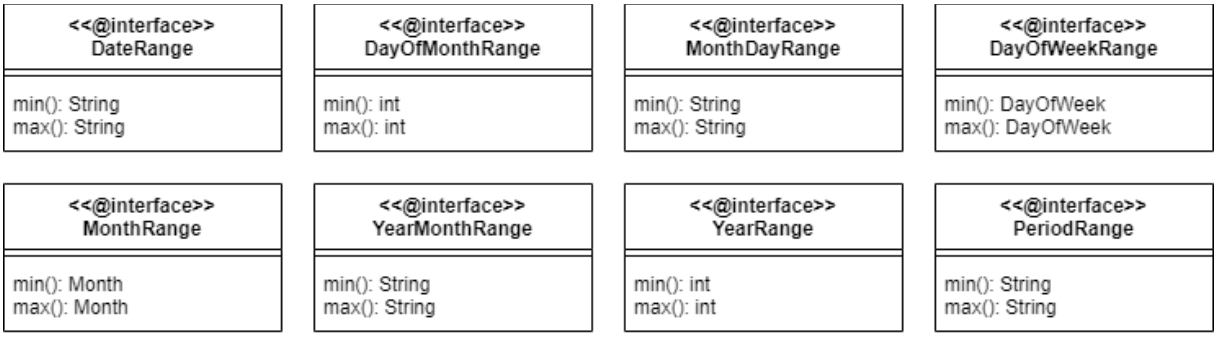
Der Quellcode sowie der Änderungsverlauf können über die folgenden Pull requests eingesehen werden:

- https://github.com/jlink/jqwik/pull/141
- https://github.com/jlink/jqwik/pull/147
- https://github.com/jlink/jqwik/pull/155
- https://github.com/jlink/jqwik/pull/156
- https://github.com/jlink/jqwik/pull/176
- https://github.com/jlink/jqwik/pull/178
- https://github.com/jlink/jqwik/pull/183
- https://github.com/jlink/jqwik/pull/184

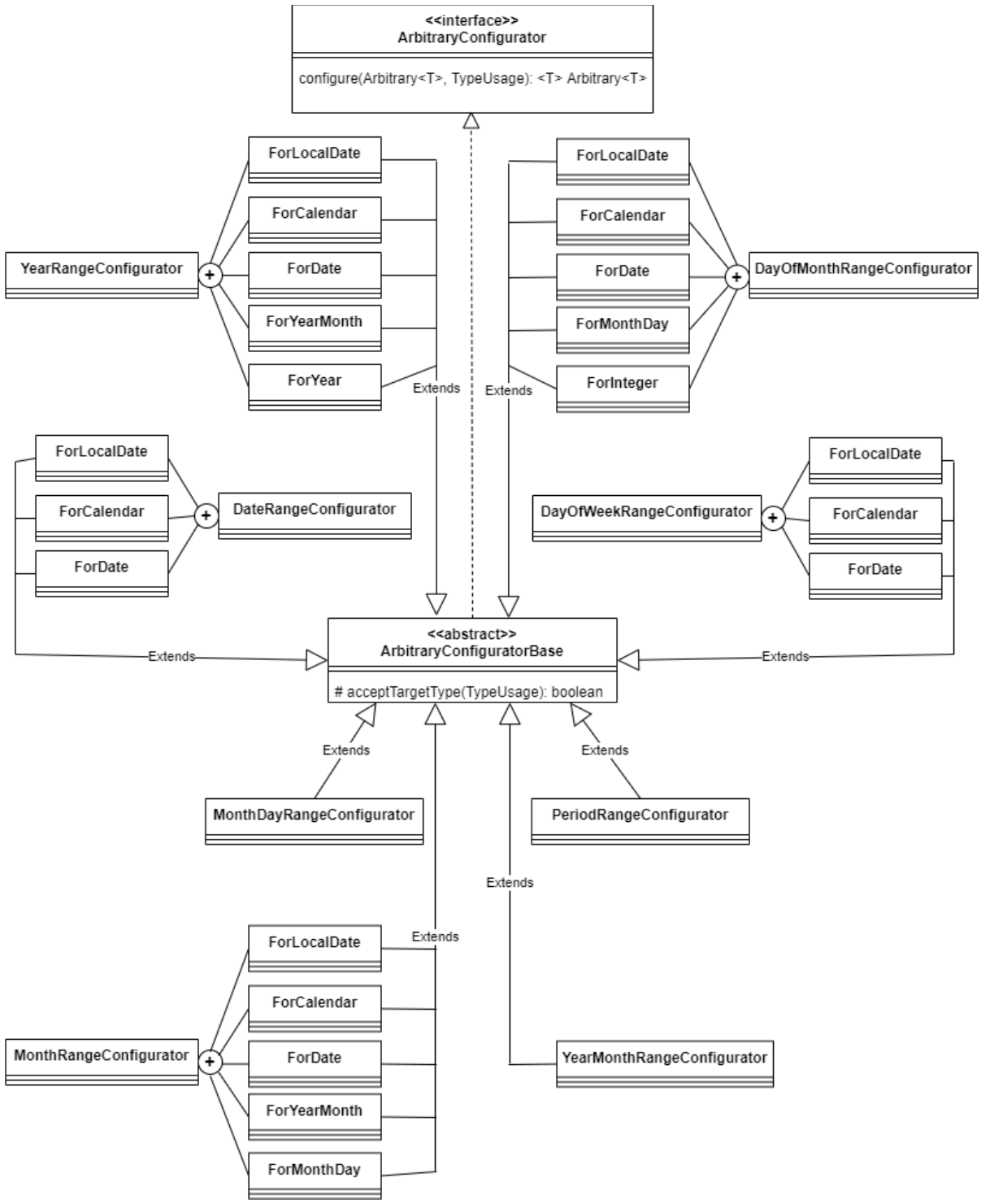
Anhang G – Design Daten-Generierung 1



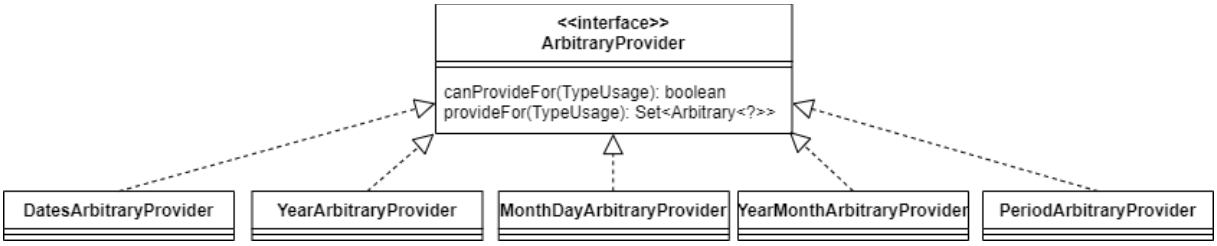
Anhang H – Design Daten-Generierung 2



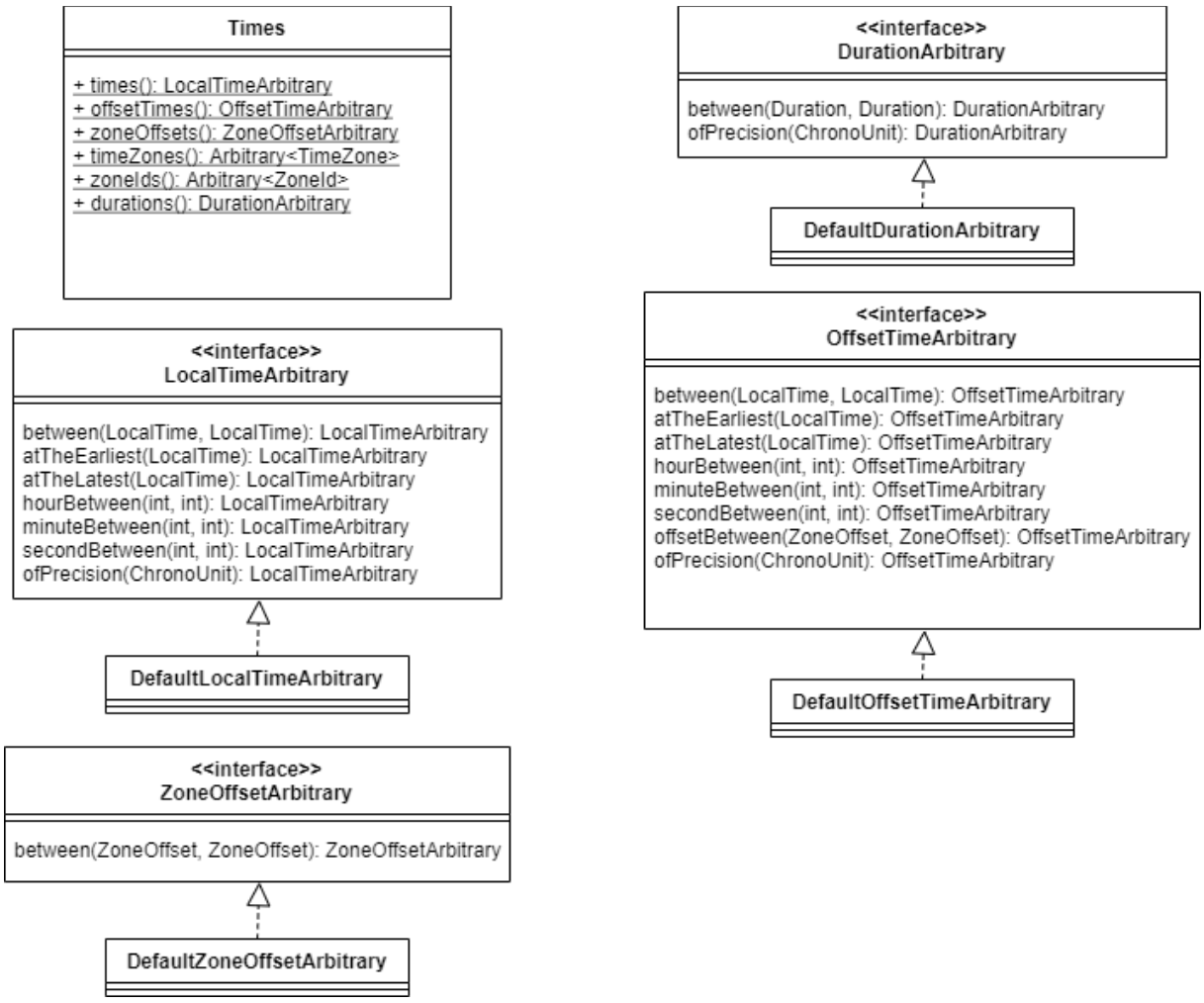
Anhang I – Design Daten-Generierung 3



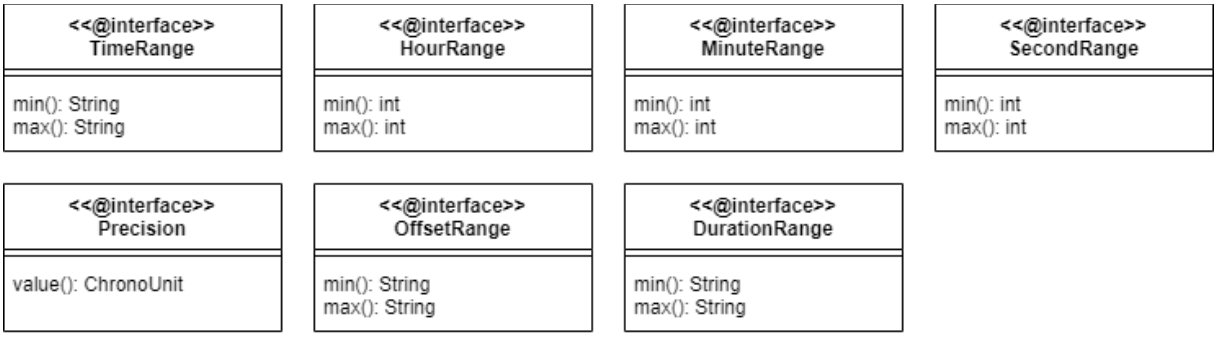
Anhang J – Design Daten-Generierung 4



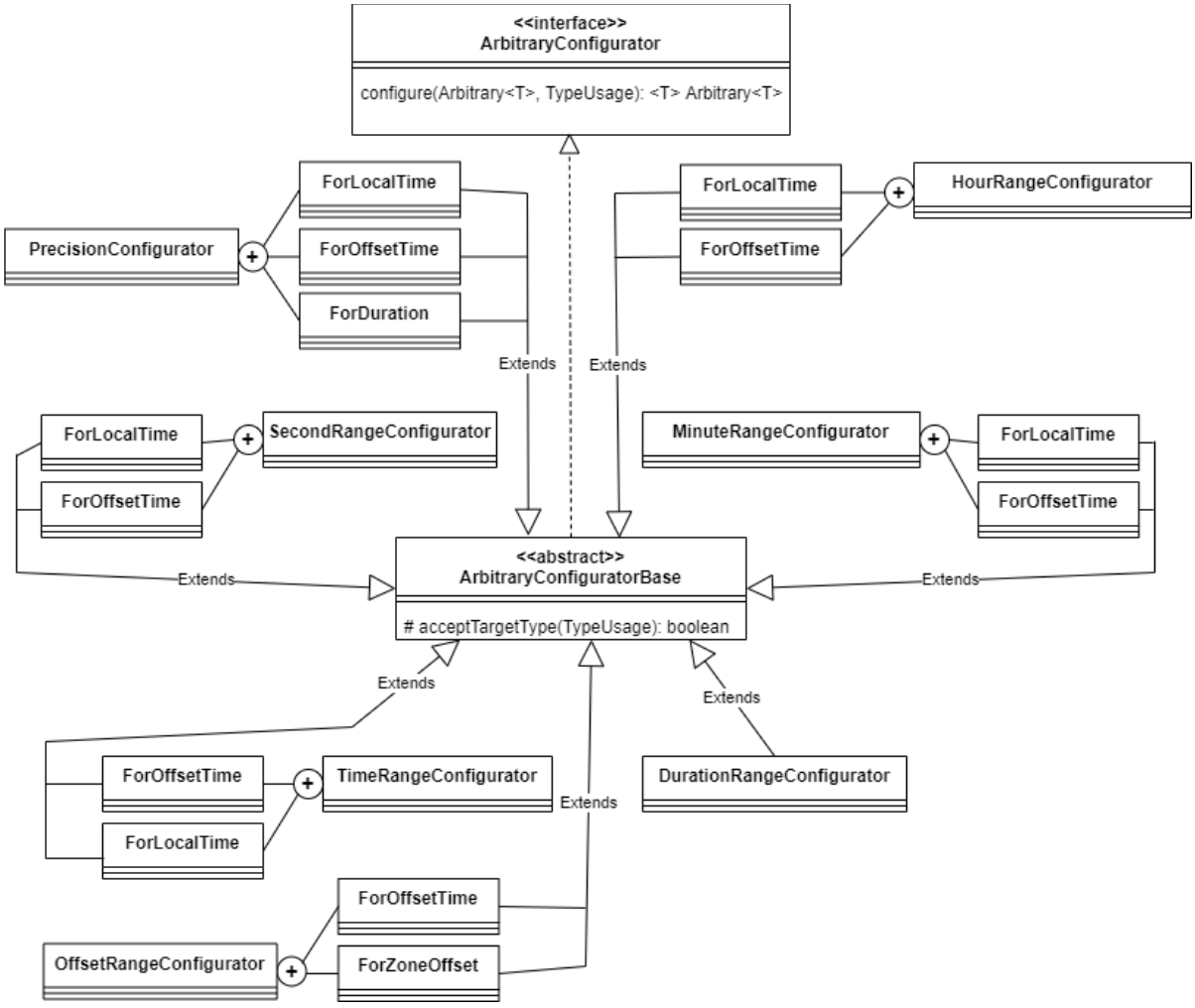
Anhang K – Design Zeiten-Generierung 1



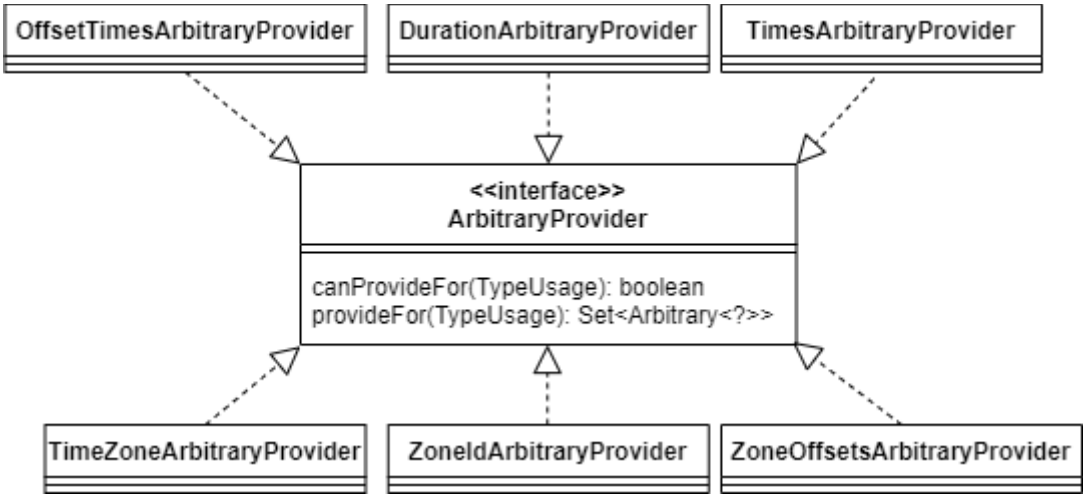
Anhang L – Design Zeiten-Generierung 2



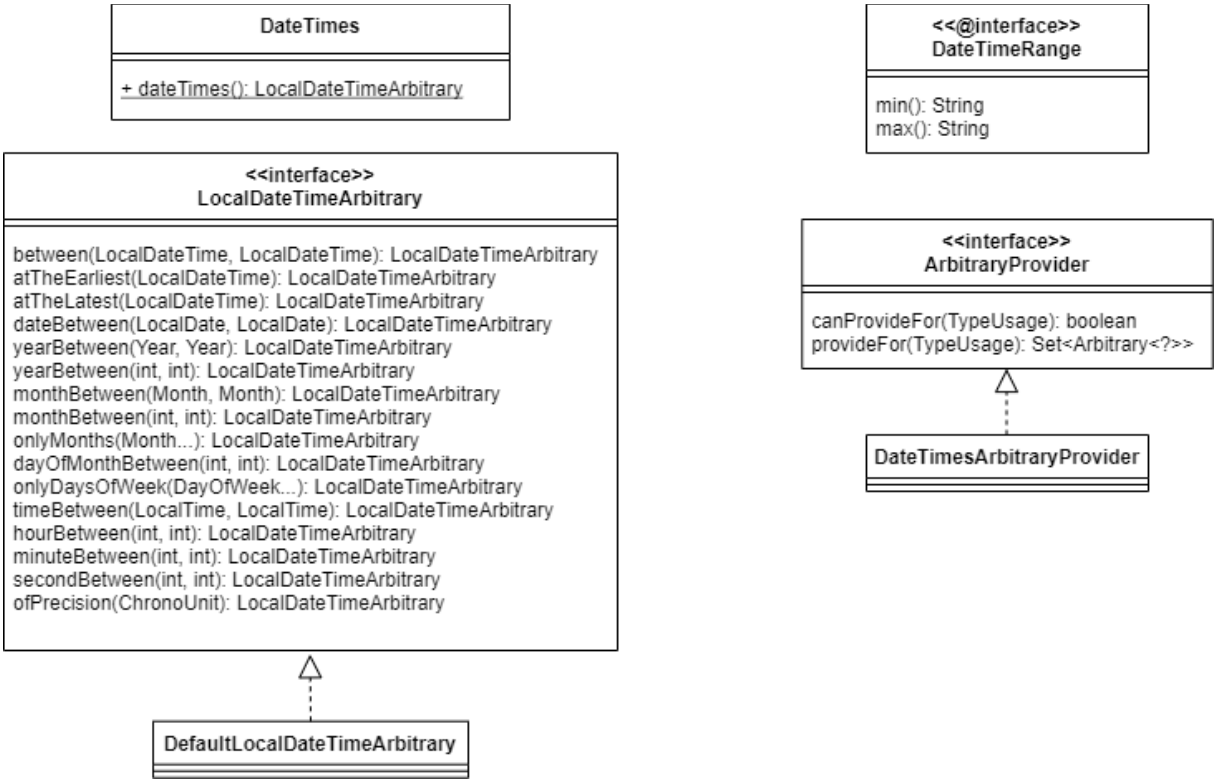
Anhang M – Design Zeiten-Generierung 3



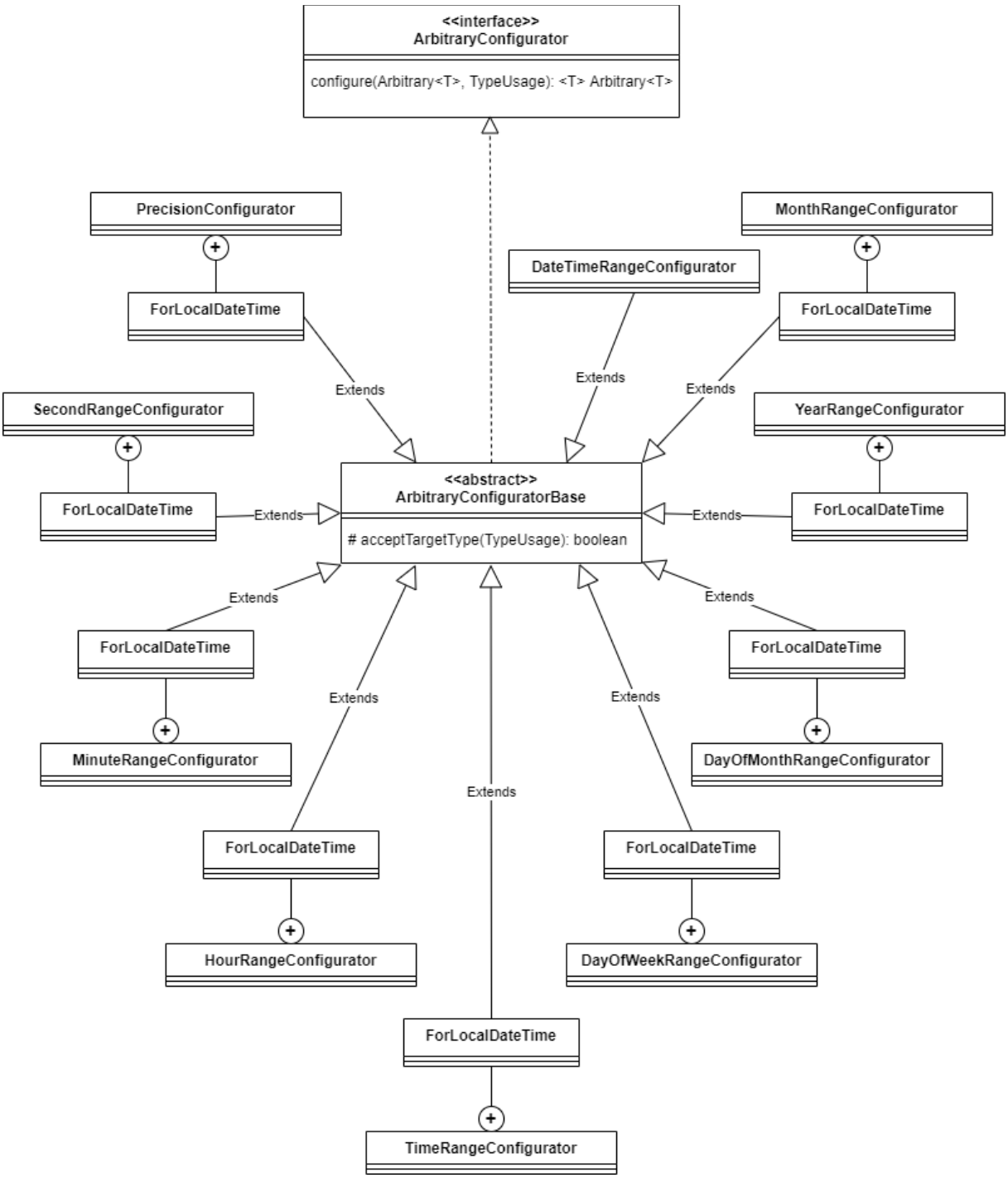
Anhang N – Design Zeiten-Generierung 4



Anhang O – Design Datums und Zeit-Generierung 1



Anhang P – Design Datums und Zeit-Generierung 2



Literaturverzeichnis

Arab, I., Bourhnane, S. & Kafou, F. (2018). Unifying Modeling Language-Merise Integration Approach for Software Design. *International Journal of Advanced Computer Science and Applications, 9(4).* https://mediatum.ub.tum.de/doc/1510115/file.pdf

Aubert, A. (2019, 19. Dezember). Systemtests: ein objektiver Blick auf das Produkt. *AT INTERNET.* https://blog.atinternet.com/de/systemtests-ein-objektiver-blick-auf-das-produkt/

Avci, O. & Wagner, H. (2005). *Das chronische Problem der Anforderungsanalyse und die Frage: Fehler vermeiden oder früh entdecken?* Informatik 2005 – Informatik Live! Band 2. https://dl.gi.de/handle/20.500.12116/28214

baeldung. (2020, 29. September). Java Service Provider Interface. *Baeldung.* https://www.baeldung.com/java-spi

Baganz, J. (2017, 15. September). JUHU - NIE WIEDER TESTS SCHREIBEN. *ACCSO.* https://accso.de/magazin/property-based-testing/

Basil, V. R. & Turner, A. J. (1975). Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering.* https://doi.org/10.1109/TSE.1975.6312870

Bikos, K. & Buckle A. (o. D.). Warum gibt es Schaltsekunden? *timeanddate.de.* https://www.timeanddate.de/zeitzonen/schaltsekunden

Blog.ssanj.net. (2016, 26. Juni). Property-based Testing Patterns. https://blog.ssanj.net/posts/2016-06-26-property-based-testing-patterns.html

Böck, H. (2020, 23. Dezember). Irritierende Behördenmeldung über akzeptable Mailadressen. *golem.de.* https://www.golem.de/news/berlin-irritierende-behoerdenmeldung-ueber-akzeptable-mailadressen-2012-153029.html

Brichzin, P. (2015). Agile Softwareentwicklung – Erfahrungsbericht eines Oberstufenprojekts im Wahlpflichtunterricht. *Informatik allgemeinbildend begreifen.* https://dl.gi.de/bitstream/handle/20.500.12116/2026/63.pdf

Bush, D. M. (2018, 23. Januar). Property Based Testing Revealed - A Better Way to Test. *Dave's Notebook.* https://davembush.github.io/property-based-testing-revealed-a-better-way-to-test/

chrissikraus & Augsten, S. (2020, 28. Februar). Was bedeutet Implementierung? *DEV INSIDER.* https://www.dev-insider.de/was-bedeutet-implementierung-a-903068/

Cohn, M. (o. D.). Agiles Arbeiten – iterativ und inkrementell. *agile academy.* https://www.agile-academy.com/de/scrum-master/agiles-arbeiten-iterativ-und-inkrementell/

Computerlexikon.com. (1998, 14. August). Engine. https://www.computerlexikon.com/definition-engine

DER SPIEGEL. (2007, 31. Dezember). Millennium-Bug. Die Nacht, in der wir alle noch einmal davonkamen. *SPIEGEL Geschichte.* https://www.spiegel.de/geschichte/millennium-bug-a-948986.html

Disterer, G. & Rose, M. (2009). Nicht funktionale Anforderungen bei der Inbetriebnahme von Anwendungen. *HMD Praxis der Wirtschaftsinformatik,* 46, 80-91. https://link.springer.com/article/10.1007/BF03340328

doubleSlash. (o. D.). Software Design & Software Architektur. https://www.doubleslash.de/technologie/softwareentwicklung/software-design-software-architektur/

Dubien, N. (2018, 23. März). Introduction to Property Based Testing. *Criteo R&D Blog.* https://medium.com/criteo-engineering/introduction-to-property-based-testing-f5236229d237

Ebanhesaten, K. & Stenhorst, C. (2011). Testautomation in großen Softwareprojekten. *HMD Praxis der Wirtschaftsinformatik,* 48(2), 88-92. https://link.springer.com/content/pdf/10.1007/BF03340571.pdf

Ekssir-Monfared, M. (2010). Systemintegrationstest. *Softwaretechnik-Trends*, 30. http://pi.informatik.uni-siegen.de/stt/30_3/01_Fachgruppenberichte/TAV/07_Systemintegrationstest_GI_2spaltig.pdf

Elberzhager, F. & Naab, M. (2015). Nutzung von Architekturinformationen zur Beherrschung der Komplexität im Software-Integrationstest. *Softwaretechnik-Trends*, 35(3). http://pi.informatik.uni-siegen.de/stt/35_3/01_Fachgruppenberichte/2_38_GI-TAV_submission_4.pdf

Elektronik Kompendium. (2020). IPv4-Adressen. https://www.elektronik-kompendium.de/sites/net/2011211.htm

Epple, A. (o. D.). Lose Kopplung mit Java 9. *eppleton.* https://eppleton.de/news/lose-kopplung-mit-java-9_2017-06-15.html

form4. (o. D.). Iterative Softwareentwicklung. https://www.form4.de/methoden/iterative-softwareentwicklung/

Franz, K. (2007). *Handbuch zum Testen von Web-Applikationen.* Springer. https://doi.org/10.1007/978-3-540-68185-4_14

Friske, M. (2013). Fehlermanagement in Großprojekten – Erfahrungen und Best Practices. *Softwaretechnik-Trends*, 33(4). http://pi.informatik.uni-siegen.de/stt/33_4/01_Fachgruppenberichte/TAV/01_Friske.pdf

Friske, M. & Schlingloff, H. (2005). Von Use Cases zu Test Cases: Eine systematische Vorgehensweise. *MBEES.* https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.176.2203&rep=rep1&type=pdf#page=11

Glinz, M. & Fritz, T. (2006). Testen von Software. *Software Engineering.* https://www.ifi.uzh.ch/rerg/courses/archives/hs14/se/Vorlesungsmaterial/Kapitel_08_Testen.pdf

GNU. (o. D.). 3.7 The SUnit testing package. https://www.gnu.org/software/smalltalk/manual/html_node/SUnit.html

Günther, M. (o. D.). Property-based Testing mit ScalaCheck. *entwickler.de.* https://entwickler.de/leseproben/testing-scalacheck-579837780.html

Heimlich, A. (2018, 26. September). UNIT TESTS TUTORIAL – TESTAUTOMATISIERUNG AUF VERSCHIEDENEN STUFEN – TEIL 2. *SIMPLYTEST*. https://www.testautomatisierung.org/tutorial-testautomatisierung-unit-tests/

Hinrichs, J. (2011, 1. September). Unit Tests: Zeitverschwendung? *Projektmanagement Softwareentwicklung.* http://www.it-projektmanagement24.de/projektrolle/projektleiter/unit-tests-zeitverschwendung.html

Hughes, J. (2020). How to Specify It! In: W. Bowman, R. Garcia (Eds.), *Trends in Functional Programming*. (pp. 58-83). Springer, Cham. https://doi.org/10.1007/978-3-030-47147-7_4

Ionos. (o. D.). IPv6: Grundlagen. https://www.ionos.de/hilfe/server-cloud-infrastructure/ip-adressen/ipv6-grundlagen/

Ionos. (2020, 7. Mai). Dateiendung .jar: So öffnen und nutzen Sie eine .jar-Datei. https://www.ionos.de/digitalguide/server/knowhow/jar-format/

Ionos. (2020, 28. September). Was ist das V-Modell? https://www.ionos.de/digitalguide/websites/web-entwicklung/v-modell/

Jan-Dirk. (2020, 3. Februar). Was ist JUnit? Was sind JUnit Tests? *IT-Talents.* https://www.it-talents.de/blog/it-talents/was-ist-junit-was-sind-junit-tests

jqwik. (o. D. a). The jqwik User Guide. https://jqwik.net/docs/current/user-guide.html

jqwik. (o. D. b). Interface StringArbitrary. https://jqwik.net/docs/1.3.9/javadoc/net/jqwik/api/arbitraries/StringArbitrary.html

JAX TV. (2017, 4. Oktober). *JUnit-5-Test-Engine selbst gemacht* [Video]. vimeo. https://vimeo.com/236707523

Jürjens, J. (2012). Methodische Grundlagen des Software-Engineering. *TU Dortmund.* https://rgse.uni-koblenz.de/web/pages/teaching/ss12/mgse/tudo_intern/MGSE12_T4-1_Grundlagen_Softwaretesten.pdf

Kalenborn, A., Will, T., Thimm, R., Raab, J. & Fregin, R. (2006). Java-basiertes automatisiertes Test-Framework. *Wirtschaftsinformatik*, 48(6), 437-445 https://link.springer.com/content/pdf/10.1007/s11576-006-0096-9.pdf

Klensin, J. (2004). Application Techniques for Checking and Transformation of Names. *Network Working Group.* https://tools.ietf.org/html/rfc3696

Knapp, D. & Yilmaz, M. (2016). Tools und Techniken in der Testpyramide: Wo eine Größe nicht allen passt. *OBJEKTspektrum.* https://www.sigs-datacom.de/uploads/tx_dmjournals/Knapp_Yilmaz_OTS_Testing_16.pdf

Kraftfahrt-Bundesamt. (o. D.). Bestand. https://www.kba.de/DE/Statistik/Fahrzeuge/Bestand/bestand_node.html

Link, J. (2018, 16. Juli). Property-based Testing in Java: Patterns to Find Properties. *My Not So Private Tech Life.* https://blog.johanneslink.net/2018/07/16/patterns-to-find-properties/

Link, J. (2018, 29. März). Property-based Testing in Java: Jqwik - a JUnit 5 Test Engine. *My Not So Private Tech Life.* https://blog.johanneslink.net/2018/03/29/jqwik-on-junit5/

Link, J. (2020, 20. Oktober). Provide convenience method for email generation. *jqwik.* https://github.com/jlink/jqwik/issues/127

Link, J. (2020, 11. November). Add default generation for String with `@Email` annotation. *jqwik.* https://github.com/jlink/jqwik/issues/130

Link, J. (2020, 9. Dezember). Add jqwik Module for Generation of Dates and Times. *jqwik.* https://github.com/jlink/jqwik/issues/140

Link, J. (o. D.). How to Specify It! In Java! *johanneslink.net.* https://johanneslink.net/how-to-specify-it/

Luber, S. & Augsten, S. (2017, 10. Mai). Was ist eine IDE? *DEV INSIDER.* https://www.dev-insider.de/was-ist-eine-ide-a-600703/

Maciver, D. (o. D.). In praise of property-based testing. *increment.* https://increment.com/testing/in-praise-of-property-based-testing/

Macke, S. (2017, 20. März). Einführung in Build-Werkzeuge – Anwendungsentwickler-Podcast #97. *IT-BERUFE PODCAST.* https://it-berufe-podcast.de/einfuehrung-in-build-werkzeuge-anwendungsentwickler-podcast-97/

Macke, S. (2018, 9. Januar). Property-based Testing mit JUnit QuickCheck. *heise Developer.* https://www.heise.de/developer/artikel/Property-based-Testing-mit-JUnit-QuickCheck-3935767.html

Mauerer, W. & Jaeger, M. C. (2013). Open Source Engineering Processes. it - *Information Technology*, 55(5), 196-203. https://doi.org/10.1515/itit.2013.1008

Merdes, M. (2018, 8. Januar). Die Highlights von JUnit 5: Neues für Java-Tester. *JAXenter.* https://jaxenter.de/highlights-junit-5-65986

Müller, S. (2015, 27. November). DIE TEST-PYRAMIDE. *OPEN KNOWLEDGE.* https://www.openknowledge.de/test-pyramid/

Orlov, S. (2019, 2. Dezember). QS-Dilemma: Manuelles oder automatisiertes Testen. *innowise.* https://innowise-group.com/de/blog/qs-dilemma-manuelles-oder-automatisiertes-testen-16

Panitz, S. E. (2010). Objektorientierte Softwareentwicklung. *Hochschule RheinMain.* https://www.cs.hs-rm.de/~panitz/oose/skript.pdf

Papadakis, M. & Sagonas, K. (2011). A PropEr integration of types and function specifications with property-based testing. In Proceedings of the 10th ACM SIGPLAN workshop on Erlang (Erlang '11). *Association for Computing Machinery, New York*, NY, USA, 39–50. https://doi.org/10.1145/2034654.2034663

Parthesius, M. (2020, 20. Januar). Y2K38: Das Jahr-2038-Problem. *Mac Life.* https://www.maclife.de/news/y2k38-jahr-2038-problem-100115824.html

Peters, M. (2015, 31. Mai). FAQ - was ist das? Einfach und verständlich erklärt. *CHIP.* https://praxistipps.chip.de/faq-was-ist-das-einfach-und-verstaendlich-erklaert_41263

PHP. (o. D.). Was ist PHP? https://www.php.net/manual/de/intro-whatis.php

Picot, A. & Fiedler, M. (2008). Open Source Software und proprietäre Software. In: Depenheuer O., Peifer KN. (Eds.) *Geistiges Eigentum: Schutzrecht oder Ausbeutungstitel* (pp. 165-185). Springer. https://doi.org/10.1007/978-3-540-77750-2_13

Redaktion JAXenter. (2017, 12. Oktober). JUnit 5: So bastelt man seine eigene Test-Engine. *JAXenter.* https://jaxenter.de/junit-5-test-engine-selbst-gemacht-62472

Schmidt, A. & Schönwald, R. (2005). Iterativ-inkrementelles Software-Engineering in komplexen IT-Projekten. *Software Engineering 2005.* https://dl.gi.de/handle/20.500.12116/28323

Semenova, E. (o. D.). Vorteile und Nachteile der Testautomatisierung aus Projektperspektive. *redbots.* https://redbots.de/blog/vorteile-nachteile-testautomatisierung/

simplifier. (2017, 21. Juli). Die 6 Phasen der Softwareentwicklung – Teil 1. https://simplifier.io/die-6-phasen-der-softwareentwicklung/

SIMPLYTEST. (o. D.) Integrationstests. https://www.testautomatisierung.org/lexikon/integrationstests/

Tüv Nord. (2016, 8. September). Wofür steht die Bezeichnung ISO 9001:2015? https://www.tuev-nord.de/explore/de/erklaert/wofuer-steht-die-bezeichnung-iso-90012015/

Ullenboom, C. (2010, 15. April). JUnit 4 Tutorial, Java-Tests mit JUnit. *Java Blog für Programmierer.* http://www.tutego.de/blog/javainsel/2010/04/junit-4-tutorial-java-tests-mit-junit/

Verifysoft. (2012, 19. Juli). Entwicklertests. https://www.verifysoft.com/de_entwicklertests.html

Verifysoft. (2017, 2. Juni). Unit-Tests. https://www.verifysoft.com/de_unit_test.html

Voß, T. (2017, 16. Oktober). JUNIT 5 – DIE NEUE GENERATION DES UNIT-TESTENS. *JAVAPRO*. https://javapro.io/junit-5-die-neue-generation-des-unit-testens/

Was War Wann? (o. D.). Jahr 0 – Der Beginn war nicht der Anfang der Zeitrechnung. https://www.was-war-wann.de/000/00/0.html

welt. (2010, 12. September). Beginn einer neuen Zeitrechnung. https://www.welt.de/iphone_app/historyapp/article9585787/Beginn-einer-neuen-Zeitrechnung.html

Zachmann, G. (2010, 26. April). Informatik II Schleifeninvarianten. *Clausthal University, Germany.* https://cgvr.informatik.uni-bremen.de/teaching/info2_10/folien/07_invarianten.pdf