WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

Diplomarbeit

zum Thema

Generierung von Testdaten für randomisiertes Testen

eingereicht an der
Fakultät Physikalische Technik/Informatik der
Westsächsischen Hochschule Zwickau

Zur Erlangung des akademischen Grades
Diplom-Informatiker (FH)

vorgelegt von
Vetter, Sven
Matrikelnummer: 41821

1. Gutachter:	Prof. Dr. rer. nat. Ralf Laue Westsächsische Hochschule Zwickau
2. Gutachter:	Prof. Dr. Peter Mißbach Media Projekt GmbH
Ausgabedatum:	23.02.2024
Abgabedatum:	26.07.2024

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
EBT	Example Based Test
FQDN	Fully Qualified Domain Name
HTTP	Hyper Text Markup Language
IANA	Internet Assigned Numbers Authority
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
LCG	Linear Congruential Generator
MBT	Model Based Test
PBT	Property Based Test
RCPT	Recipient
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
TLD	Top-Level-Domain
VB	Visual Basic
UI	User Interface

Glossar

API	Programmschnittstelle, der von einem Softwaresystem für andere Programme zur Anbindung zur Verfügung gestellt wird und Funktionen bereitstellt
Annotation	Strukturelement, welches zusätzliche Metadaten bereitstellt. Wird vom Compiler bei der Übersetzung nicht beachtet. Annotationen werden einer Klassendefinition hinzugefügt
ASCII	Amerikanischer Code für Informationsaustausch innerhalb von IT-Systemen. Umfasst das lateinische Alphabet, sowie die zehn arabischen Ziffern, einige Satzzeichen und verschiedene Steuerzeichen, wie z. Bsp. Tabulatoren oder Zeilenvorschübe
Boilerplate-Code	Quellcode-Segmente, die sich an mehreren Stellen mit geringen oder gar keinen Änderungen wiederholen
C#	Typsichere objektorientierte Programmiersprache, die im Rahmen der .NET-Plattform entwickelt und auf diese optimiert wurde
Domain	Einzigartige Adresse, die für Webseiten oder andere Webdienste verwendet wird
Extension	Konzept, mit dem bestehende Klassen um weitere Methoden erweitert werden können, ohne den Quellcode der Originalklasse verändern zur müssen
Exception	Unerwartetes Ereignis, das während der Programmausführung auftritt
F#	Funktionale Programmiersprache entwickelt von Microsoft
Framework	Programmiergerüst, das grundlegende Struktur und Funktionen bereitstellt
.NET	Software-Plattform zur Entwicklung und Ausführung von Anwendungen
NuGet	Paketmanager für .NET
Repository	Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten

REST	Architektur-Paradigma für die Entwicklung von Webservices
Single Page Application	Einzelne Webseite, die Inhalte dynamisch aktualisiert
StackOverflow	Internetplattform für Softwareentwicklung zum Teilen von Programmierkenntnissen
Testsuite	Menge von Testskripten, die im Rahmen eines Testlaufs ausgeführt werden
Test-Framework	Sammlung von Richtlinien und Tools zur Erstellung von Softwaretests

1 Einleitung

Softwaretests nehmen eine wichtige Rolle in der Softwareentwicklung ein, indem sie zur Verbesserung der Codequalität beitragen und den Wartungsaufwand reduzieren. Durch das Aufdecken von Fehlern während der Entwicklungs- und Testphase können Probleme frühzeitig adressiert und gelöst werden. Die Zuverlässigkeit, mit der ein System arbeitet, hängt davon ab, wie hoch der Aufwand ist, der für das Testen aufgebracht wird. Mit steigender Testanzahl steigt auch der Aufwand, der für die Bereitstellung von Testfällen aufgebracht werden muss.

Eine Möglichkeit diesen Aufwand möglichst auf ein Minimum zu reduzieren, stellen Test-Frameworks wie xUnit oder NUnit zur Verfügung. Mit diesen können sogenannte Example Based Tests (EBT) erstellt und in Testsuites gruppiert und verwaltet werden. In diesem Kontext bedeutet ein solcher Test, dass eine einzelne Einheit gegenüber einzeln fest vorgegebenen Eingabeparametern getestet wird und ein konkretes Ergebnis erwartet wird. Zu diesem Ergebnis wird im Anschluss eine Annahme getroffen. Meist stellt diese einen Vergleich mit einem Wert dar, der als erwartetes Ergebnis definiert wird.

EBT testen einzelne Einheiten lediglich stichprobenartig mit Eingabeparametern, die vom Entwickler fest vorgegeben sind. Problematisch wird dies, wenn solche Einheiten nur mit bestimmten Konstellationen von Eingabeparameter Fehler verursachen und diese vom Entwickler in dem Test nicht berücksichtigt werden. Zufallsgenerierte Tests, zu denen auch Property Based Tests (PBT) gehören, folgen einem anderen Schema, welches versucht, die Problematik mit Tests gegenüber endlichen Mengen von Eingabeparametern, so wie es bei EBT der Fall ist, zu lösen.

1.1 Problemstellung und Abgrenzung

Die Generierung von zufällig erzeugten Testdaten stellen mehrere Herausforderungen dar. Die generierten Testdaten müssen so gestaltet werden, dass sie sich realen Daten mit denen sie verglichen werden, möglichst annähern. Dies erfordert ein tieferes Verständnis der Datenstrukturen und -typen, die im Kontext des zu testenden Systems relevant sind. Viele Datentypen besitzen spezifische Regeln oder Muster, die eingehalten werden müssen, damit es sich um einen gültigen Wert handelt. Bei der Generierung von Testdaten ist demzufolge auf die Einhaltung der Spezifikationen

zu achten. Zudem ist neben der Qualität der generierten Testdaten auch die effiziente Generierung dieser ein entscheidender Faktor. Der Prozess zur Generierung muss effizient gestaltet sein.

Zunächst soll eine Evaluierung bestehender Testbibliotheken im .NET-Umfeld durchgeführt werden. Dabei soll geprüft werden, inwiefern zufallsgenerierte Tests bereits umsetzbar sind und welche Datentypen diese Bibliotheken bereits unterstützen. Darüber hinaus sollen die Bibliotheken anhand einer Festlegung verschiedener Kriterien bewertet und auf deren Ergebnis aufbauend der Fokus auf eine Bibliothek festgelegt werden.

Ziel soll eine Erweiterung dieser Testbibliothek sein, auf deren Basis weitere Datentypen generierbar gemacht werden sollen. Dabei ist auf die Einhaltung der dafür notwendigen Spezifikationen zu achten. Datentypen können außerdem innerhalb dieser Spezifikation auch unterschiedliche Kriterien erfüllen. Dies erfordert die Implementierung einer Möglichkeit diese Kriterien in einer bestimmten Weise steuerbar zu machen. Die Arbeit strebt darüber hinaus an, die entwickelte Erweiterung der Testbibliothek der Gemeinschaft zur Verfügung zu stellen. Die Bereitstellung soll dafür in Form eines öffentlichen GIT-Repositories erfolgen, sodass Entwickler darauf aufbauen und eine Weiterentwicklung durchführen können.

1.2 Motivation

Die Arbeit befasst sich mit der Generierung von zufallsgenerierten Testdaten. Dabei wird zunächst auf die Grundlagen von „Automatisierten Softwaretests“ (Kapitel 2) und „Property Based Testing“ (Kapitel 3) eingegangen. Im Anschluss dazu wird eine Evaluierung von Testbibliotheken zum zufallsgenerierten Testen durchgeführt und die Festlegung auf eine der Bibliotheken getroffen (Kapitel 4). Darauf aufbauend werden Datentypen untersucht, die von der Testbibliothek nicht unterstützt werden und deren technische Spezifikationen untersucht (Kapitel 5). Als Ergebnis aus der Untersuchung erfolgt die Realisierung durch Implementierung entsprechender Generatoren auf Grundlage der untersuchten Datentypen, sowie der Option zur Individualisierung der generierten Testdaten durch Anpassen des Generatorverhaltens (Kapitel 6). Abschließend werden Testmethoden analysiert, mit denen die generierten Werte auf Einhaltung ihrer Spezifikation getestet werden können (Kapitel 7).

2 Automatisierte Softwaretests

Das Kapitel geht auf die Grundlagen von Softwaretests ein, indem es die unterschiedlichen Phasen, beginnend von Unit Tests bis hin zu Systemtests, untersucht. Es wird die Rolle von Test-Frameworks im Kontext der Automatisierung betrachtet. Schlussendlich wird eine Anwendung präsentiert, um die praktische Anwendung der einzelnen Phasen näher betrachten und verdeutlichen zu können.

Softwaretests sind ein Prozess zur Überprüfung und Validierung eines Softwaresystems um sicherzustellen, dass die Ausführung fehlerfrei bleibt. Diese Tests identifizieren Defekte und Fehler in der Implementierung [1]. Werden Änderungen am Quellcode durchgeführt, erfordert dies ein erneutes Testen des System. Damit wird gewährleistet, dass Änderungen keinen Einfluss auf die restlichen Bestandteile einer Implementierung haben. Eine manuelle Ausführung erfordert nicht nur einen erhöhten Zeitaufwand bei der Erfassung und Prüfung, aufgrund der vielen Wiederholungen besteht die Gefahr in eine Routine zu verfallen, wodurch das Risiko, dass Fehler unentdeckt bleiben, steigt. Die Automatisierung dieser Tests stellt sicher, dass die Entwicklung im Vordergrund stehen kann. Der Zeitaufwand, der für die Testphase aufgewendet werden muss, sinkt. [2]

2.1 Beispielanwendung

Um die verschiedenen Testphasen zu veranschaulichen, wurde eine simple Beispielanwendung entwickelt. Sie zeigt einen einfachen Webservice in Form einer REST-Schnittelle. Der Service kann mit Hilfe des HTTP-Protokolls angesprochen werden. Informationen können damit an den Service übertragen werden, wo sie verarbeitet und in einer Datenbank gespeichert werden. Im umgekehrten Fall können diese Informationen auch wieder abgerufen werden.

Abbildung 1 zeigt den grundlegenden Aufbau und Workflow der Web-API. Die Controller stellen die Endpunkte zur Verfügung und behandeln sowohl HTTP-Requests, wie auch HTTP-Responses. Die Repository-Klassen stellen eine Schnittstelle zur Datenbank dar und kümmern sich um Lese- und Schreiboperationen von und zum persistenten Speicher. Aufgebaut ist der Service auf Basis der Drei-Schichten-Architektur. Die Geschäftslogik ist jedoch einfach genug gehalten, wodurch auf die Service-Schicht in diesem konkreten Beispiel verzichtet wurde. Ein Verwendung würde den

Aufbau lediglich zusätzlich verkomplizieren, ohne einen realistischen Mehrwert zu generieren. Stattdessen kümmert sich zusätzlich der Controller um die Ausführung der Geschäftslogik.

Es empfiehlt sich jedoch im Produktivbetrieb auf eine separate Service-Schicht zu setzen, die sich um die Verarbeitung der Informationen kümmert. Der Autor Martin Fowler beschreibt dies in seinem Buch „Patterns of Enterprise Application Architecture“, sobald mehrere Clients oder andere transaktionelle Ressourcen vorgesehen sind, dass sich die Implementierung einer Service-Schicht lohnt. [3, S.129]

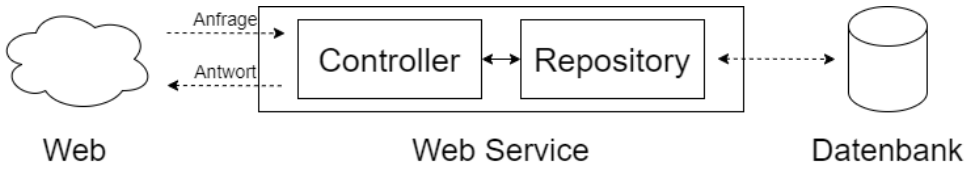

Abbildung 1: Aufbau des Webservices

Die Web API stellt eine Reihe von Endpunkten bereit, die über das HTTP-Protokoll angesprochen werden können.

Methode	Endpunkt	Beschreibung
Get	/persons	Lieferte eine Liste mit allen Personen zurück
Post	/persons	Fügt eine neue Person zur Liste hinzu
Delete	/persons	Entfernt eine Person aus der List

Tabelle 1: Endpunkt der Web API

2.2 Testebenen

Softwaretests bestehen aus zahlreichen Einzelmaßnahmen, die üblicherweise über mehrere Testebenen hinweg ausgeführt werden. Daraus ergeben sich individuelle Testziele für jeden Testfall und jede Teststufe. Die Testautomatisierungspyramide wurde ursprünglich vom Autor Mike Cohn eingeführt. Sie beschreibt die unterschiedlichen Testphasen und deren Verhältnis zueinander. Dabei beschreibt Cohn sein Konzept damit, dass an der Basis die meisten Tests ausgeführt werden sollten und sich

diese Zahl mit zunehmender Steigerung in den Schichten der Pyramide immer weiter reduzieren sollte. Er begründet dies mit der zunehmenden Komplexität, die Tests innerhalb der jeweiligen Schicht annehmen, sowie der damit verbundenen Kostensteigerung sowie der Tatsache, dass Tests aufgrund der zunehmenden Komplexität weitaus langsamere Ausführungszeiten aufweisen. [2, S. 307-308]

Die Definition des Begriffs „Systemtest“ ist nicht eindeutig. Cohn beschreibt in seiner Definition die oberste Schicht als UI-Tests. Ham Vocke beschreibt in einem Artikel, dass das Konzept in der Form so in der Praxis nicht immer umsetzbar ist [4]. So gibt es Anwendungsfälle, wie zum Beispiel die in Kapitel 2.1 eingeführte Web API, in denen keine grafische Bedienoberfläche vorhanden ist. Dementsprechend ist die Beschreibung der Testpyramide ungenau. Eine Bedienoberfläche ließe sich im vorliegenden Fall bspw. als Webanwendung implementieren. In der heutigen Zeit, in der Single Page Applications ein gängiges Mittel zur Implementierung einer Webanwendung darstellen, ließen sich für diese eigenständige Unit Tests implementieren. Die Testpyramide würde dementsprechend hier in vollem Umfang Anwendung finden.

Der Autor Adrian Kröger geht in seiner Literaturanalyse zu „Methoden automatisierter Softwaretests in modernen DevOps Umgebungen“ auf die Testpyramide ein und unterstützt die These, dass die Begriffe bei verschiedenen Autoren unterschiedliche Anwendung finden [5]. Martin Fowler beschreibt diese in seinem Buch „Software Testing – A Craftman’s Approach“, ebenfalls als Systemtest [6, S. 3]

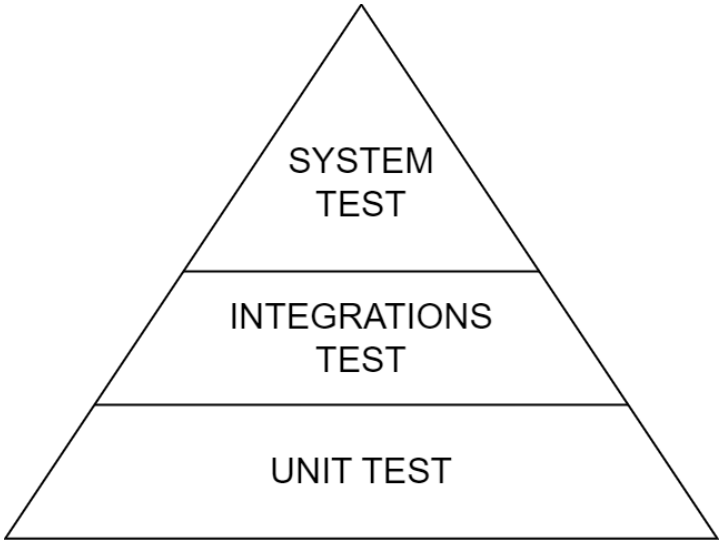

Abbildung 2: Testpyramide nach Cohn

2.2.1 Unit Test

Im Lauf der Jahre sind viele Definitionen entstanden, was einen Unit Test beschreibt, wie er definiert wird und was ihn auszeichnet. Diese Unstimmigkeiten werden dadurch belegt, dass diese Art von Test oft mit anderen Testtypen, wie bspw. einem Integrationstest, verwechselt wird. Einfach ausgedrückt beschreibt ein Unit Test eine Arbeitseinheit. Im Kontext von Softwareentwicklung definiert sich eine solche Arbeitseinheit als Funktion oder eine Methode einer bestimmten Objektklasse. [7, S. 19-20]

Der Fokus bei einem Unit Test wird auf eine einzelne Einheit gelegt. Ein weiterer Begriff wäre in diesem Zusammenhang auch eine Komponente, weshalb bei einem Unit Test auch von einem Komponententest gesprochen wird. Unterschiedliche Komponenten werden dabei vollkommen losgelöst und getrennt voneinander auf ihre Funktionalität geprüft [7, S. 19-20]. Das vereinfacht das Testen, wenn Implementierungsdetails anderer Komponenten außer Acht gelassen werden können. Weist die Komponente jedoch direkte Abhängigkeiten auf, beeinflussen diese die Implementierung der Komponente, sodass keine isolierte Betrachtung mehr möglich ist. Daher muss in diesen Fällen sichergestellt werden, dass Abhängigkeiten durch entsprechende Platzhalter ersetzt werden, deren Verhalten festgelegt ist und somit stets identische Testbedingungen garantieren. [7, S. 19-20]

Unit Tests können auf unterschiedliche Art und Weise implementiert werden. Dennoch teilen alle Unit Tests die gleichen Merkmale. So ist einer ihrer Hauptvorteile ihre Geschwindigkeit. Je kürzer die Ausführungszeit ist, umso öfter kann eine Testsuite ausgeführt werden [8, S. 79]. Sie führen einen bestimmten Codeabschnitt isoliert und damit unabhängig vom restlichen Code aus. Dies schließt auch externe Faktoren wie Abhängigkeiten mit ein. Sind Tests voneinander abhängig, sind diese zustandsbehaftet. Damit besteht die Gefahr, dass Änderungen eines Tests unweigerlich die Ergebnisse anderer Tests beeinflussen und dafür sorgen, dass diese fehlschlagen. Ein solches Verhalten deutet auf eine schlechte Isolation hin. Damit wird auch das Merkmal der Wiederholbarkeit beeinträchtigt. Unit Tests sind abgekoppelt von einem initialen Status und können wiederholt ausgeführt werden. Die Reihenfolge der Ausführung darf dabei keine Rolle spielen und muss vollkommen willkürlich möglich sein. Das Ergebnis eines Tests muss zudem vorhersagbar sein. Tests müssen demzufolge ein deterministisches Verhalten aufweisen [7, S. 20].

Bevor ein Test ausgeführt werden kann, müssen im Vorfeld die Bedingungen für das zu testende Szenario eingerichtet werden. Im einfachsten Fall genügt es dazu die Klasse, deren Implementierung im Anschluss geprüft werden soll, zu instanziieren [9, S. 156-157]. Spielen in die Implementierung, wie in Listing 1 gezeigt, weitere externe Faktoren in Form von Abhängigkeiten mit ein, müssen diese im Vorfeld gleichermaßen darauf eingestellt werden, sodass bei diesen für jede Testausführung das gleiche Verhalten zu erwarten ist. Damit die Wiederholbarkeit und Vorhersagbarkeit gewährleistet bleiben, muss sichergestellt werden, dass auch für die aufgerufenen Methoden der direkten Abhängigkeit stets die gleichen Werte zurückgeliefert werden.

```
1  class EmailIsInUse : IValidatorAsync<string>
2  {
3      private readonly IPersonRepository _repository;
4
5      public EmailIsInUse(IPersonRepository repository)
6      {
7          _repository = repository;
8      }
9
10     public async Task<bool> IsValid(string email)
11     {
12         var person = await _repository.ReadOneBy(email);
13         return person != null;
14     }
15 }
```

Listing 1: Klasse validiert, ob eine E-Mail-Adresse bereits verwendet wird

Sind die Bedingungen für das zu testende Szenario eingerichtet, gilt es die Methode aufzurufen. Dabei sollte darauf geachtet werden, dass der Test nur eine Interaktion mit der zu testenden Klasse umfasst. Eine Interaktion kann einen Methodenaufruf oder das Abrufen oder Ändern einer Eigenschaft darstellen [10]. So wird sichergestellt, dass ein Test leicht zu verstehen bleibt. Die Ausführung von mehreren Interaktionen innerhalb eines Tests verkompliziert die Fehlersuche im Falle eines fehlgeschlagenen Tests.

Wurde eine Interaktion durchgeführt, gilt es das Ergebnis zu überprüfen. Hier entscheidet sich, ob der Test fehlschlägt oder erfolgreich gewesen ist. Wie die Überprüfung aussieht, hängt von der Art des Test ab. Dabei sollte darauf geachtet werden, je-

den Testfall auf eine Annahme (Assert) zu begrenzen. Dave Astels begründet das in seinem Artikel „One Assertion Per Test“ damit, dass sich Tests auf einen möglichst kleinen spezifischen Aspekt fokussieren sollten [10]. Das mach den Test nicht nur einfacher zu lesen, sondern führt zu einer besser steuerbaren Weiterentwicklung.

```
1  [Fact]
2  public void Sum_WhenTwoIntegerValueProvided_ReturnsSumOfValues()
3  {
4      // arrange
5      var calc = new Calculator();
6
7      // act
8      var testResult = calc.Sum(5, 3);
9
10     // assert
11     var expectedResult = 8;
12     Assert.Equal(expectedResult, testResult);
13 }
```

Listing 2: Ein Unit Test besitzt drei Phasen

Ein Test besteht aus den drei Teilen Arrange, Act und Assert. Diese bilden das AAA-Pattern. Dabei handelt es sich um ein Muster, welches den Quellcode eines Tests in eine einheitliche Struktur untergliedert. Der Vorteil ist, dass Tests dadurch einfach zu lesen und zu verstehen sind. Einfacher zu lesende Tests führen zu weniger Wartungszeit und damit verbunden zu weniger Kosten. Das Arrange-Act-Assert-Pattern ist auch bekannt als Given-When-Then-Pattern. Technisch gesehen gibt es im Vergleich zum AAA-Pattern keinen Unterschied. Lediglich die Struktur ist für Menschen ohne Programmierkenntnisse besser verständlich, was den einzigen Unterschied darstellt. [8, S. 42-43]

2.2.2 Integrationstests

Das oberste Ziel eines Integrationstests ist die Sicherstellung der Integration unterschiedlicher Komponenten. Eine andere Definition dafür wäre, dass mit einem solchen Test die Funktionalität von Abhängigkeiten überprüft wird. Wie in Kapitel 2.2.1 beschrieben, dienen Unit Tests dazu, die Geschäftslogik einzelner Komponenten auf lokale Fehler zu prüfen. Die isolierte Betrachtung reicht jedoch nicht aus. Integrationstests prüfen daher das Zusammenspiel zwischen unterschiedlichen Komponen-

ten und externen Systemen. Ein Unit Test prüft das Verhalten einer einzelnen Komponente, ist schnell ausführbar und führt eine isolierte Betrachtung durch. Erfüllt ein Test nicht wenigstens eine dieser Voraussetzungen, handelt es sich nicht mehr um einen Unit Test, sondern um einen Integrationstest. [8, S. 186-187]

```
1  class EmailIsInUse : IValidatorAsync<string>
2  {
3      private readonly IPersonRepository _repository;
4
5      public EmailIsInUse(IPersonRepository repository)
6      {
7          _repository = repository;
8      }
9
10     public async Task<bool> IsValid(string email)
11     {
12         var person = await _repository.ReadOneBy(email);
13         return person != null;
14     }
15 }
```

Listing 3: Klasse zur Überprüfung ob eine E-Mail-Adresse bereits von einer Person verwendet wird

In der Praxis prüfen Integrationstests stets die Interaktion von Komponenten in Verbindung mit externen Abhängigkeiten. Solche Abhängigkeiten liegen außerhalb des Systems [8, S. 186-187]. Die Klasse „EmailInUse“ in Listing 3 besitzt eine Abhängigkeit zu einer Implementierung der Schnittstelle „IPersonRepository“. Das Repository Entwurfsmuster stellt eine Möglichkeit zur Verfügung die Datenzugriffslogik zu und von einer Datenquelle zu zentralisieren. Dabei gilt es zu beachten, dass ein Integrationstest auch einen Unit Test repräsentieren kann, wenn dessen Abhängigkeiten vollständig durch Platzhalter (Mock) ersetzt werden können. Der Test wird dadurch wieder isoliert und bleibt schnell ausführbar.

Softwaresysteme besitzen oftmals eine solche externe Abhängigkeit. Im Normalfall repräsentiert dies eine Datenbank. Das bedeutet, dass ein Verbindungsaufbau zur Datenbank notwendig ist, sowie die Datenabfrage bzw. Manipulation von Informationen. Das sorgt nicht nur für eine verzögerte Ausführungszeit, sondern führt bei der Ausführung von mehreren Entwicklern im Parallelbetrieb auch zu einem Konkurrenzverhalten. Die Datenbank stellt bei der Ausführung nicht die gleichen Ausgangsvoraussetzungen bereit, wodurch die Wiederausführbarkeit beeinträchtigt wird, sowie

die Fehlersuche erschwert wird. Auch stellt eine nicht erreichbare Datenbank zum Ausführungszeitpunkt eine Fehlerquelle dar.

```
public class PersonRepositoryStub : IPersonRepository
{
    private List<Person> _persons = new List<Person>();

    public PersonRepositoryStub()
    {
        _persons = new List<Person>()
        {
            new Person()
            {
                Id = 1,
                Email = "demo.user@example.com",
                FirstName = "Demo",
                LastName = "User"
            },
            // ...
        };
    }

    public Task<Person?> ReadOneBy(string email)
    {
        var person = _persons.FirstOrDefault(p => p.Email.Equals(email));
        return Task.FromResult(person);
    }

    // ...
}
```

Listing 4: Stub eines PersonRepositories

Um solche Seiteneffekte zu vermeiden, werden diese Abhängigkeiten durch Mock-Objekte ersetzt. Hierbei wird die Annahme getroffen, dass die für den Test benötigten Abhängigkeiten bereits durch Unit Tests in ihrer Funktionalität abgesichert wurden. Mock-Objekte sind sogenannte Ersetzer, die ein vorgegebenes Verhalten besitzen. Dadurch kann sichergestellt werden, dass Abhängigkeiten stets ein kontrolliertes Ergebnis zurückliefern.

Der Begriff „Mock“ stellt einen Überbegriff für eine Reihe von Stellvertretern, die externe Ressourcen ersetzen sollen, dar. Ihr Verhalten ist vordefiniert. Damit wird gewährleistet, dass diese Objekte stets das gleiche erwartete Ergebnis zurückliefern. Die beiden am Häufigsten verwendeten Typen stellen dabei „Mock“ und „Stub“ dar. Der Zweck von Stubs ist das Nachahmen von konkreten Implementierungen, indem

es fest vorgegebene Werte zurückliefert. Der Test kann damit trotz externer Abhängigkeiten in einer isolierten Testumgebung ausgeführt werden. Der Entwickler hat damit die Kontrolle über den Testablauf und kann Annahmen treffen. Mocks wiederum prüfen die Interaktion zwischen Objekten und deren Abhängigkeiten. Sie stellen damit sicher, dass diese dem erwarteten Verhalten entsprechen [11].

Der entscheidende Unterschied zwischen einem Stub und einem Mock besteht demzufolge in ihren Aufgaben. Stubs fokussieren sich auf den Zustand und das Verhalten der zu testenden Einheit. Während also ein Stub dabei unterstützt, eingehende Interaktionen zu simulieren, werden Mocks verwendet, um ausgehende Interaktionen zu simulieren, wenn zum Beispiel Daten an ein Repository gesendet werden und diese eine Änderung an der Datenquelle hervorrufen [11].

Die Validierungsklasse in Listing 3 besitzt eine Abhängigkeit zu einem PersonRepository, welches Daten aus einer Datenbank abruft oder speichert. Da die Klasse lediglich Kenntnis über die zu implementierende Schnittstelle IPersonRepository besitzt, kann die Abhängigkeit ohne Probleme durch eine weitere Implementierung ersetzt werden. Listing 4 zeigt eine mögliche Implementierung eines Stub, welches die eigentliche Implementierung ersetzen kann. An dieser Stelle wird auf die Nutzung eines Stubs zurückgegriffen. Die Validierungsklasse nutzt für ihre Implementierung die Abhängigkeit, um Informationen abzurufen. Abgesehen davon finden keine Interaktionen mit dieser statt. Demzufolge muss der Platzhalter lediglich Informationen zurückliefern und sicherstellen, dass diese einer klaren Vorgabe folgen, damit das Verhalten stets dem gleichen Muster folgen kann. Der Stub liefert bei jeder Interaktion die gleichen Datensätze zurück.

2.2.3 Systemtests

Bei Systemtests werden alle Komponenten einer Anwendung untersucht. Damit wird sichergestellt, dass alle Komponenten des Systems als einheitliches Ganzes funktionieren. Ziel dieser Art von Tests ist es nicht, Fehler zu ermitteln, sondern das geforderte Verhalten zu überprüfen [6, S. 253]. Im Gegensatz zum Komponenten- oder Integrationstest, die gegen technische Spezifikationen prüfen, wird bei Systemtests aus Sicht des Kunden geprüft. Die Ausführung solcher Tests erfolgt dabei nah an der späteren produktiven Umgebung. Ziel ist es zu validieren, ob das System die gestellten Anforderungen, sowohl funktionale als auch nicht funktionale, erfüllt. Merkmale

nicht funktionaler Anforderungen können bspw. die Performance, Benutzbarkeit oder die Kundenzufriedenheit repräsentieren. [12, S. 57-60]

Systemtests sind äußerst komplex und erfordern einen hohen Aufwand bei ihrer Ausführung, weshalb ihre Ausführungszeit im Vergleich zu Komponenten- oder Integrationstests bedeutend schlechter ausfällt. Dementsprechend bilden sie den geringsten Anteil innerhalb der Testphase.

2.3 Test-Frameworks

Test-Frameworks stellen Richtlinien und Regeln bereit, die beim Erstellen von Testfällen verwendet werden. Sie stellen ein einheitliches Programmiermodell zur Verfügung. So lassen sich Testfälle als einfache Klasse mit Methoden definieren und besser organisieren. Test-Frameworks stellen sogenannte Test Runner bereit, womit Tests durch einen simplen Mausklick ausgeführt werden können. Tests können damit problemlos eingerichtet, hinzugefügt oder wieder entfernt werden. Schlagen Tests fehlt, liefert der Test Runner detaillierte Informationen über den Grund des Fehlschlags, einschließlich verfügbarer Ausnahmeinformationen.

Die am weitesten verbreiteten Test-Frameworks in .NET sind xUnit und NUnit. Beide Frameworks besitzen eine hohe Popularität. Während NUnit seit seiner Entwicklung 2002 eine große Community aufgebaut hat und eine umfangreiche Dokumentation besitzt, wurde es inzwischen von xUnit als populärstes Test-Framework abgelöst. Im Unterschied ist xUnit einfacher strukturiert und lässt sich besser erweitern. Es wird zudem von der .NET Foundation unterstützt und ist Teil der neueren Versionen von .NET Core. Dementsprechend wird für die Testausführung innerhalb der Arbeit auf das Test-Framework xUnit gesetzt.

3 Property Based Testing

Das Kapitel beleuchtet Property Based Testing (PBT) als Technik zum Testen mit zufällig generierten Daten. Es deckt die historische Entwicklung von PBT, seinen schematischen Aufbau, sowie die zugrundeliegenden Techniken und Konzepte ab. Darüber hinaus klärt das Kapitel über die Einordnung innerhalb der Testpyramide auf.

3.1 Ursprung und Abgrenzung

Die Autoren Hughes und Koen beschreiben in einem Artikel im Jahr 2000 ein von ihnen selbst entwickeltes Framework namens „QuickQueck“. Dieses bot Haskell-Programmierern die Möglichkeit sogenannte Properties zu definieren und diese gegen zufallsgenerierte Eingabeparameter auszuführen [13]. Die Idee basiert auf einem Artikel von Sergio Antoy und Dick Hamlet aus dem Jahr 1992, in dem sie untersuchten Software nicht mit gezielten Einzelwerten, sondern gegen eine formale Spezifikation zu testen. Das Ziel sollte es sein, die Prüfung auf Konsistenz und Vollständigkeit zu ermöglichen, insbesondere wenn weitere Operationen zu einem Datentyp hinzugefügt werden [14].

Neben der Testmethode des Testens auf Properties gibt es auch noch einen Ansatz, der auf Basis von Modellen Tests durchführt. Dieser nennt sich Model Based Testing (MBT) und verfolgt das Testen unter Verwendung von expliziten Modellen, um das Verhalten eines Systems zu beschreiben. Dieser soll kein Gegenstand dieser Arbeit sein.

3.2 Definition

Wie bereits in Kapitel 2.2.1 beschrieben, kann ein Test zum Testen eines bestimmten Codeabschnittes in drei Teile unterteilt werden:

1. Auswählen eines gültigen Eingabeparameters und bestimmen der erwarteten Ausgabe
2. Ausführen des Programmcodes und Ausgabeparameter erfassen
3. Entscheiden, ob die Kombination aus Ein- und Ausgabeparameter dem erwarteten Verhalten entspricht

Das Ziel ist es möglichst alle Schritte zu automatisieren. Bei näherer Betrachtung lässt sich feststellen, dass Schritt zwei heutzutage vollständig durch Test-Frameworks automatisiert werden kann. Bereits mehrere Ansätze wurden unternommen, um den letzten Schritt zu automatisieren. Dazu haben Entwickler in den meisten Fällen eine Art von Spezifikation ihres Quellcodes definiert. Dies scheiterte daran, dass die formale Spezifikation die Größe des eigentlichen Quellcodes erreicht hatte [15]. Die Generierung von gültigen Eingabeparametern auf eine effiziente Art und Weise wurde ebenfalls untersucht. Schlussendlich liegt die Verantwortung, geeignete Eingabeparameter zu definieren weiterhin beim Entwickler [16].

Mit Property Based Tests als Ansatz wird das Ziel verfolgt, gültige Eingabeparameter automatisiert zu generieren. Tester definieren dafür bestimmte Regeln. Die Überprüfung der Ergebnisse erfolgt nach der Testdurchführung auf Basis von Eigenschaften. Dies unterscheidet diese Testmethode vom klassischen Test, bei dem ein erwarteter Wert mit dem Ergebnis der Ausführung verglichen wird und damit eine gewisse Erwartungseinhaltung einhergeht.

3.2.1 Property

Eine Property ist eine charakteristische oder bedingte Eigenschaft, die eine Funktion, Programm oder System unter Testbedingungen erfüllen sollte. Sie beschreibt dabei das zu erwartende Systemverhalten, ohne dazu spezifische Ausgabewerte für bestimmte Eingaben anzugeben. Properties bilden den zentralen Bestandteil von Property Based Testing. Anders als bei Example Based Tests werden keine fixen Parameter, sondern Regeln definiert, denen die Ausgaben des Systems entsprechen müssen. Dies geschieht unabhängig von den Eingabeparametern.

Die Abgrenzung, was genau eine Property definiert, ist dabei nicht immer exakt möglich. Das liegt darin begründet, dass eine logische Systemeinheit auch aus mehreren technischen Bedingungen bestehen kann. Eine Property kann, ähnlich wie ein Unit Test, aus mehr als einer Assertion bestehen. Die Definition von geeigneten Properties ist daher nicht immer eine triviale Angelegenheit, wenn das vollständige Spektrum von Eingabeparametern abgedeckt werden muss.

3.2.2 Generator

Der Generator ist für die Generierung von zufälligen Eingabeparametern zuständig. Die Zufälligkeit hängt dabei stark von der Implementierung ab. Sie können neben einfachen Datentypen, wie Zahlen oder Zeichenketten, auch komplexe Strukturen wie Arrays, Objekte oder rekursive Datenstrukturen generieren.

Die meisten Frameworks bringen von Haus aus bereits Implementierungen mit, um einfache Datentypen, wie Zahlen oder Zeichenketten, zu generieren. Manchmal stellen diese aber auch bereits Generatoren für komplexere Datentypen wie bspw. für Datum oder Zeitstempel zur Verfügung. In der Regel müssen für komplexe Datenobjekte jedoch eigenständige Generatoren implementiert werden.

Ein wesentlicher Aspekt ist die Geschwindigkeit eines Generators. Tests werden durch randomisierte Tests mit unzähligen unterschiedlichen Eingabeparametern ausgeführt. In Kombination mit weiteren Tests steigt die Zahl der Anwendung von Generatoren dementsprechend häufig an. Je performanter ein Generator also Eingabeparameter generiert, desto kürzer fallen die Testausführungszeiten aus. Dementsprechend häufiger können die Tests ausgeführt werden. Des Weiteren sollten Generatoren, je nach bereitgestelltem Zufallsgenerator, ein deterministisches Verhalten aufweisen. Das bedeutet, ein generierter Wert sollte auf der Zufälligkeit dieses Zufallszahlengenerators beruhen. Ist das nicht der Fall, weist der Generator kein deterministisches Verhalten nach. Ein fehlgeschlagener Test kann damit nicht reproduziert werden. [17]

Um eine Reproduzierbarkeit zu gewährleisten, werden Seeds verwendet. Ein Seed definiert den Startwert, der von einem Algorithmus zur Erzeugung von Pseudo-Zufallszahlen verwendet wird. Die Zahlen, die diese Algorithmen erzeugen, erscheinen zufällig, folgen jedoch einer mathematischen Formel. Der Seed definiert demzufolge den Anfangszustand eines Generators.

```
static void Main(string[] args)
{
    var random = new Random(40);
    Console.Write("1st sequence:\t");
    for (var i = 0; i < 10; i++)
    {
        Console.Write(random.Next(0, 10));
    }

    var anotherRandom = new Random(40);
    Console.Write("\n2nd sequence:\t");
    for (var i = 0; i < 10; i++)
    {
        Console.Write(anotherRandom.Next(0, 10));
    }
}
```

Listing 5: Erzeugung von Zufallszahlen identischer Reihenfolge

Das gezeigte Beispiel in Listing 5 demonstriert, wie der Seed mit der Random-Klasse in C# funktioniert. Beide Instanzen der Random-Klasse werden mit dem Seed 40 instanziiert und produzieren identische Zahlenfolgen. Dies erlaubt Entwicklern die Wiederherstellung der exakt gleichen Bedingungen, um damit Fehler reproduzierbar zu machen.

3.2.3 Shrinker

Da Eingabeparameter zufallsgeneriert sind, kommt es auch vor, dass komplexe Werte entstehen und für einen Test verwendet werden. Ein Test, der mit einem solchen Wert fehlschlägt ist mitunter schwer nachzuvollziehen. Daher stellt sich die Frage, ob ein identischer Test nicht auch mit einer einfacheren Kombination an Eingabeparametern fehlschlagen kann, die sich auf die wesentlichen, für den Fehlschlag verantwortlichen Informationen, bezieht.

Diese Frage versucht der Shrinker zu beantworten. Dabei handelt es sich um ein technisches Verfahren, welches versucht, die minimale Kombination von Eingabeparametern zu identifizieren, die einen Test fehlschlagen lassen. Fehler lassen sich dadurch wesentlich schneller identifizieren. Für einfache Datentypen ist die Logik, nach der ein Shrinker arbeitet, bereits in Generatoren vorhanden. Je nach Anwendungsszenario kann eine Implementierung jedoch eigene Logik notwendig machen.

3.3 Beispiel

Ein Beispiel ist eine Schnittstelle für Listen, die ausschließlich Unicode-UTF-16-Zeichen enthalten können. Die Liste besitzt die folgenden Methoden: Distinct, Sort, Count.

```
1  public List<char> GenerateListWithRandomLetters()
2  {
3      var list = new List<char>();
4      for (var  i = 0; i < Random.Next(); i++)
5      {
6          var character = GetRandomLetter();
7          list.Add(character);
8      }
9      return list;
10 }
```

Listing 6: Methode zum Generieren einer Liste

Wie eine Implementierung zur Generierung einer solchen Liste aussehen kann, zeigt das Listing 6. Die Methode erstellt eine Liste mit zufälligen Zeichen und gibt diese im Anschluss zurück. Die Länge der Liste ist dabei jedes Mal zufällig. Auf dieser Basis könnten folgende Properties für diese Liste als gültig erachtet werden:

Liste.Distinct().Count() == Liste.Distinct().Distinct().Count()

Liste.Count() == Liste.Sort().Count()

Die Methode „Distinct()“ wird verwendet, um doppelte Werte aus einer Liste zu entfernen. Damit wird sichergestellt, dass die in der Liste enthaltenen Werte alle unterscheidbar sind. Mit Hilfe der Methode „Count()“ kann die Anzahl der Werte in einer Liste ermittelt werden.

Dementsprechend kann die Annahme getroffen werden, dass ein mehrfaches Aufrufen der Methode „Distinct()“ zum gleichen Ergebnis führen muss. Die Erwartung ist, dass die Anzahl der Elemente in einer Liste gleich bleiben, unabhängig davon ob „Distinct()“ einmalig oder mehrfach aufgerufen wurde. Die Anzahl der Element sollte

sich auch im Fall einer Sortierung, wie in der zweiten Property beschrieben, nicht ändern. Hier findet lediglich eine Positionsänderung der Elemente innerhalb der Liste statt.

Sobald ein Test fehlt schlägt, versucht der Shrinker das Problem sukzessive einzugrenzen. In diesem Beispiel könnte der Shrinker diese Einschränkung durch Entfernen eines Elements und anschließender Testwiederholung erreichen. Listen tendieren beim Shrinken in Richtung einer leeren Liste.

3.4 Einordnung Teststufe

Üblicherweise werden Property Based Tests auf der Ebene von Unit Tests verortet. Dort spielen diese ihre Stärken aus. Durch ihre kurzen Ausführungszeiten ist der Einsatz auf dieser Ebene am effizientesten. Prinzipiell ist eine Einstufung innerhalb der Ebene von Integrations- oder Systemtests denkbar. Die Integration von PBT innerhalb dieser Testebenen stellt jedoch eine zusätzliche Herausforderung dar. So müssen externe Abhängigkeiten simuliert werden und dafür Sorge getragen werden, dass die Ausführungszeiten in einem akzeptablen Rahmen bleiben. Entwickler hadern oft mit der Definition von Spezifikationen, die das gesamte Systemverhalten betreffen [18].

3.5 Untersuchung von Zufallszahlengeneratoren

Die Generierung von zufälligen Werten ist ein wesentlicher Bestandteil, wenn es um Testautomatisierung im Allgemeinen und vor allem um die Nutzung von randomisierten Tests geht. Es existieren viele Algorithmen, die die Effizient und Qualität beeinflussen. Mit der Klasse „System.Random“ existiert in C# bereits eine Implementierung für die Generierung von Zufallszahlen. Dem gegenüber existieren jedoch noch weitere Algorithmen, wie zum Beispiel der PCG. Das Kapitel geht genauer auf beide Algorithmen ein und untersucht die Vorteile, sowie die Einschränkungen.

3.5.1 System.Random

Die Klasse „System.Random“ stellt bereits grundlegende Möglichkeiten zur Generierung von Pseudo-Zufallszahlen bereit. Prinzipiell findet die Klasse bereits Anwendung in vielen Bereichen wie Computerspielen, bis hin zu kryptografischen Anwendungen. Sie basiert auf einer modifizierten Version von Donald E. Knuths „subtractive random number generator algorithm“ [19].

Zwar wird die Klasse in vielen Bereichen eingesetzt, dennoch handelt es sich hierbei um einen Pseudo-Zufallszahlen-Generator. Generierte Zahlen sind demnach nicht tatsächlich zufällig. Es besteht die Möglichkeit, diese unter bestimmten Voraussetzungen vorherzusagen [20]. Ist diese Einschränkung bspw. in Fällen, wo kryptografisch sichere Zahlen verwendet werden sollen eher als Nachteil anzusehen, stellt sie im Kontext der Testautomatisierung jedoch einen wichtigen Punkt dar, um fehlgeschlagene Tests mit der gleichen Testmenge nochmals wiederholt überprüfen zu können.

3.5.2 Permuted Congruential Generator

Der Permuted Congruential Generator (PCG) ist ein fortschrittlicher Algorithmus zum Generieren von Pseudo-Zufallszahlen. Er ist bekannt für seine einfache Implementierung, schnelle Ausführung und statistischen Eigenschaften. Entwickelt wurde dieser von Melissa E. O'Neill und ist Teil einer Gruppe von Generatoren, die als „PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation" vorgestellt wurden. [21]

Die Idee dahinter ist die Nutzung von Eigenschaften von Kongruenzrelationen, um Generatoren zu erstellen, die eine leichte Implementierung ermöglichen. PCG sind weitaus performanter und zeichnen sich durch eine schnelle Ausführungszeit aus. Das macht sie besonders geeignet für Anwendungsbereiche, wo es auf hohe Leistung ankommt. Im Kontext von Testautomatisierung, im speziellen von Unit Tests, ist das ein wichtiger Punkt. [22]

4 Evaluierung von Tools

Das Kapitel untersucht unterschiedliche Bibliotheken, mit denen zufallsgenerierte Tests durchgeführt werden können. Für die Untersuchung werden unterschiedliche Kriterien definiert, die für die Betrachtung als wichtig angesehen werden. Am Schluss soll auf Basis der ermittelten Ergebnisse eine Bibliothek zur Auswahl stehen, auf der aufbauend im weiteren Verlauf eine Erweiterung implementiert wird.

4.1 Kriterien bei der Software-Evaluation

Die Kriterien, die für diesen Vergleich herangezogen werden, umfassen Aspekte wie die Funktionalität, Performance, Benutzerfreundlichkeit, Dokumentation, Community und Support. Im Folgenden wird beschrieben, aus welchen Gründen diese Kriterien für eine Evaluierung von Bedeutung sind.

4.1.1 Funktion und Performance

Bei der Auswahl einer Softwarebibliothek spielen Funktion und besonders die Performance eine entscheidende Rolle. Durch die Möglichkeit, hunderte Tests mit randomisierten Testdaten durchzuführen, muss der dafür eingesetzte Generator entsprechend performant arbeiten. Die Unterstützung einer Vielzahl von Datentypen spielt zudem eine wesentliche Rolle. Je mehr Datentypen unterstützt werden, desto breiter können verschiedene Testszenarien abgedeckt werden. Die Fähigkeit, sich nahtlos in bekannte Test-Frameworks wie NUnit oder xUnit zu integrieren, ist ebenfalls von großer Bedeutung. Dies ermöglicht die Bewertung, wie nahtlos sich randomisierte Tests in den bestehenden Testprozess integrieren lassen. Zudem ist die Erweiterbarkeit der Bibliothek ein wichtiger Faktor, der berücksichtigt werden muss. Da sich der Fokus auf die Programmiersprache C# und die objektorientierte Programmierung legt, sollte sich die Bibliothek in Projekte dieser Technologiestacks ohne enormen Aufwand integrieren lassen.

Um die Performance verschiedener Softwarebibliotheken zu untersuchen, wird ein Testfall erstellt, der einen identischen Ablauf für die zu vergleichenden Bibliotheken vorsieht. Dieser Testfall zielt darauf ab eine zufällige Zeichenkette zu generieren und diese zweimal umzukehren. Es wird erwartet, dass die Zeichenkette nach der zweimaligen Umkehrung identisch mit der ursprünglichen Zeichenkette ist. Ein besonderes Interesse liegt auf der Durchführungszeit, die für die Ausführung von 100 Tests

benötigt wird. Microsoft Visual Studio bietet mit dem integrierten Test Explorer eine praktische Möglichkeit diese Ausführungszeit zu messen. Um sicherzustellen, dass die Bedingungen für beide Bibliotheken gleich sind, wird eine Funktion zur Umkehrung einer Zeichenkette erstellt, die von beiden Bibliotheken für den Testfall angewendet wird.

```
1  public string ReverseString(string str)
2  {
3      var charArray = str.ToCharArray();
4      Array.Reverse(charArray);
5      return new string(charArray);
6  }
7
8  [Fact]
9  public void CsCheck_ReverseString_TestRoundTripping()
10 {
11     Gen.String.Sample(str =>
12     {
13         var otherString = ReverseString(ReverseString(str));
14         return str.Equals(otherString);
15     });
16 }
17
18 [Property]
19 public Property FsCheck_ReverseString_TestRoundTripping()
20 {
21     return Prop.ForAll(
22             Arb.Default.String().Filter(str => str != null),
23             str => str.Equals(ReverseString(ReverseString(str)))
24         );
25 }
```

Listing 7: Funktion zum Umkehren einer Zeichenkette und Tests mit FsCheck und CsCheck auf Round-tripping

Listing 7 zeigt eine mögliche Implementierung zur Umkehrung eines Strings. Der dazu gehörige Test untersucht die Annahme, dass durch die Umkehrung einer Zeichenkette mit Hilfe der Funktion und die anschließende Rückkonvertierung mit Hilfe der gleichen Funktion am Schluss der Originalwert zurückgeliefert wird.

4.1.2 Benutzerfreundlichkeit und Dokumentation

Ein wichtiger Faktor für Entwickler und Tester ist die Benutzerfreundlichkeit. Die einfache Handhabbarkeit einer Bibliothek ist ein Aspekt, der die Effizienz und Produktivität bei der Erstellung von Tests beeinflussen kann. Dazu gehört nicht nur eine klare Struktur, sodass die bereitgestellten Funktionen in ihren Aufgaben klar interpretierbar sind, sondern auch eine gute technische Dokumentation.

Eine gute technische Dokumentation gibt Aufschluss über die technischen Aspekte und Möglichkeiten der Bibliothek. Das sind typischerweise Informationen zur Architektur, Datenstrukturen, Algorithmen oder anderen technischen Details. Für einen leichteren Einstieg sind Tutorials vorhanden, die die grundlegenden Funktionen anhand einfacher Beispiele erklären. Typischerweise wird eine allgemeine Zusammenfassung der Bibliothek in Form einer Readme-Datei bereitgestellt, die zusammen mit dem Quellcode ausgeliefert wird, die einen Überblick über die bereitgestellten Funktionalitäten bereitstellt. [23]

4.1.3 Community und Support

Eine starke Community kann umfangreiches Wissen bereitstellen und Erfahrungsberichte austauschen, die in der von den Softwarebibliotheken bereitgestellten Dokumentation nicht behandelt werden. Eine aktive Community hilft darüber hinaus dabei schnell auf Probleme reagieren zu können und Lösungsansätze bzw. weiterführende Informationen liefern zu können. Dementsprechend kann diese auch als weiterführende Dokumentation interpretiert werden.

Die Aktivität einer Community lässt sich in Bezug auf unterschiedlichen Aspekten bewerten. Ein Aspekt kann die Untersuchung bekannter Plattformen sein, über die Nutzer Informationen bereitstellen und die als Anlaufstelle bei Fragen dienen kann. StackOverflow ist im Bereich der Softwareentwicklung eine dieser zentralen Anlaufstellen und bietet spezifische Möglichkeiten die Aktivität zu überprüfen. So gibt die Anzahl der gestellten Fragen, die mit einem bestimmten Tag versehen wurden, Aufschluss über das Volumen der Fragen. Antwortet ein Nutzer auf eine Frage, gilt diese als beantwortet, sobald eine der Antworten hochgestuft wurde oder eine Antwort vom Fragesteller als akzeptierte Antwort markiert wurde. Alle anderen Fragen gelten als unbeantwortet und geben einen Aufschluss, wie aktiv die Community ist. Die Aktualität der Fragen lässt Rückschlüsse auf Unklarheiten im Bezug auf die Bibliothek

schließen. Befindet sich der Zeitstempel der zuletzt gestellten Fragen weit in der Vergangenheit kann das darauf hindeuten, dass die Implementierung der Bibliothek wenig Spielraum für Unklarheiten liefert. Je nach Größe kann jedoch auch eine zu kleine Community eine Begründung sein. Abschließend können „Bountied Questions“, ein Indikator sein. Dabei werden Fragen mit einem „Kopfgeld“ versehen. Daran lässt sich erkennen welchen Wert die Community diesen Fragen beimisst. Eine höhere Anzahl deutet darauf hin, dass Fragen besonders wichtig oder herausfordernd sind.

Anhand der Informationen von GIT lassen sich Rückschlüsse über die Aktualität und den Grad der Unterstützung ableiten. Dazu zählen Informationen über den Zyklus, in dem Änderungen im Repository durchgeführt werden. Weitere Indikatoren für eine aktive Unterstützung können die Zahl der Sterne und Forks sein. Die Zahl der Sterne kann Aufschluss über die Popularität und Höhe des generellen Interesses der Community liefern. Ein Repository mit einem Stern zu markieren kann jedoch mehrere Gründe haben. So markieren Nutzer interessante Repositories auch, um sie für sich zu „bookmarken“, ohne es aktiv zu verwenden. Die Anzahl der Forks kann ein weiterer Indikator sein. Er gibt Aufschluss über die Anzahl der Nutzer, die es verwenden oder zur Weiterentwicklung beitragen. Die Gründe für einen Fork können jedoch vielfältig sein. Nutzer „forken“ Repositories bspw. auch aus Gründen des persönlichen Interesses oder um mit dem Quellcode zu experimentieren.

4.2 Tools

Für die Untersuchung eines geeigneten Tools stehen mit FsCheck und CsCheck zwei Bibliotheken zum zufallsgenerierten Testen zur Verfügung. Der Fokus bei der Untersuchung liegt neben der Performance auch auf der Erweiterbarkeit. Bekannte und oft verwendete Test-Frameworks wie xUnit oder NUnit liefern bereits Grundfunktionalitäten, mit denen zufallsgenerierte Testdaten erzeugt werden können. Diese beschränken sich jedoch auf wenige primitive Datentypen wie Integer oder Boolean. Auch lassen sich die Parameter im Fall von fehlgeschlagenen Tests auf den Wertebereich eingrenzen.

4.2.1 FsCheck

FsCheck ist eine in F# geschriebene Bibliothek zum automatisierten Testen von .NET-Applikationen. Entwickler erzeugen dazu Properties, die von Funktionen, Methoden oder Objekten geprüft und erfüllt werden müssen. Die Spezifikationen dieser Properties werden dabei in F#, C# oder VB definiert. Diese Properties werden von FsCheck im Anschluss gegen eine große Anzahl von zufallsgenerierten Werten getestet. Properties können darüber hinaus auch miteinander kombiniert werden. Schlägt eine Eigenschaft fehl, grenzt FsCheck den Fehler auf ein Minimum ein und zeigt dies an einem entsprechend minimalen Gegenbeispiel an. [24]

```
1  public int Addition(int x, int y)
2  {
3      return x + y;
4  }
5
6  [Fact]
7  public void Sample_Commutativity()
8  {
9      var property = (int x, int y) => Addition(x, y) == Addition(y, x);
10     Prop.ForAll(
11         Arb.From<int>(),
12         Arb.From<int>(),
13         Property
14     )
15     .QuickCheckThrowOnFailure();
16 }
```

Listing 8: Testfall unter Verwendung von FsCheck in xUnit zum Prüfen des Kommutativgesetz

Tests können in FsCheck autark ohne zusätzliche Tools erstellt und ausgeführt werden. Es lässt sich darüber hinaus aber auch in bekannte Test-Frameworks problemlos integrieren. Dazu stellt FsCheck Extensions für NUnit und xUnit bereit. Die Integration wurde durch andere Nutzer um weitere Test-Frameworks zusätzlich erweitert. Listing 8 zeigt einen Testfall, wie er in FsCheck definiert wird. Zur Ausführung der Tests wird xUnit verwendet. Durch die Annotation „[Fact]“ wird die Methode als Testfall markiert. Im Methodenkörper wird die Prüfung der Grundrechenoperation Addition auf das Kommutativgesetz geprüft.

```
[Property]
public bool TestCommutativity(int x, int y)
{
    return Addition(x, y) == Addition(y, x);
}
```

Listing 9: Testfall unter Verwendung von FsCheck in xUnit als Property

Der gezeigte Testfall lässt sich durch die Verwendung von Erweiterungen weiter vereinfachen. Listing 9 zeigt den gleichen Testfall. Durch das Ersetzen der Annotation „[Fact]“ durch „[Property]“, wird die Methode damit zur Property annotiert. Anders als bei einem Unit Test, bei dem kein Rückgabewert verwendet wird, wird die erzeugte Property von der Methode als Parameter übergeben. FsCheck generiert im Anschluss automatisch die Eingabeparameter des entsprechenden Typs. Dadurch werden Testfälle kompakter und beinhalten weniger Boilerplate-Code.

```
public static class StringGenerator
{
    public static Arbitrary<string> Generate()
    {
        return Arb.Default.String().Filter(str => str != null);
    }
}

[Property(Arbitrary = [typeof(StringGenerator)])]
public Property Test_StringReverse_with_FsCheck(string inputString)
{
    var doubleReversedString = ReverseString(ReverseString(inputString));
    var result = inputString.Equals(doubleReversedString);

    return result.ToProperty();
}
```

Listing 10: Testfall zur Prüfung von String-Umkehrung mit FsCheck

Der Testfall in Listing 10 untersucht die Funktion „ReverseString“ auf ihr Verhalten. Dazu wird der Testmethode ein Eingabeparameter in Form einer zufallsgenerierten Zeichenkette übergeben, auf die im Anschluss die Funktion „ReverseString“ zweimal angewendet wird. Die Erwartungshaltung ist, dass die zweifach umgekehrte Zeichenkette mit dem ursprünglichen Eingabeparameter übereinstimmt. Durch den in Listing

10 angepassten String-Generator wird sichergestellt, dass stets eine zufallsgenerierte Zeichenketten zurückgeliefert wird. FsCheck generiert per Standard Zeichenketten von unterschiedlicher Länge sowie in einem von zehn Fällen einen Null-Wert. Die Verwendung des angepassten Generators wird durch die Angabe innerhalb der Annotation „[Property]“ sichergestellt. Die Ausführungszeiten belaufen sich bei den Testläufen im Bereich zwischen 67ms und 69ms.

Für die Generierung von Testdaten stellt FsCheck bereits eine Vielzahl von Generatoren bereit. Darunter befinden sich neben primitiven Datentypen wie Bool, Byte, Int oder Float auch komplexere Datentypen wie DateTime, Strings, Lists, ein- und zweidimensionale Arrays, sowie Sets, Maps und Objekte. Außerdem lassen sich Funktionen von und zu diesen und zu den genannten Typen generieren. [25]

```
1 Message:
2  System.Exception : Falsifiable, after 1 test (1 shrink) (StdGen
3 (2136929152,297348547)):
4  ...
```

Listing 11: Fehlermeldung mit Angabe des verwendeten Seeds bei einem Testfehler

Schlägt ein Test fehl, ist es oft nützlich, wenn der gleiche Test nochmals mit den exakt gleichen Eingabeparametern ausgeführt werden kann. Wie in Listing 11 zu sehen, zeigt FsCheck hierfür die Seed des Pseudo-Zufallszahlengenerators an, sobald ein Test fehlgeschlagen ist.

```
1  [Fact]
2  public void FsCheck_Commutativity_WithSeed()
3  {
4      var property = (int x, int y)
5          => Addition(x, y) == Addition(y, x);
6
7      var config = Configuration.QuickThrowOnFailure;
8      config.Replay = FsCheck.Random.StdGen.NewStdGen(2136929152,
9  297348547);
10
11     Prop.ForAll(
12             Arb.From<int>(), // Parameter x
13             Arb.From<int>(), // Parameter y
14             property
15         )
16     .Check(config);
17 }
```

Listing 12: FsCheck Test mit gleicher Seed

Der zurückgelieferte Seed kann dann über ein angepasstes Konfigurationsobjekt dem Test übertragen werden. Das Beispiel in Listing 12 zeigt den identischen Test aus Listing 8 mit der Änderung, dass der Test in diesem Fall stets mit den gleichen Eingabeparametern durchgeführt wird. Anders als in Listing 8 wird der Test in diesem Fall nicht mit der Methode „QuickCheckThrowOnFailure()“, sondern mit der Methode „Check()“ ausgeführt. Diese erwartet ein Configuration-Objekt. Im Vorfeld kann diesem der angegebene Seed zugewiesen werden.

Die Dokumentation fällt im Vergleich zu anderen Bibliotheken in diesem Kontext vergleichsweise umfangreich aus. Sie bietet ein Tutorial, was den Einstieg und die Einarbeitung in die Funktionen der Bibliothek erleichtern. Außerdem werden die wichtigsten Aspekte, wie die Definition von Properties, die Generierung von Testdaten oder die Ausführung von Tests ausreichend erklärt. Unterstützt wird dies durch konkrete Beispiele und Quellcode. Für die Beispiele wird in den meisten Fällen F# als Sprache verwendet, vereinzelt finden sich jedoch auch Beispiele geschrieben in C#. Derzeit beschreibt die Dokumentation die Funktionen in der Version 2. Die aktuell entwickelte und gewartete Version ist jedoch Version 3. Dementsprechend ist die Dokumentation an manchen Stellen nicht vollständig. Die einzige Ausnahme bildet hier die API-Referenz. Deren Klassen und Namensräume sind vollständig beschrieben und erklärt. [25]

Um herauszufinden, wie aktiv die Community ist, wurden Analysen auf StackOverflow durchgeführt und Fragen ausgewertet. Um einen Anhaltspunkt zu erhalten, wurden Fragen einbezogen, die mit dem Schlüsselwort „fscheck“ markiert wurden. Zum Zeitpunkt der Analyse im Mai 2024 finden sich 151 Fragen auf StackOverflow, versehen mit dem erwähnten Schlüsselwort. Davon sind 19 Fragen als unbeantwortet markiert. Daraus lässt sich ableiten, dass die Nachfrage nach Lösungsansätzen relativ gering ausfällt. Die Mehrheit der Probleme ist damit bereits behandelt worden. Auf keine der gestellten Fragen wurde darüber hinaus ein Kopfgeld ausgesetzt. Die zuletzt gestellte Frage findet sich am 07.05.2023 wieder. [26]

Auf GIT wurde das Repository von 1100 Nutzern mit einem Stern versehen und es wurden 153 Forks erstellt. Die letzte Änderung wurde am 27.04.2024 eingecheckt [24]. Von NuGet wurde die Bibliothek ca. 1,5 Millionen mal heruntergeladen. Mit den Werten von StackOverflow, GIT und NuGet lässt sich belegen, dass, verglichen zu CsCheck, FsCheck mit Abstand die aktivere Community besitzt.

4.2.2 CsCheck

CsCheck ist eine C#-Bibliothek zum randomisierten Testen. Inspiriert wurde sie wie die meisten Bibliotheken von QuickQueck. Die Generierung von Testdaten, sowie das Shrinking, basiert, anders als FsCheck, auf dem Permutation Congruential Generator (PCG).

PCG ist eine Familie von Zufallszahlengeneratoren, die für ihre Einfachheit, Geschwindigkeit, Speichereffizienz und statistische Qualität hervorgehoben werden. Im Gegensatz zu vielen Allzweckgeneratoren sind PCG-Generatoren so gestaltet, dass sie eine hohe Unvorhersagbarkeit aufweisen. Diese Generatoren integrieren die Einfachheit und Geschwindigkeit von Linear Congruential Generators (LCG) mit der statistischen Qualität und der Unvorhersagbarkeit von Permutationsfunktionen. Anders als viele Allzweckgeneratoren, sind PCG-Generatoren so gestaltet, dass sie eine hohe Unvorhersagbarkeit aufweisen. Aufgrund dieser Kombination eignen sich PCG-Generatoren für eine Vielzahl von Anwendungen, von Simulationen bis hin zu Kryptographie, in denen eine hohe Sicherheit und eine qualitativ hochwertige Zufälligkeit erforderlich sind. [22]

Der Funktionsumfang von CsCheck erstreckt sich über die reine Durchführung von randomisierten Tests hinaus. Zusätzlich bietet das Tool die Möglichkeit, Model Based

Tests oder Metamorphic Tests durchzuführen. Im Kontext dieser Arbeit sind jedoch diese Erweiterungen nicht relevant. Die Automatisierung des Shrinking-Prozesses macht die Definition von Arbitrary-Klassen überflüssig. Dies führt zu einer Reduzierung des Boilerplate-Codes und verbessert die Übersichtlichkeit des Quellcodes erheblich. Durch die Zuweisung eines Seeds zu Testfällen wird deren Reproduzierbarkeit im Fehlerfall erleichtert. Es ist jedoch zu beachten, dass die Komplexität der Testfälle Grenzen für die Anwendbarkeit dieser Methode setzt. Ein zentraler Aspekt im Testprozess, insbesondere in Anwendungen mit Mehrbenutzerfähigkeit, ist das "Concurrency Testing". Dies bezieht sich auf das Testen in einem Szenario, in dem mehrere Benutzer gleichzeitig dieselben Aktionen durchführen. CsCheck ist so konzipiert, dass es nicht erforderlich ist die Tests wiederholt auszuführen. CsCheck unterstützt lediglich Standardtypen wie Strings, Integers, DateTime, sowie Arrays, Lists und Sets. [27]

```
1  private int Addition(int x, int y)
2  {
3      return x + y;
4  }
5
6  [Fact]
7  public void Test_Commutative_with_CsCheck()
8  {
9      var generator = Gen.Int;
10
11     Gen.Select(generator, generator)
12         .Sample((op1, op2) =>
13         {
14             var add = Addition(op1, op2);
15             var addReverse = Addition(op2, op1);
16             Assert.Equal(add, addReverse);
17         });
18 }
```

Listing 13: Testfall unter Verwendung von CsCheck in xUnit zum Prüfen des Kommutativgesetz

CsCheck ist so konzipiert, dass es in den Entwicklungsprozess ohne großen Aufwand integriert werden kann. Listing 13 zeigt einen Testfall zum Überprüfen, ob bei der Ausführung der mathematischen Operation Addition die Reihenfolge der zu addierenden Zahlen keinen Einfluss hat. Die Methode „Test_Commutative_with_CsCheck“ wird durch die Annotation „[Fact]“ als Testfall

markiert und wird fortan ausgeführt. Die Integration in Test-Frameworks, wie xUnit oder NUnit ist dadurch einfach und ohne großen Aufwand möglich. Das Repository bringt von Haus aus eine Vielzahl von verschiedenen Testbeispielen mit und bietet damit einen guten Einstieg. Abseits der bereitgestellten Beispieltests und der standardisierten Readme-Datei existiert keinerlei Dokumentation.

```
1  [Fact]
2  public void Test_StringReverse_with_CsCheck()
3  {
4      var property = (string inputString) =>
5      {
6          var doubleReversed = ReverseString(ReverseString(inputString));
7          Assert.Equal(inputString, doubleReversed);
8      };
9
10     Gen.String.Sample(property);
11 }
```

Listing 14: Testfall zur Prüfung der Zeichenkettenumkehrung mit CsCheck

Listing 14 zeigt einen Testfall zur Prüfung der Umkehrung einer Zeichenkette unter der Verwendung von CsCheck. Dazu wird die Property definiert, in der der Eingabeparameter zweimal umgekehrt wird. Im Anschluss wird eine Annahme in Form eines Asserts ausgeführt, in dem angenommen wird, dass ursprüngliche Eingabeparameter und der zweifach umgekehrte Wert identisch sind. Für die Generierung wird auf die Klasse Gen zurückgegriffen, die bereits eine Funktion zur Generierung einer zufallsgenerierten Zeichenkette bereitstellt. Die Ausführungszeiten belaufen sich bei den Testläufen im Bereich zwischen 6ms und 7ms.

StackOverflow liefert für die Bibliothek CsCheck keine plausiblen Zahlen, aus denen eine stichhaltige Aktivität innerhalb der Plattform analysiert werden könnte. Die Suche nach einem Schlüsselwort „cscheck“ resultiert in keinen Suchtreffern. Das GIT-Repository wurde von 121 Nutzern mit einem Stern versehen und es wurden drei Forks erstellt. Die bislang letzte Änderung wurde am 27.04.2024 eingecheckt [27].

4.3 Auswahl

Die Auswahl zwischen CsCheck und FsCheck für randomisierte Tests in C#-Projekten kann aufgrund verschiedener Faktoren schwierig einzuschätzen sein. FsCheck ist bekannter und wird in vielen Blogposts und Artikeln zum Thema randomisierter Tests als Bibliothek bevorzugt. Das bedeutet, es existieren eine größere Community und mehr Ressourcen, die bei der Findung von Lösungsansätzen bei vorhandenen Problemen helfen können. CsCheck hingegen bietet weniger Dokumentation und Ressourcen. Beide Bibliotheken, CsCheck und FsCheck, lassen sich in NUnit und xUnit Test-Frameworks integrieren. Durch die leichtere Integration in eine bestehende Test-Infrastruktur, wird dies als wichtiger Aspekt erachtet, wenn gleich auch dieser Punkt sowohl auf FsCheck, als auch CsCheck zutrifft. Die Unterstützung von Datentypen ist in beiden Bibliotheken identisch, was bedeutet, dass die Tests auf ähnliche Weise für verschiedene Datentypen geschrieben werden können.

Anthony Lloyd hat in seinem Repository verschiedene Random-Testing-Bibliotheken verglichen. Er hebt hervor, dass CsCheck im Vergleich zu anderen Bibliotheken, die eher dem ursprünglichen QuickCheck-Design treu bleiben, in der Lage ist, im Fall eines fehlgeschlagenen Tests die Eingabeparameter auf einen weitaus kleineren Wert zu schrumpfen, mit dem im Anschluss eine einfachere Fehleranalyse möglich ist. Dies deutet auf eine fortschrittlichere Test-Shrinking-Fähigkeit hin. [28]

CsCheck wurde in C# entwickelt, während FsCheck die funktionale Sprache F# verwendet. Die Entwicklung bzw. Erweiterung von FsCheck setzt dementsprechend Kenntnisse in F# und der funktionalen Programmierung voraus. Für Projekte, die in C# geschrieben sind, fällt CsCheck aufgrund seiner einfacheren Integration in C#-Projekte attraktiver aus. Dieser Punkt, sowie die deutlich bessere Performance bei der Ausführung von Tests, zielen schlussendlich auf CsCheck als Bibliothek ab, die für weitere Untersuchungen fokussiert wird.

5 Datentypen

In diesem Kapitel wird ein umfassender Überblick über verschiedene Datentypen und ihre spezifischen Eigenschaften gegeben. Der Fokus liegt dabei auf der Identifizierung der Rahmenbedingungen, die für jeden Datentyp charakteristisch sind. Diese Kenntnisse sind von Bedeutung, da sie die Grundlage für die Entwicklung von Generatoren bilden, die in der Lage sind, zufällig generierte Werte dieses Datentyps zu erzeugen. Ein wesentliches Ziel dieser Untersuchung ist es, sicherzustellen, dass alle möglichen Sonderfälle, die bei der Verwendung dieser Datentypen auftreten können, abgedeckt werden und eine effektive sowie robuste Implementierung der Generatoren gewährleisten.

E-Mails sind ein wesentliches Kommunikationsmittel und werden in vielen Bereichen eingesetzt. Sie werden häufig zur Authentifizierung genutzt, um sich gegenüber einer Anwendung oder einem Dienst zu identifizieren. Dies bezieht auch das Zurücksetzen eines Passwortes, indem zum Beispiel ein Link oder Code, mit dessen Hilfe ein Benutzer sein Passwort zurücksetzen kann, mit ein. Bildungs- und sozialfördernde Einrichtungen nutzen diese für administrative Mitteilungen, um über Lehrpläne, Kursangebote oder Veranstaltungen zu informieren. E-Mail-Adressen können in unterschiedlichen Formen und Strukturen auftreten. Im Folgenden geht das Kapitel näher auf die Formen einer E-Mail-Adresse ein und untersucht anhand der Spezifikationen, welche Formen gültig sind, um auf dieser Erkenntnis aufbauend einen Generator entwickeln zu können, der gültige E-Mail-Adressen erzeugen kann.

5.1 E-Mail

Die Spezifikation für E-Mail-Adressen legt fest, dass sie aus einer lokal interpretierbaren Zeichenkette und einer Internet-Domain bestehen. Voneinander getrennt werden beiden Teile durch die Angabe eines „@“-Zeichens (ASCII-Wert 64) [29, Sektion 3.4.1]. Die Syntax folgt dementsprechend dem Muster „lokalanteil@domainanteil“, wobei der Lokalanteil und Domäne jeweils bestimmte Längenbeschränkungen besitzen. Der Lokalanteil kann eine Länge von bis zu 64 Zeichen besitzen, während die Domäne einen vollständig qualifizierten Domainnamen (FQDN) repräsentieren muss und eine maximale Länge von 255 Zeichen erreichen kann. [30, Sektion 2-3]

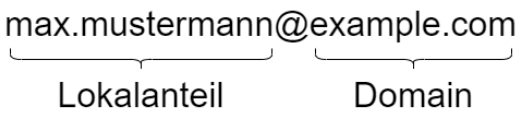

Abbildung 3: Gültigen E-Mail-Adresse

Zusätzlich können E-Mail-Adressen über einen Anzeigennamen verfügen, der in eckigen Klammern <> angegeben wird. Cyberkriminelle nutzen dies oft für „Display Name Spoofing“ oder „Email Address Spoofing“, um potentiellen Opfern einen falschen Namen vorzuspielen. Neben der heute gängigen Syntax existierten im Lauf der Zeit verschiedene andere Formen, wie der X.400 Standard und die UUCP Bang Path Notation, die jedoch durch den Internet-Standard, der von der Internet Engineering Task Force (IETF) beschlossen wurde, ersetzt.

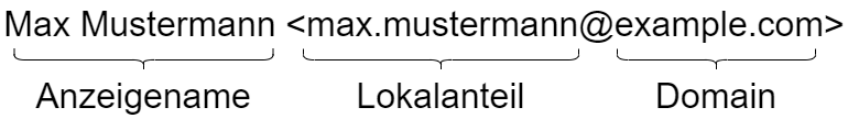

Abbildung 4: Gültigen E-Mail-Adresse mit Anzeigename

5.1.1 Lokalanteil

In RFC 2821 wird das Simple Mail Transfer Protocol (SMTP) detailliert beschrieben. Der lokale Anteil einer E-Mail-Adresse ist dafür konzipiert, nur von dem zu empfangenden Host interpretiert zu werden.

> *„Consequently, and due to a long history of problems when intermediate hosts have attempted to optimize transport by modifying them, the local-part MUST be interpreted and assigned semantics only by the host specified in the domain part of the address.“* (vgl. RFC2821, Sektion 2.3.10)

RFC 2822 beschreibt in der Sektion 3.4.1 die Spezifikation weiter und geht auf Details ein:

> *„An addr-spec is a specific Internet identifier that contains a locally interpreted string followed by the at-sign character (“@”, ASCII value 64) followed by an Internet domain. The locally interpreted string is either a quoted-string or a dot-atom.“* (vgl. RFC2822, Sektion 3.4.1)

Ein Punktatom (dot-atom) wird beschrieben als eine Reihe von Atomen (atoms), die jeweils mit einem Punkt von einander getrennt angegeben werden. Ein Atom besteht aus einer Reihe von alphanumerischen Zeichen und kann außerdem eine Reihe von Sonderzeichen enthalten. Der Punkt wird zur Verkettung mehrerer Atome verwendet und fließt demzufolge nicht in die Aufzählung der Sonderzeichen mit ein.

Erlaubte Zeichen sind alle ASCII-Zeichen mit Ausnahme der Sonderzeichen "(),:;<>@[\], sowie dem Leerzeichen und den ASCII-Steuerzeichen, wie zum Beispiel das Horizontaltabulator-Zeichen (ASCII-Zeichencode 9) [31, Sektion D]. Die Verwendung von Umlauten ist nicht erlaubt [31, Sektion 3.3]. Zu beachten gilt, dass der Punkt nicht das erste oder letzte Zeichen eines Atoms sein darf. Die Angabe von mehreren Punkten in fortlaufender Folge ist gleichermaßen nicht zulässig. Zu den Sonderzeichen gehören: !#$%&'*+-\/=?^_`{|}~ [30, Sektion 3].

Eine Erweiterung ermöglicht die Nutzung von internationalisierten Adressen, indem diese in UTF-8 kodiert werden. Dadurch wird der verwendbare Zeichensatz erweitert, sodass nicht nur Umlaute erlaubt sind, sondern auch Nicht-ASCII-Zeichen. Abseits dieser Erweiterung gelten weiterhin die gleichen Regeln für das Parsen, sowie die zu verwendenden Begrenzungszeichen [32, Sektion 3.5]. Die Erweiterung wird im RFC 6532 näher beschrieben. Zum jetzigen Zeitpunkt ist das Dokument als „proposed“ markiert und ist demzufolge eine vorgeschlagene Norm [33].

Alternativ kann der Lokalanteil auch in Anführungszeichen gefasst werden. Mit dieser Variante können zu den bereits genannten Zeichen auch noch Leerzeichen, sowie die Zeichen "(),:;<>@[\] verwendet werden. Befinden sich Backslash oder Anführungszeichen in der Zeichenkette, müssen diese mit einem Backslash maskiert werden. Ein in Anführungszeichen gesetzter Lokalanteil wird wie eine einzige Einheit behandelt. Semantisch gesehen handelt es sich um ein Atom, indem die beginnenden

und endenden Anführungszeichen dem Lokalanteil nicht als zugehörig gedeutet werden. Sie werden dennoch als Zeichen interpretiert und fließen damit in die Maximallänge des Lokalanteils mit ein [34, Sektion 3.2.4].

Im Vergleich zur Angabe mit Punktatomen sind E-Mail-Adressen, deren Lokalanteil in Anführungszeichen gesetzt wurden und mit einem Punkt beginnen oder enden, genauso gültig, wie auch E-Mail-Adressen mit mehrfach aufeinander folgenden Punkten. Listing 15 zeigt Beispiele für gültige E-Mail-Adressen, deren Lokalanteil in Anführungszeichen gesetzt wurde.

```
1 “.demo.user“@example.com
2 “demo.user.“@example.com
3 “demo..user“@example.com
4 “demo\“user“@example.com
5 “ “@example.com
6 “demo\\user“@example.com
```

Listing 15: Gültige E-Mail-Adressen unter Verwendung von Anführungszeichen

Obwohl ein in Anführungszeichen gesetzter Lokalanteil in der Praxis selten Verwendung findet, gibt es Gründe für deren Einsatz. Dabei ist jedoch zu beachten, dass nicht alle E-Mail-Clients und Server die Art der Angabe unterstützen [35, Sektion 2.3.10]. Dies kann dazu führen, dass Server und Clients beim Austauschen von E-Mails, aufgrund der unterschiedlichen Unterstützung, Probleme bekommen. Die Verarbeitung von E-Mails mit zitierten Adressen kann komplexer sein, da zusätzliche Logik erforderlich ist, um die Anführungszeichen und möglicherweise andere spezielle Zeichen zu erkennen und zu behandeln. Die Leistung kann dadurch beeinträchtigt werden, insbesondere wenn es sich um große Volumen an E-Mails handelt. Systeme sollten daher E-Mail-Adressen dieser Form nicht automatisch durch Filterregeln blockieren oder ablehnen, sondern diese an den zuständigen Zielhost zur Evaluierung weiterleiten [30, Sektion 3].

5.1.2 Domain

Im Kontext der E-Mail-Kommunikation muss der Domainteil einer E-Mail-Adresse strengen Richtlinien folgen. Die Domäne ermöglicht eine eindeutige Identifizierung des Mail-Servers, an den die E-Mail gesendet werden soll. Die Angabe erfolgt als „Fully Qualified Domain Name“ (FQDN). Sollte der Domainname nicht aufgelöst werden können, wodurch die Kommunikation einschließlich der Möglichkeit einen Fehlerbericht zu senden blockiert werden würde, kann dies durch die Angabe einer IP-Adresse umgangen werden. Die „Simple Mail Transfer Protocol“ (SMTP)-Spezifikation erlaubt die Angabe einer IP-Adresse, eingeschlossen in eckigen Klammern. Sowohl IPv4- als auch IPV6-Adressen sind für diese Zwecke zulässig. Es wird jedoch dringend davon abgeraten, IP-Adressen anstelle von FQDNs zu verwenden, es sei denn, es handelt sich um Test- oder Fehlerbehandlungszwecke. [30, Sektion 3], [36, Sektion 4.1.3]

Standardmäßig ist es nicht unüblich, dass Mail-Server E-Mail-Adressen ohne einen FQDN nicht akzeptieren. Relevant ist dies für Mail-Server, die mehrere Domains verwalten. Der FQDN in der E-Mail-Adresse ist erforderlich, um den Benutzer der Adresse der entsprechenden Domain zuordnen zu können. E-Mail-Adressen, die mit einer IP-Adresse gesendet werden, werden oft als Spam markiert. [30, Sektion 3]

```
1 demo.user@example.com
2 demo.user@localhost
3 demo.user@[172.12.8.22]
```

Listing 16: Beispiele für gültige E-Mail-Adressen

Die RFC 5321 legt in den Abschnitten 2.3.4 und 2.3.5 die Spezifikation für Host- und Domänennamen fest. Besonders hervorgehoben wird in Abschnitt 2.3.4 nochmals die Empfehlung, Hosts nicht durch numerische Adressen, sondern durch Namen zu identifizieren:

> *Hosts are known by names (see the next section) they SHOULD NOT be identified by numerical addresses, i.e., by address literals as described in Section 4.1.2.“* (vgl. RFC 5321, Sektion 2.3.4)

Ein Domainname setzt sich aus einem oder mehreren DNS Labels zusammen, die durch das Punkt-Zeichen miteinander verbunden werden, sofern mehrere Labels vorhanden sind. Bei der Verwendung einer Top-Level-Domain (TLD) ist auch eine einzelne Zeichenkette ohne Punktangabe zulässig. Diese Labels werden auch als LDH Label (Letter, Digit, Hyphen) bezeichnet. Die in diesen Labels erlaubten Zeichen sind eine Teilmenge des ASCII-Zeichensatzes, bestehend aus den Zeichen a bis z, A bis Z, den Ziffern 0 bis 9, sowie dem Minus-Zeichen. Labels dürfen nicht mit einem Minus-Zeichen beginnen oder enden.

Jedes der verwendeten Label ist auf eine maximale Länge von 63 Zeichen beschränkt und besteht ausschließlich aus alphanumerischen Zeichen, sowie Ziffern und dem Minus-Zeichen. Bei der Verwendung des Minus-Zeichens besteht eine zusätzliche Einschränkung insofern, dass dies weder das erste Zeichen noch das letzte Zeichen sein darf. [30, Sektion 2]

Internationalisierte Domain Names (IDN) erlauben neben der Verwendung des eingeschränkten ASCII-Zeichensatzes auch die Darstellung von Nicht-ASCII-Zeichen und werden im RFC 6532 näher beschrieben. Die Spezifikation befindet sich zum jetzigen Zeitpunk im Status „Proposed Standard“. Damit gilt der Standard generell als stabil und wurde durch die Community ausreichenden Reviews und Tests unterzogen. Änderungen am Inhalt der Spezifikation können jedoch nicht ausgeschlossen werden, sollten sich bessere Lösungen anbieten [37, Sektion 4.1.1].

5.2 IPv4

IP-Adressen bestehen aus zwei Teilen, einem Netzanteil und einem Hostanteil. Der Netzanteil spezifiziert dabei einen Bereich des Netzes, in welchem das Gerät enthalten ist, ähnlich wie es zum Beispiel in einem bestimmten Bereich einer Telefonnummer der Fall ist. Der Hostanteil wiederum kennzeichnet ein bestimmtest Gerät in diesem Netzsegment, ähnlich wie eine Telefonnummer ein bestimmtes Gerät innerhalb einer Ortsvorwahl identifiziert.

Adressen besitzen eine feste Länge von vier Oktetts, die mit einem Punkt miteinander verbunden werden. Jedes Oktett ist ein Ziffernblock mit einer Länge von acht Bit. Dementsprechend kann jeder Block einen Wert zwischen 0 und 255 annehmen. Eine IP-Adresse ist also 32 Bit lang. Damit sind ca. 4,3 Milliarden Adressen möglich. [38, Sektion 2.3]

Die Internet Engineering Task Force (IETF), sowie die Internet Assigned Numbers Authority (IANA) haben verschiedene Adressbereiche für speziale Verwendungszwecke reserviert [39, Sektion 4] [40, Sektion 2.2.2]. So sind zum Beispiel drei Adressbereiche für private Netzwerke reserviert. Diese beziehen sich auf eine Reihe von nutzbaren IP-Adressen, die nicht im Internet und ausschließlich von lokalen Netzen verwendet werden können. Diese Adressen sind nicht öffentlich „routbar“. [41, Sektion 3] [39, Sektion 4]

Adressblock	Adressbereich	Beschreibung/ Verwendung
0.0.0.0/8	0.0.0.0 - 0.255.255.255	aktuelles Netz
10.0.0.0/8	10.0.0.0 - 10.255.255.255	Privates Netz
127.0.0.0/8	127.0.0.0 - 127.255.255.255	Loopback
169.254.0.0/16	169.254.0.0 - 169.254.255.255	IPv4 Link-Local Adressen
172.16.0.0/12	172.16.0.0 - 172.31.255.255	Privates Netz
192.0.0.0/24	192.0.0.0 - 192.0.0.255	reserviert
192.0.2.0/24	192.0.2.0 - 192.0.2.255	Dokumentation (TEST-NET-1)
192.88.99.0/24	192.88.99.0 - 192.88.99.255	6to4-Anycast Weiterleitungspräfix
192.168.0.0/16	192.168.0.0 - 192.168.255.255	Privates Netz
198.18.0.0/15	198.18.0.0 - 198.19.255.255	Netz-Benchmark-Tests
198.51.100.0/24	198.51.100.0 - 198.51.100.255	Dokumentation (TEST-NET-1)
203.0.113.0/24	203.0.113.0 - 203.0.113.255	Dokumentation (TEST-NET-1)
240.0.0.0/4	233.252.0.0 - 233.252.0.255	reserviert
255.255.255.255/32	255.255.255.255	Broadcast

Tabelle 2: Reservierte IPv4-Adressbereiche

6 Realisierung

Im Kapitel wird die Funktionsweise von CsCheck dargelegt, einschließlich seines Aufbaus und der Methodik zur Definition von Testfällen. Anschließend wird auf die Herausforderungen eingegangen, die bei der Generierung von Testdaten unterschiedlicher Datentypen auftreten können.

6.1 Funktionsweise CsCheck

Für die Implementierung von Generatoren für die Bibliothek CsCheck ist zunächst ein Einblick in die Funktionsweise notwendig. Daraus resultiert auch die Entscheidung, in welcher Form die Bibliothek erweitert werden kann. Je nach Architektur gibt es unterschiedliche Möglichkeiten.

6.1.1 Aufbau Generator

Für die Generierung von randomisierten Testdaten ist zunächst wichtig zu verstehen, wie ein Generator aufgebaut ist und Testdaten generiert werden.

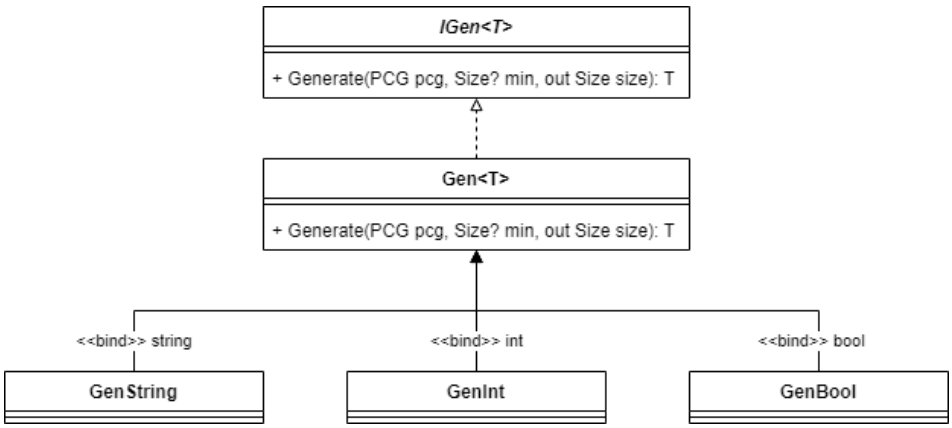

Abbildung 5: Struktur von Generatoren

Abbildung 5 zeigt die Darstellung und Abhängigkeiten eines Generators für einen bestimmten Datentyp. Jeder Generator leitet sich dabei von der abstrakten Klasse „Gen“ ab. Über den generischen Datentyp wird der konkrete Datentyp festgelegt, für den randomisierte Werte generiert werden sollen. Die Komplexität des Datentyps spielt dabei keine Rolle. Die Logik zum Generieren wird entsprechend in der Methode „Generate()“ implementiert.

```
1  public sealed class GenString : Gen<string>
2  {
3      static readonly Gen<string> d = Gen.Char.Array
4          .Select(i => new string(i));
5
6      public override string Generate(PCG pcg, Size? min, out Size size)
7          => d.Generate(pcg, min, out size);
8
9      // ...
10 }
```

Listing 17: Die Klasse GenString verwendet den Generator GenChar als Assoziation

Diese Klassen sind mit dem Schlüsselwort „sealed“ markiert. Damit sind diese Klassen für jede weitere Veränderung geschlossen. Eine als „sealed“ markierte Klasse kann nicht weiter abgeleitet werden und ist damit für jegliche Erweiterung geschlossen. Dieses Verhalten kann mit dem Schlüsselwort „final“ gleichgesetzt werden, mit dem in der Programmiersprache Java eine Klasse als geschlossen markiert werden kann.

6.1.2 Testfall

In Kapitel 4 wurde bereits auf CsCheck und dessen Funktion eingegangen. Testfälle lassen sich ohne Probleme in bestehende Test-Frameworks, wie zum Beispiel xUnit, integrieren. Testfälle werden auch unter Verwendung von CsCheck weiterhin mit den Annotationen „Fact“ und „Theory“ zu Testfällen deklariert. Im Kern bleibt die Handhabung dementsprechend identisch mit der von Example Based Tests. Damit kann die Verwendung auch ohne weiteres in bestehende Testsuites integriert werden.

```
1  [Fact]
2  public void Sort_DifferentExecutionSameResult()
3  {
4      int[] arr = [1, 2, 3];
5      int[] arr2 = [1, 2, 3];
6
7      arr = arr.Append(int.MinValue).ToArray();
8      Array.Sort(arr);
9
10     Array.Sort(arr2);
11     arr2 = arr2.Prepend(int.MinValue).ToArray();
12
13     Assert.Equal(arr, arr2);
14 }
```

Listing 18: Unterschiedliche Wege enden beim gleichen Ergebnis [42]

Ebenso bleibt die Struktur eines Unit Tests unverändert. Der in Listing 18 gezeigte Testfall zeigt ein weiteres Beispiel, bei dem geprüft wird, ob die Reihenfolge einen Einfluss auf die Ausführung hat. In diesem Fall wird geprüft, ob das Sortierverhalten eines Arrays auf beiden Wegen zum gleichen Ergebnis führt. Dabei muss das vorherige Anhängen des Integer-Minimalwertes und anschließende Sortieren, das identische Resultat zurückliefern, wie das Sortieren der Liste bevor im Anschluss der Integer-Minimalwert an den Beginn des Arrays gesetzt wird. Dieses Prinzip ist auch unter dem Namen „Kommutativität“ bekannt und wurde bereits im Listing 8 beim Beispiel der Addition angewendet. Dieser Prozess wiederholt sich bis die maximale Anzahl von Testdurchläufen erreicht ist. Im Umkehrschluss wird der Testfall als fehlgeschlagen gewertet, sollte sich herausstellen, dass beide Werte unterschiedlich sind. Der Test wird an dieser Stelle abgebrochen. Weitere Testdurchläufe finden nicht statt.

6.1.3 Unterstützte Datentypen

CsCheck unterstützt bereits eine Reihe von Datentypen für die Generierung von zufallsgenerierten Werten. Dazu zählen alle primitiven Datentypen wie Boolean, alle integralen numerischen Typen und Gleitkommatypen. Zusätzlich existieren bereits Generatoren für das Erstellen eines Zahlenwertes in einem bestimmten Wertebereich, der nicht einem der integralen numerischen Typen entspricht. Die Generator-Klasse „GenUInt16“ erstellt zum Beispiel Zahlenwerte im Bereich von 0 bis 15.

Des weiteren unterstützt CsCheck bereits die Generierung von einzelnen Zeichenketten (string), sowie einzelnen Zeichen (char), als auch die Erstellung von generierten eindeutigen IDs (GUID). Für die Generierung von Daten und Zeitstempeln sind außerdem verschiedene Generatoren vorhanden. Hier lassen sich neben einzelnen Zeitstempeln und Datums auch vollständige Zeitpunkte (Datum und Zeitstempel) erzeugen.

6.2 Generierung von Testdaten

Das Kapitel beschreibt die Implementierung der einzelnen Generatoren zu Datentypen, die in Kapitel 5 untersucht wurden.

6.2.1 IPv4

Die Implementierung eines Generators für IPv4-Adressen erscheint auf den ersten Blick nicht besonders komplex zu sein. Auf Basis der im Kapitel untersuchten Spezifikation ergeben sich folgende Richtlinien:

- der Rückgabewert muss vom Typ String oder IPAddress sein
- die Zeichenkette besteht aus Ziffern und drei Punkten
- die Zeichenkette besteht aus vier Ziffernblöcken im Bereich von 0 bis 255
- jeder Ziffernblock ist durch einen Punkt voneinander getrennt
- die minimale Länge der Zeichenkette beträgt sieben Zeichen (0.0.0.0)
- die maximale Länge der Zeichenkette beträgt 15 Zeichen (255.255.255.255)

Für die Umsetzung der Implementierung unter Einhaltung der ermittelten Richtlinien ergeben sich mehrere Möglichkeiten, eine IPv4-Adresse zu generieren.

```
public sealed class GenIPv4ByInt32 : Gen<string>
{
    static readonly Gen<uint> genUInt = Gen.UInt;
    public override string Generate(PCG pcg, Size? min, out Size size)
    {
        var integer = genUInt.Generate(pcg, min, out size);

        var octets = new uint[]
        {
            integer / (256 * 256 * 256),
            integer / (256 * 256) % 256,
            integer / 256 % 256,
            integer % 256
        };

        return string.Join('.', octets);
    }
}
```

Listing 19: Generator für IPv4-Adressen

Eine Möglichkeit bestünde in der Verwendung eines 32-Bit Integer. Dieser besitzt die gleiche Länge wie die einer IPv4-Adresse und kann demzufolge in vier 8-Bit Blöcke aufgeteilt werden. IPv4-Adressen enthalten keine negativen Werte. Dementsprechend wird für die Generierung des Ausgangsparameters für die Berechnung auf einen unsigned Integer als Datentyp zurückgegriffen. Die Formel zur Umrechnung eines Zahlenwertes in eine IP-Adresse basiert auf der Annahme, dass jedes der vier Oktetts als ein Faktor von 256 zum Ergebnis beiträgt.

```
public sealed class GenIPv4ByInt16 : Gen<string>
{
    static readonly Gen<uint> genUInt256 = Gen.UInt256;
    public override string Generate(PCG pcg, Size? min, out Size size)
    {
        var octets = new uint[]
        {
            genUInt256.Generate(pcg, min, out size),
            genUInt256.Generate(pcg, min, out size),
            genUInt256.Generate(pcg, min, out size),
            genUInt256.Generate(pcg, min, out size)
        };

        return string.Join('.', octets);
    }
}
```

Listing 20: Alternativer Weg zur Generierung zufälliger IPv4-Adressen

Eine andere Möglichkeit wäre für jedes einzelne Oktett einen eigenen zufälligen Wert zu erzeugen. CsCheck stellt bereits eine Implementierung für einen Generator zur Verfügung, welcher einen unsigned Integer in dem Bereich von 0 bis 255 erzeugt. Die in Listing 20 gezeigte Implementierung nutzt diesen Generator, um jeweils vier Werte innerhalb des gültigen Bereiches zu erzeugen. Weitere Berechnungen sind in diesem Fall nicht mehr notwendig.

Beide Implementierungen führen zum gleichen Ergebnis. Jedoch stellt sich die Frage, inwiefern die beiden Implementierungen Unterschiede in den Ausführungszeiten aufweisen. Property Based Tests werden, wie im Kapitel 3 untersucht, weitaus häufiger ausgeführt als Example Based Tests. CsCheck führt in seiner Standardeinstellung Tests mit 100 zufallsgenerierten Werten aus. Dementsprechend oft werden die Werte auch neu berechnet. Für einen Vergleich sind diese Werte zu gering, um repräsentative Ergebnisse zu erhalten. Aufgrund der sehr simplen Testfälle, der Test generiert lediglich eine zufällige IPv4-Adresse und gibt im Anschluss „true“ zurück, sind die Ausführungszeiten nahezu unmessbar gering.

Durchläufe	32-Bit Integer	4x 16-Bit Integer
100	6 ms	6 ms
1.000	6 ms	6 ms
10.000	8 ms	9 ms
100.000	19 ms	22 ms
1.000.000	120 ms	155 ms
10.000.000	848 ms	1,13 s

Tabelle 3: Ausführungszeiten unterschiedlicher IPv4 Implementierungen

Wie in Tabelle 3 erkennbar, gibt es deutliche Unterschiede, je häufiger zufallsgenerierte Werte für einen Test durchgeführt werden. Die Werte repräsentieren den Durchschnitt aus drei durchgeführten Testläufen. Während die Ausführungszeit bei zehn Millionen Durchläufen im ersten Test, in dem die Umrechnung auf Basis eines 32 Bit unsigned Integers angewendet wird, im Schnitt 848 Millisekunden in Anspruch nimmt, ist im zweiten Test ein deutlicher Anstieg bei der Berechnung, die auf der Verknüpfung von vier separat erstellten Werten basiert, zu erkennen. Der Unterschied

wird umso deutlicher, je mehr Iterationen durchgeführt werden. Daraus lässt sich schlussfolgern, dass die Operation zum Generieren mehrerer zufallsgenerierter Werte im Kontext der Performance weitaus kostenintensiver ist, als die Behandlung eines einzeln generierten Wertes.

```
1  static void Main(string[] args)
2  {
3      IPAddress ip;
4
5      // IPv4 with byte array => 107.70.178.215
6      ip = new IPAddress(new byte[] { 107, 70, 178, 215 });
7
8      // IPv4 with long => 82.65.26.0
9      ip = new IPAddress(1720658);
10
11     // IPv4 with string => 127.0.0.1
12     ip = IPAddress.Parse("127.0.0.1");
13 }
```

Listing 21: Unterschiedliche Möglichkeiten eine IPAddress zu konstruieren

Alternativ dazu existiert in C# mit der Klasse System.Net.IPAddress bereits eine Implementierung für eine IP-Adresse als Datentyp. Die Klasse besitzt mehrere Konstruktor-Überladungen und lässt sich bspw. mit Hilfe eines Byte-Arrays oder mit einem 64-Bit-Integer erstellen.

```
1  public sealed class GenIPv4Adress() : Gen<IPAddress>
2  {
3      static readonly Gen<uint> genUInt = Gen.UInt;
4
5      public override IPAddress Generate(PCG pcg, Size? min, out Size size)
6      {
7          var value = genUInt.Generate(pcg, min, out size);
8          return new IPAddress(value);
9      }
10 }
```

Listing 22: Implementierung eines Generator zur Erzeugung von Werten vom Typ IPAddress

Listing 22 zeigt eine mögliche Implementierung eines Generators, der einen zufällig generierten Wert vom Typ IPAddress zurückliefert. Der bei der Instanziierung übergebene Wert entspricht einem unsigned 32-Bit-Integer. Dadurch wird der Adressbereich auf den von IPv4 bekannten Bereich eingegrenzt. Die Klasse lässt sich zwar auch mit Werten außerhalb dieses Bereichs instanziieren, der Generator besitzt jedoch die Einschränkung auf Adressen innerhalb der Internet Protocol Version 4. Die Klasse verursacht beim Versuch sich selbst über die Methode „ToString()“ als String zurückzuliefern eine Exception.

```
1  static void Main(string[] args)
2  {
3      IPAddress ip;
4
5      // IPv4 0.0.0.0
6      ip = new IPAddress(0);
7
8      // IPv4 1.0.0.0
9      ip = new IPAddress(1);
10
11     // IPv4 255.255.255.255
12     ip = new IPAddress(4294967295);
13 }
```

Listing 23: System.Net.IPAddress Klasse in .NET

Im Vergleich zur Implementierung in Listing 19 handhabt die Klasse IPAddress die Berechnung des übergebenen Wertes auf anderem Weg. So entspricht der Wert 1 der Adresse „1.0.0.0“ und der Maximalwert eines 32-Bit-Integers der Adresse „255.255.255.255“. Hier stellt sich die Frage, ob sich die Ausführungszeiten im Vergleich zur Implementierung in Listing 19 unterscheiden. Dabei wird bei der Ausführung der Generierung als String, auf die Ausführungszeiten aus der Tabelle zurückgegriffen und diese im Anschluss mit den Ergebnissen aus der Generierung mit der bereits von .NET angebotenen Implementierung verglichen.

Durchläufe	32-Bit Integer	IPAddress
100	6 ms	5 ms
1.000	6 ms	5 ms
10.000	8 ms	6 ms
100.000	19 ms	11 ms
1.000.000	120 ms	52 ms
10.000.000	848 ms	479 ms

Tabelle 4: Ausführungszeiten Generierung von IPAddress

Die Testbedingungen sind in diesem Fall identisch. Die Ausführungszeiten repräsentieren auch in diesem Fall den Durchschnitt aus drei durchgeführten Testläufen. Die Ergebnisse zeigen, dass die Generierung noch einmal deutlich performanter ausfällt.

6.2.2 Domain

Domains sind vielseitig und finden in verschiedenen Bereichen Anwendung. Sie werden häufig zur Kennzeichnung von Diensten, die über das Internet angeboten werden, verwendet. Bekannte Vertreter dafür sind Webseiten oder E-Mail-Dienste. Domains identifizieren im Allgemeinen eine Netzwerkressource, wie zum Beispiel einen Personal Computer.

Domains besitzen eine Syntax, die unter anderem im RFC 1035 festgelegt ist. Im Kapitel 5.1.2 wurden die Voraussetzungen näher untersucht, die eingehalten werden müssen, damit es sich um eine gültige Domain handelt. Für die Generierung einer zufälligen Domain, die als gültig interpretiert werden kann, müssen eine Reihe von Richtlinien beachtet werden [43, Sektion 2.3.4]:

- Eine Domain ist eine Zeichenkette mit einer maximalen Länge von 255 Zeichen
- Eine Domain besteht aus einem oder mehreren Labels, die aus 63 Zeichen oder weniger bestehen
- Eine Domain besteht aus einem oder mehreren Labels, die durch einen Punkt „.“ miteinander verbunden sind

- Erlaubt sind LDH-Label, bestehend aus einer Teilmenge des ASCII-Zeichensatzes (a-z, A-Z, 0-9) und dem Minus-Zeichen
- Ein Label darf mit einem Minus-Zeichen weder starten noch enden
- Groß- und Kleinbuchstaben werden als gleichwertig erachtet

Die Zahl der Richtlinien, die eine Zeichenkette als gültige Domain verifizieren, zeigt, dass eine Umsetzung im Kontext von zufällig ausgewählten Zeichen eine aufwendigere Implementierung erfordert. Die möglichen Umsetzungen sind auch an dieser Stelle sehr variabel und vielfältig. Nach näherer Betrachtung der notwendigen Schritte kristallisieren sich zwei Varianten heraus, die weiter untersucht werden sollen. Dabei spielt die Performance wiederum eine entscheidende Rolle. Wie schon bereits im Kapitel 6.2.1 zur Generierung einer IPv4-Adresse erkannt, hat die Anzahl der zufällig generierten Werte eine direkten Einfluss darauf.

```
public sealed class GenDomainnameByConcatLabels : Gen<string>
{
    static readonly Gen<string> genLabel = Gen.String[
        Gen.Char[
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
            "abcdefghijklmnopqrstuvwxyz0123456789-"
        ], 1, 63];

    static readonly Gen<bool> genAnotherLabel = Gen.Bool;

    public override string Generate(PCG pcg, Size? min, out Size size)
    {
        var domainName = generateLabel(pcg, min, out _);
        var anotherLabel = genAnotherLabel.Generate(pcg, min, out size);

        while (anotherLabel)
        {
            var label = generateLabel(pcg, min, out size);
            bool maxLengthExceeded = domainName.Length + label.Length + 1
> 255;
            if (maxLengthExceeded)
            {
                break;
            }

            domainName = string.Join(".", label, domainName);
            anotherLabel = genAnotherLabel.Generate(pcg, min, out size);
        }
        return domainName;
    }

    private string GenerateLabel(PCG pcg, Size? min, out Size size)
    {
        string domainName = genLabel.Generate(pcg, min, out size);
        bool isNotValid = domainName.StartsWith('-')
                || domainName.EndsWith('-')
                || domainName.Contains("--");
        return isNotValid
            ? GenerateLabel(pcg, min, out size)
            : domainName;
    }
}
```

Listing 24: Generierung eines Domainnamens durch Verkettung zufällig generierter Labels

Eine Option zur Umsetzung wäre die Erzeugung eines Labels unter Einhaltung der Spezifikationen. D.h. die Gesamtlänge von 63 Zeichen darf nicht überschritten werden. Minus-Zeichen dürfen prinzipiell enthalten sein, jedoch dürfen sie weder am Beginn oder Ende positioniert sein. Die Methode „GenerateLabel()“ stellt bei der Generierung eines neuen Labels diese Bedingungen sicher. Auch eine Aneinanderreihung mehrerer Minus-Zeichen wird geprüft und das Label bei Vorhandensein verworfen.

Der Ablauf wird solange fortgesetzt, bis die Bedingungen für ein gültiges Label zutreffen.

Im Anschluss sollte entschieden werden, ob ein weiteres Label generiert und verkettet werden sollte. Die Entscheidung sollte auch an dieser Stelle zufällig erfolgen. Bei der weiteren Fortführung und Verkettung ist jedoch darauf zu achten, dass die maximale Länge der Domain nicht überschritten wird. Betrachtet man die Implementierung in Listing 24 stellt sich die Frage, in welchem Bereich sich die Zahl der Iterationen bezogen auf die Generierung zufällig generierter Werte befindet.

$$Label_{MaxAnzahl} = \frac{Domain_{MaxZeichen} - Label_{MinZeichen}}{Label_{MinZeichen} + Separator_{AnzahlZeichen}}$$

Formel 1: Berechnung maximale Label-Anzahl in Domains

Auf Basis der recherchierten Spezifikationen lässt sich die Anzahl der maximal möglichen Labels einer Domain mit der Formel 1 berechnen. Die Anzahl ist daher abhängig von der Länge der Domain. Da der Punkt als „Separator“ lediglich zwischen zwei Labels vorkommen darf, muss von der Länge der Domain die minimale Label-Länge einmalig subtrahiert werden. In der Theorie kann eine Domain nach dieser Formeln dementsprechend aus maximal 127 Labels bestehen (die Domain besitzt eine maximale Länge von 255, ein Label muss mindestens ein Zeichen lang sein). Im anderen Extrem kann eine Domain entsprechend aus einem einzigen Label bestehen.

Im Zuge der Implementierung wird der Grenzwert von 127 jedoch nur in der Theorie erreichbar sein. Vielmehr tendieren generierte Domains in die Richtung zwischen ein bis zwei Labels und nehmen in ihrer Wahrscheinlichkeit mehr als zwei Labels zu erhalten, mit jeder weiteren Iteration ab. Dabei gilt, dass je mehr Labels eine Domain enthalten soll, desto unwahrscheinlicher dieser Fall eintreten wird. Das liegt darin begründet, dass vor Generierung weiterer Labels eine Abfrage erfolgt, ob weitere Labels generiert werden sollen. Diese Abfrage wird durch einen zufällig erstellten Booleschen Wert realisiert. Ein solcher kann mit „true“ und „false“ zwei unterschiedliche Werte annehmen. Dementsprechend kann die Wahrscheinlichkeit, mit der eine bestimmte Anzahl von Labels erstellt wird, wie folgt berechnet werden:

$$P(A)=\frac{Fälle_{Anzahl}}{2\cdot Label_{Anzahl}+1}$$

Formel 2: Wahrscheinlichkeit mit der eine bestimmte Anzahl von Labels generiert wird

Nicht berücksichtigt wird an dieser Stelle die Tatsache, dass durch die Verkettung immer weiterer Labels die maximale Länge der Domain überschritten werden könnte.

```
1  public sealed class GenDomainnameByString : Gen<string>
2  {
3      static readonly Gen<string> genDomain = Gen.String[
4          Gen.Char[
5              "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
6              "abcdefghijklmnopqrstuvwxyz0123456789-"
7          ], 1, 255];
8
9      static readonly Gen<uint> genPosition = Gen.UInt64;
10
11     public override string Generate(PCG pcg, Size? min, out Size size)
12     {
13         var domainName = generateLabel(pcg, min, out size);
14         var currentPosition = genPosition.Generate(pcg, min, out size);
15
16         while (isValidPostion(domainName, currentPosition))
17         {
18             domainName = SetDot(domainName, currentPosition);
19             currentPosition += genPosition.Generate(pcg, min, out size);
20         }
21
22
23         return domainName;
24     }
25 }
```

Listing 25: Generierung einer Domain durch Ersetzen einzelner Zeichen

Eine weitere Option eine Domain zu generieren bestünde in der einmaligen Generierung einer Zeichenkette unter Einhaltung der Richtlinien. Eine zufällige Ersetzung bestimmter Zeichen durch einen Punkt, unterteilt die Domain im Anschluss in mehrere Labels. Das Ersetzen setzt im Vorfeld jedoch eine weitere Überprüfung voraus. Schließlich kann ein bestimmtes Zeichen in der Zeichenkette nur unter der Bedingung ersetzt werden, wenn das vorangestellte, sowie das folgende Zeichen weder einem Minus, noch einem Punkt entspricht. Ausgehend von der Implementierung in

Listing 25 lässt sich der Bereich, in dem sich die Anzahl von Generierungen befindet, auch in diesem Fall mit einer Formel ausrechnen:

$$Gen_{MinAnzahl} = Gen_{AnzahlDomain} + \frac{Domain_{Zeichen} - Label_{MinZeichen}}{Label_{MinZeichen} + Separator_{Zeichen}}$$

$$Gen_{MaxAnzahl} = Gen_{AnzahlDomain} + \frac{Domain_{Zeichen} - Label_{MaxZeichen}}{Label_{MaxZeichen} + Separator_{Zeichen}}$$

Formel 3: Berechnung minimaler und maximaler Anzahl von Generierungen

Die Anzahl der Generierungen ist folglich auch hier abhängig von der Länge der generierten Zeichenkette, die als Domain verwendet werden soll. Je länger die Zeichenkette ist, desto wahrscheinlicher sind weitere Iterationen, um diese in weitere Labels zu unterteilen. Dennoch lässt sich die Anzahl auf einen bestimmten Wertebereich eingrenzen. So sind bei einer Zeichenkette bestehend aus einem Zeichen maximal zwei Generierungen möglich (eine für die Zeichenkette und eine weitere um die Position für ein Punkt-Zeichen zu generieren). Auf der anderen Seite sind bei einer Zeichenkette mit 255 Zeichen maximal 128 Generierungen möglich. Diese Kennwerte stellen jedoch die Grenzwerte dar und sind in der Praxis nur schwer erreichbar.

Mit dieser Erkenntnis wäre interessant zu erfahren, welche der beiden Implementierungen im Schnitt die performantere Implementierung darstellt. Dazu werden beide mit unterschiedlich vielen Durchläufen getestet. Das Ergebnis wird durch die Berechnung des Durchschnitts aus drei Testdurchläufen errechnet.

Durchläufe	Label-Verknüpfung	String
100	7 ms	7 ms
1.000	9 ms	12 ms
10.000	27 ms	47 ms
100.000	190 ms	362 ms
1.000.000	912 ms	2,2 s
10.000.000	8,9 s	17,2 s

Tabelle 5: Ausführungszeiten unterschiedlicher Domain Implementierungen

Die Auswertung zeigt, dass die Performance in beiden Implementierungen bereits bei wenigen Durchläufen stark auseinander geht. Demnach ist die Erstellung einzelner Labels und deren Verknüpfung weitaus performanter und sollte bevorzugt verwendet werden.

6.2.3 E-Mail

Im Kern kann der Wert einer E-Mail-Adresse als Zeichenkette interpretiert werden, der sich aus zwei Teilen zusammensetzt, die mit Hilfe eines „@“-Zeichens mit einander verbunden werden. Wie bereits in Kapitel 5.1 untersucht, besteht der Domainteil entweder aus einer Domain oder einer IP-Adresse. Für beide Typen sind bereits im vorangegangenen Kapitel entsprechende Generatoren implementiert worden. Daher liegt der Fokus der Implementierung im Wesentlichen auf dem Lokalanteil. Für diesen gibt es eine Reihe von Richtlinien einzuhalten:

- Die Gesamtlänge beträgt höchstens 256 Zeichen
- Der Lokalanteil besteht aus 64 oder weniger Zeichen
- Für den Lokalanteil erlaubte Zeichen sind alphanumerische Zeichen inkl. der Sonderzeichen !#$%&'*+-\/=?^_`{|}~
- Das „.“-Zeichen darf verwendet werden, jedoch weder am Anfang noch am Ende stehen. Eine Angabe mehrerer Punkte in Folge ist nicht zulässig.

- Wird der Lokalanteil in Anführungszeichen gesetzt, entfallen die Sonderregelungen, die das „.“-Zeichen betreffen
- In Anführungszeichen gesetzt, sind die Sonderzeichen "(),:;<>@[\] und Leerzeichen zulässig
- Backslash und Anführungszeichen müssen innerhalb der Anführungszeichen durch ein Backslash maskiert werden

An dieser Stelle fällt auf, dass die Summe aus einem Lokalanteil mit 64 Zeichen Länge und einem Domainanteil von 255 Zeichen Länge zusammen mit dem „@“-Zeichen, 320 Zeichen entspricht. Eine solche E-Mail-Adresse ist laut den Spezifikationen ungültig. Das heißt, die Länge des Lokalanteils hat direkten Einfluss auf die maximale Länge des Domainteils. Eine Domain kann nach dieser Erkenntnis dementsprechend eine Länge zwischen 191 und 255 Zeichen annehmen, basierend auf der Länge des Lokalanteils. Da der implementierte Domain-Generator jedoch auf Basis der Spezifikation von Domains implementiert wurde, entsteht an dieser Stelle eine Diskrepanz. Es ist also darauf zu achten, dass der Wert einer generierten Domain nochmals validiert und dabei sichergestellt wird, dass es bei der Verwendung als Domainanteil eine Länge von über 255 Zeichen nicht erreicht wird.

Einen Einfluss auf die Verwendung einer IP-Adresse hat dies jedoch nicht. Eine IP-Adresse besteht im Extremfall aus 15 Zeichen. Selbst unter Einbeziehung der notwendigen Angabe von eckigen Klammern würde dies den maximalen Wert nicht überschreiten. Aus diesen Richtlinien lassen sich einige Sonderfälle ableiten, die im Listing 26 dargestellt sind. In allen Fällen handelt es sich um eine gültige E-Mail-Adresse.

```
1 x@example.com
2 user.name+tag+sorting@example.com
3 name/surname@example.com
4 admin@example
5 " "@example.org
6 "john..doe"@example.org
7 user-@example.org
```

Listing 26: E-Mail-Adressen können trotz unkonventionellem Lokalanteil gültig sein

Die Implementierung für diesen Generator besteht im Wesentlichen aus zwei Teilen. Die Generierung eines Wertes für den Lokalteil, sowie für die Generierung des Domainteils. Für den Lokalteil existieren laut den Untersuchungen in Kapitel 5.1.1 zwei Formen, die sich in ihrer Verhaltensweise unterscheiden. Zum einen gibt es eine Form, in der der Lokalteil in Anführungszeichen gesetzt wird. In dieser Form sind wesentlich mehr Zeichen zulässig. Darüber hinaus sind auch bspw. Zeichenfolgen bestehend aus mehreren aufeinanderfolgenden Punkten gültig. Die andere Form, in der der Lokalteil nicht in Anführungszeichen gesetzt wird, besitzt wesentlich striktere Richtlinien.

```
public sealed class GenEmail : Gen<string>
{
    static readonly Gen<string> genDomain = new
GenDomainnameByConcatLabels();
    static readonly Gen<string> genIpv4 = new GenIPv4ByInt32();
    static readonly Gen<bool> genBool = Gen.Bool;
    static readonly Gen<string> genLocalQuoted = Gen.String[1, 62];
    static readonly Gen<string> genLocal = Gen.String[
        Gen.Char["ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" +
            "0123456789!#$%&'*+-\\/=?^_`{|}~."], 1, 64];

    public override string Generate(PCG pcg, Size? min, out Size size) {
        var localPart = CreateLocalPart(pcg, min, out size);
        var domainPart =  CreateDomainPart(localPart, pcg, min, out size);

        return string.Join("@", localPart, domainPart);
    }

    private string CreateLocalPart(PCG pcg, Size? min, out Size size) {
        var shouldBeQuoted = genBool.Generate(pcg, min, out size);

        string? local;
        if (shouldBeQuoted) {
            local = genLocalQuoted.Generate(pcg, min, out size);
            // escape “ and quote local part
            local = local.Replace("\"", "\\\"");
            local = string.Format("\"{0}\"", local);
        } else {
            do {
                local = genLocal.Generate(pcg, min, out size);
            } while (IsInvalid(local));
        }
        return local;
    }

    private string CreateDomainPart(string localPart, PCG pcg, Size? min,
out Size size) {
        var shouldUseDomain = genBool.Generate(pcg, min, out size);

        string? domain;
        if (shouldUseDomain) {
            domain = genDomain.Generate(pcg, min, out size);
            if (domain.Length + localPart.Length + 1 + 1 > 256) {
                // sanitize domain
                var maxLen = 255 - localPart.Length;
                domain = domain.Substring(maxLen);
                // maybe after remove a part from the
                domain = domain.StartsWith(".") ? domain :
domain.Substring(1);
            }
        } else {
            domain = genIpv4.Generate(pcg, min, out size);
            // ip addresses should be set into brackets
            domain = string.Format("[{0}]", domain);
        }
        return domain;
    }
}
```

Listing 27: Implementierung eines Generators für zufällig E-Mail-Adressen

Beide Formen besitzen unterschiedliche Richtlinien. Dadurch entsteht an dieser Stelle eine Entscheidung, welche Form verwendet werden soll. Diese Entscheidung kann mit Hilfe eines zufällig generierten Booleschen Wertes umgesetzt werden. Für die Generierung des eigentlichen Lokalteils wird auf einen String-Generator zurückgegriffen, für den je nach gewählter Form eine Einschränkung in seinem Zeichensatz definiert wird. Unabhängig von der gewählten Form existieren für beide Varianten weitere Validierungen, um sicher zu stellen, dass die Zeichenkette den Richtlinien entspricht.

Für den Domainteil gibt es, wie auch schon beim Lokalteil, im Kern zwei unterschiedliche Varianten. Dieser kann eine Domain oder eine IPv4-Adresse darstellen. Dementsprechend wird an dieser Stelle ähnlich verfahren und die Wahl durch einen Booleschen Wert umgesetzt. In sehr seltenen Fällen kann es bei der Generierung von Domains dazu führen, dass der entstandene Wert vier Labels - bestehend aus Ziffern - enthält und damit eine IPv4-Adresse repräsentiert. In diesem Fall muss die generierte Domain als IPv4-Adresse interpretiert und entsprechend behandelt werden.

```
1 8@[0.0.0.6]
2 "s&Z#q)0_>2o$"@c6G-t6bMt.G2d
3 z?@ki1zAA1V.m2
```

Listing 28: Ergebnisse aus dem E-Mail-Generator

Die Ergebnisse aus Listing 28 zeigen, dass die generierten E-Mail-Adressen im Normalfall sehr kryptische Werte annehmen. Dennoch handelt es sich dabei laut der Spezifikation auf technischer Ebene um gültige Adressen.

6.3 Individualisierung von Zufallswerten

Die erzeugten Zufallswerte entsprechen zwar den Spezifikationen und decken den Gültigkeitsbereich ab, in der Praxis kommt es jedoch vor, dass nur bestimmte Wertebereiche nicht betrachtet werden sollen. So finden E-Mail-Adressen mit einer IP-Adresse als Domain kaum Anwendung und sollten auch laut der Spezifikation nur zu Testzwecken eingesetzt werden [36, Sektion 2.3.4]. Demzufolge würde eine Möglichkeit ausschließlich E-Mail-Adressen ohne IP-Adressen als Domain zu generieren, eine valide Option.

Damit ein Generator solche Bedingungen berücksichtigen kann, muss dieser in einer Form konfigurierbar gemacht werden. Je nach Generator können diese Bedingungen vielfältig sein. Für die Umsetzung bietet sich hierfür das Builder Pattern an. Dieses erlaubt die Erstellung eines einzelnen Objektes, welches viele unterschiedliche Konfigurationsparameter besitzen kann.

6.3.1 Ausschluss von IPv4-Adressebereichen

Bestimmte Bereiche lassen sich ausgrenzen, indem der generierte Ziffernwert geprüft wird, ob sich dieser innerhalb eines bestimmten Wertebereiches befindet. Dazu müssen die reservierten Adressbereiche im Vorfeld umgerechnet werden, um festzustellen, welche Zahlenbereich diese jeweils einnehmen.

Für die Ermittlung ist zuerst zu verstehen, wie die Klasse System.Net.IPAddress intern einen Wert vom Typ long in eine IP-Adresse umrechnet. Wie bereits in Listing 23 festgestellt, berechnet die Klasse für den Wert 1 die IPv4-Adresse „1.0.0.0“ und für den Maximalwert 4294967295 entsprechend die IPv4-Adresse „255.255.255.255“. Dem zugrunde liegend kann eine mögliche Umrechnung anhand der folgenden Funktion stattfinden.

```
1  public static uint IPAddressToUInt(IPAddress ip)
2  {
3      var octets = new byte[4];
4
5      ip.GetAddressBytes().Reverse().ToArray().CopyTo(octets, 0);
6      return BitConverter.ToUInt32(octets, 0);
7  }
```

Listing 29: Funktion zur Umrechnung eines IPAddress Objektes in einen unsigned 32-Bit-Integer

Ableitend von dieser Funktion können die Wertebereiche in die entsprechenden Ziffernblöcke umgewandelt werden. Da die reservierten Adressbereiche festgelegt sind, können die Zahlenbereiche in der Form – hartkodiert - in die Implementierung übernommen werden. Es wird davon ausgegangen, dass Änderungen in den reservierten Adressbereichen selten vorkommen, sodass eine Variable bzw. konfigurierbare Implementierung dieser Wertebereiche nicht als notwendig erachtet wird. Eine von

außen konfigurierbare Liste aller reservierten Adressbereiche würde eine Validierung dieser Einträge notwendig machen, wobei benutzerspezifischen Eingaben stets misstraut werden sollte.

```
1  public class IPv4GenOptions
2  {
3      public Dictionary<string, (uint min, uint max)> ExcludedRanges
4          { get; private set; } = [];
5
6      public IPv4GenOptions()
7      {
8          InitReservedBlocks();
9      }
10
11     public IPv4GenOptions IncludePrivateNetwork()
12     {
13         ExcludedRanges.Remove("PrivateNetwork#1");
14         ExcludedRanges.Remove("PrivateNetwork#2");
15         ExcludedRanges.Remove("PrivateNetwork#3");
16         return this;
17     }
18     ...
19 }
```

Listing 30: Option-Objekt zur Anpassung der erwartbaren IPv4-Adresswerte

Das Verhalten des Generators kann über die in Listing 30 gezeigte Implementierung beeinflusst werden. Ein solches Objekt vom Typ IPv4GenOption kann von einem IPv4-Generator aufgenommen und verarbeitet werden. Je nach gesetzten Parametern, berücksichtigt der Generator bestimmte Wertebereiche oder schließt diese aus.

Das Verhalten sieht dabei vor, dass ein zufälliger 32-Bit-Integer generiert und im Anschluss geprüft wird, ob dieser in einen Wertebereich fällt, der ausgeschlossen worden ist. Fällt die Ziffer in einen dieser ausgeschlossenen Bereiche, muss ein neuer Wert generiert werden. Dieser Vorgang kann sich entsprechend oft wiederholen. Prinzipiell besteht die Gefahr, dass der Vorgang ewig weiter läuft. Die Chancen, dass sich ein generierter Wert in einem der ausgeschlossenen Wertebereiche befindet, ist abhängig von der Größe dieses Wertebereiches. Werden alle Adressen ausgeschlossen, kommt eine erfolgreiche Generierung nie zustande.

```
public override IPAddress Generate(PCG pcg, Size? min, out Size size)
{
    int retryCounter = 0;
    uint value;
    do
    {
        if (retryCounter > Options.MaxGenRetries)
        {
            var message = "Maximum number of retries reached!";
            throw new GeneratorException(message);
        }

        value = genUInt.Generate(pcg, min, out size);
        retryCounter++;
    }
    while (IsInExcludedAddressArea(value));

    return new IPAddress(value);
}
```

Listing 31: Die Methode wirft eine Exception bei Überschreitung der Versuchsanzahl

Um dieses Problem zu beheben, wurde ein Counter implementiert, der ab einer bestimmten Anzahl von Versuchen eine Exception wirft. Die Grenze kann außerdem im Option-Objekt angepasst werden.

```
var options = new IPv4GenOptions();
options.IncludePrivateNetworks().IncludeLoopback();

var generator = new GenIPv4Adress(options);
generator.Sample(ip => ip.ToString().StartsWith("192.168."));
```

Listing 32: Instanziierung eines IPv4-Generators mit angepassten Konfiguration und Test

Die Standardeinstellung berücksichtigt bei der Generierung von IPv4-Adressen keine Adressen aus reservierten Adressblöcken. Das Option-Objekt ist dabei so aufgebaut, dass bestimmte Adressblöcke über einen Opt-in-Mechanismus einbezogen werden müssen. Soll es keine Einschränkungen bei der Generierung geben, kann dies durch die Methode „IncludeAll()“ im Option-Objekt dem Generator übermittelt werden.

6.3.2 Anpassung von E-Mail-Adressen

Die Konfiguration des Email-Generators ist nach dem gleichen Muster aufgebaut. Der Generator nimmt ein entsprechendes Option-Objekt entgegen. Durch das Option-Objekt lässt sich der generierte Wert beeinflussen. So lassen sich IP-Adressen als Domain ausgrenzen. Ein in Anführungszeichen gesetzter Lokalanteil kann zusätzlich ausgeschaltet werden.

```
1 var options = new EmailGenOptions();
2 options.AllowQuotedLocalPart().AllowIPv4();
3
4 var emailGenerator = new GenEmail(options);
5 emailGenerator.Sample(email => email.Length <= 255);
```

Listing 33: Instanziierung eines Email-Generators mit angepasster Konfiguration und Test

Die einzelnen Konfigurationsparameter im Option-Objekt besitzen bereits einen vorbelegten Wert, sodass sichergestellt wird, dass der Generator auch ohne übertragendes Option-Objekt an den entsprechenden Stellen reagieren kann. In diesem Fall wird ein Option-Objekt bei der Instanziierung des Generators aufgerufen und mit den vorbelegten Werten gearbeitet.

6.4 Generierung invalider E-Mail-Adressen

Für das Testen kann es auch durchaus sinnvoll sein, Implementierungen bewusst mit fehlerhaften Eingabeparametern zu prüfen, um deren Verhalten in diesem Fall zu prüfen. Schließlich ist es in der Praxis nicht unüblich, dass Benutzer bei der Eingabe Fehler verursachen. Dabei spielt es keine Rolle, ob diese bewusst oder unbewusst entstehen. Fehlerhafte Eingaben müssen in jedem Fall behandelt werden.

Bei dieser Problemstellung handelt es sich wiederum um ein Verhalten, wie der Generator seine Werte erzeugen soll. Dafür bietet es sich dementsprechend an, das Option-Objekt des E-Mail-Generators um eine zusätzliche Option zu erweitern. Sobald die Option aktiviert wird, führt der Generator einen weiteren Schritt aus. Nach dem dieser im ersten Schritt gültige E-Mail-Adressen erzeugt hat, werden diese in ei-

nem weiteren Schritt mit zusätzlichen Zeichen versehen. Im Kontext der Spezifikation repräsentieren diese keine gültigen E-Mail-Adressen.

```
1  public class EmailGenOptions
2  {
3      // ...
4      public bool PollutedEmailsEnabled { get; private set; } = false;
5
6      // ...
7      public EmailGenOptions PolluteEmails()
8      {
9          PollutedEmailsEnabled = true;
10         return this;
11     }
12 }
```

Listing 34: Mit der Option PolluteEmailsEnabled lassen sich ungültige Werte erzeugen

Eine wesentliche Frage stellt sich, wie diese „Verschmutzung“ von E-Mail-Adressen gehandhabt werden soll. E-Mail-Adressen können mit weiteren Zeichen versehen werden, um deren maximal gültige Länge zu überschreiten. Das gleiche gilt für einzelne Labels, sowohl im Lokal-, als auch Domainanteil. Das Hinzufügen von Minus-Zeichen am Beginn oder Ende, sowie das Aneinanderreihen mehrerer Punkte sind weitere Möglichkeiten. Leerzeichen, Anführungszeichen und Backslashes sind darüber hinaus nur gültig, wenn der Lokalanteil in Anführungszeichen gesetzt ist und diesem ein Backslash vorangestellt ist. Vergessene oder zu viele @-Zeichen können außerdem valide Gründe für eine ungültige E-Mail-Adresse darstellen.

```
1  private string CreatePollutedEmail(string local, string domain)
2  {
3      var caseNumber = local.Length % 4;
4
5      var localIsQuoted = local.StartsWith('"') && local.EndsWith('"');
6      var domainIsIpAddress = domain.StartsWith('[') &&
7  domain.EndsWith("]");
8
9      switch (caseNumber)
10     {
11         case 0:
12             local = OverExtendLocalPart(local, domain);
13             break;
14         case 1:
15             if (domainIsIpAddress)
16             {
17                 domain = OverExtendDomain(local, domain);
18             }
19             else
20             {
21                 local = OverExtendLocalPart(local, domain);
22             }
23             break;
24         case 2:
25             var c = local.Length % 2 == 0 ? '-' : '.';
26             local = localIsQuoted ? local + c : c + local;
27             break;
28         case 3:
29             local = AddSpecialChars(local);
30             break;
31     }
32     return CreateEmail(local, domain);
33 }
```

Listing 35: Methode zur "Verunreinigung" einer gültigen E-Mail-Adresse

Für die Implementierung in Listing 35 existieren vier unterschiedliche Szenarien, in denen die E-Mail-Adresse im Nachgang durch Ersetzungen und Anfügungen für ungültig deklariert wird:

- Fall 1: Der Lokalanteil wird auf eine Länge über die erlaubten 64 Zeichen hinaus verlängert.
- Fall 2: Die Gesamtlänge wird durch Erweiterung des Domainteils auf eine Länge über die erlaubten 255 Zeichen hinaus verlängert. Besteht der Domainteil aus einer IP-Adresse, wird stattdessen Fall 1 durchgeführt. Das Anfügen weiterer Zeichen an den Domainteil würde diesen für ungültig erklären.

- Fall 3: Der Lokalanteil wird entweder am Beginn bzw. am Ende um einen Punkt oder ein Minus-Zeichen erweitert. Damit ist der Lokalanteil ungültig, da beide Zeichen an diesen Stellen nicht existieren dürfen. Ist der Lokalanteil in Anführungszeichen gesetzt, wird zusätzlich die Richtlinie gebrochen, dass alle Zeichen nur innerhalb dieser stehen dürfen.
- Fall 4: Bestimmte Zeichen innerhalb des Lokalanteils werden durch Leerzeichen, Backslash und Anführungszeichen ersetzt. Die Position ist dabei abhängig von der Länge des Lokalanteils. Ist der Lokalanteil in Anführungszeichen gesetzt, wird die Richtlinie nicht gebrochen, sodass in diesem Fall der Lokalanteil sowohl am Beginn, als auch am Ende durch ein zusätzliches Label ergänzt wird.

Um eine gewisse Zufälligkeit in den Fällen zu gewährleisten, wird die Länge des Lokalanteils durch die Maximalanzahl der Fälle geteilt. Der entstandene Rest entscheidet im Anschluss, auf welche Art und Weise die E-Mail-Adresse bearbeitet werden soll. Da die Generierung des Lokalanteils bereits zufallsbasiert erfolgt, ist eine weitere Generierung von Zufallswerten an dieser Stelle nicht notwendig.

Die Switch-Anweisung stellt an dieser Stelle ein Problem dar. Diese bedingte Verzweigung ist ein Indiz dafür, dass die Klasse zu viele unterschiedliche Verhaltensmuster besitzt. Schließlich muss die Klasse angepasst werden, wenn ein weiterer Fall hinzugefügt wird. Eine Möglichkeit die Switch-Anweisung aufzulösen, wäre mit dem Einsatz des Strategy-Patterns. Damit ließen sich die unterschiedlichen Fälle in separaten Klassen kapseln [9, S. 231-232]. Welche Fälle bei der Manipulation der E-Mail-Adressen schlussendlich verwendet werden sollen, wäre bspw. auch über das Option-Objekt denkbar.

6.5 Builder und Fluent Interface

Das Instanziieren eines Generators kann je nach Umfang des Option-Objektes eine hohe Komplexität annehmen. Das Problem lässt sich bspw. mit dem Builder Pattern lösen. Ziel dieses Musters ist die Erstellung von komplexen Objekten durch Aufteilung dieses Prozesses in kleinere Schritte zu vereinfachen. [44, S. 97-98]

Mit dem Entwurfsmuster kommt in der Regel das Konzept der Fluent Interfaces einher, mit dem die Verkettung mehrere Methoden auf eine Art und Weise umgesetzt wird, sodass bei der Aneinanderreihung die Form von Sätzen in natürlicher Sprache entsteht. Damit lässt sich der geschriebene Quellcode auch wesentlich einfacher verstehen.

```
1  public interface IGenBuilder<T>
2  {
3      public T Build();
4  }
5
6  public interface IIPv4GenBuilder : IGenBuilder<GenIPv4Address>
7  {
8      public IIPv4GenBuilder IncludePrivateNetwork();
9      public IIPv4GenBuilder IncludeLoopback();
10
11     // ...
12 }
13
14 public class IPv4GenBuilder : IIPv4GenBuilder
15 {
16     private IPv4GenOptions Options { get; init; } = new IPv4GenOptions();
17
18     public GenIPv4Address Build()
19     {
20         return new GenIPv4Address(Options);
21     }
22
23     public IIPv4GenBuilder IncludeLoopback()
24     {
25         Options.IncludeLoopback();
26         return this;
27     }
28
29     public IIPv4GenBuilder IncludePrivateNetwork()
30     {
31         Options.IncludePrivateNetwork();
32         return this;
33     }
34
35     // ..
36 }
37
38 public class GenBuilder
39 {
40     public static IIPv4GenBuilder IPv4 => new IPv4GenBuilder();
41
42     // ...
43 }
```

Listing 36: Implementierung eines Builders zur Erzeugung eines IPv4-Geneators

Da jeder Datentyp unterschiedliche Merkmale aufweist, besitzt jeder Datentyp dementsprechend seinen eigenen Builder. Davon ausgehend dient die Klasse „GenBuilder“ als Ausgangspunkt für die Erzeugung eines Generators. Die Builder sind dabei so aufgebaut, dass diese das Interface der konkreten Implementierung zurückliefern. Die eigentliche Implementierung bleibt somit verdeckt. Entwickler können auf diese Weise lediglich die Methoden aufrufen, die vom jeweiligen Interface zur Verfügung gestellt werden. Der Builder selbst enthält ein Option-Objekt, dessen Werte mit Hilfe der Methoden des jeweiligen Builder-Interfaces geändert werden können. Durch die „Build“-Methode wird schlussendlich eine Instanz des Generators erzeugt und das Option-Objekt übergeben. Das Erzeugen des Option-Objektes innerhalb der konkreten Implementierung sorgt dafür, dass Änderungen nur über die öffentlichen Methoden erlaubt werden. Die Konfiguration ist damit vollständig gekapselt und bleibt unveränderbar, sobald der Generator mit der „Build“-Methode erzeugt wurde.

```
1  GenBuilder.IPv4.IncludePrivateNetwork()
2        .IncludeLoopback()
3        .Build()
4        .Sample(ip => ip.ToString().StartsWith("192.168."));
5
6  GenBuilder.Email.AllowQuotedLocalPart()
7        .AllowIPv4()
8        .Build()
9        .Sample(email => email.Length <= 255);
```

Listing 37: Erzeugung von Generatoren mit Hilfe des GenBuilders

Listing 37 zeigt, wie ein Generator mit Hilfe des Builders erzeugt werden kann. Die Klasse GenBuilder fungiert an dieser Stelle als Ausgangspunkt. Die in den Listings 22 und 27 instanziierten E-Mail- und IPv4-Generatoren lassen sich auf die gleiche Weise auch mit dem GenBuilder erzeugen. Durch das Zentralisieren der Objekterzeugungslogik im Builder, sowie die Nutzung des Fluent Interface Konzeptes, lässt sich der Quellcode besser organisieren und steigert die Lesbarkeit. Die Erzeugung und Rückgabe des Generators durch die „Build“-Methode sorgt darüber hinaus dafür, dass direkt im Anschluss mit dem Generator weiter gearbeitet werden kann. In den

im Listing 37 gezeigten Beispielen wird nach der Erzeugung direkt ein Test durchgeführt.

6.6 Implementierungserkenntnisse

Die Implementierungen haben gezeigt, dass bei der Implementierung von Generatoren der Fokus nicht nur auf das generierte Ergebnis gelegt werden sollte, sondern auch die Ausführungszeiten im Blick zu behalten sind. Durch die höheren Testdurchlaufzahlen bei randomisierten Tests summieren sich selbst kleine Unterschiede in den Testausführungszeiten sehr schnell auf. Schlechte Ausführungszeiten können die Codequalität direkt beeinflussen. Als Konsequenz wird damit die Wahrscheinlichkeit von Fehlern erhöht.

Dass die Ausführungszeiten je nach Implementierung stark voneinander abweichen können, haben Messungen unterschiedlicher Implementierungen gezeigt. Dabei wurde ein Zusammenhang zwischen der Anzahl der generierten Zufallswerte und der Ausführungszeiten erkannt. Je mehr Entscheidungen bei der Generierung eines Datentyps getroffen werden müssen, desto mehr Zufallsgenerierungen müssen möglicherweise durchgeführt werden. Die Komplexität erhöht sich in diesem Zusammenhang mit der Zahl der Richtlinien, die für die Erstellung eines Datentyps eingehalten werden müssen. Zu den Richtlinien zählen dabei nicht nur durch Spezifikationen definierte Regeln, sondern auch Konfigurationsparameter, die das Verhalten eines Generators steuern.

Bei den Recherchen hatte sich herausgestellt, dass sich Informationen in dem RFC 3696 als fehlerhaft erwiesen haben.

> *„In addition to restrictions on syntax, there is a length limit on email addresses. That limit is a maximum of 64 characters (octets) in the "local part" (before the "@") and a maximum of 255 characters (octets) in the domain part (after the "@") for a total length of 320 characters. Systems that handle email should be prepared to process addresses which are that long, even though they are rarely encountered.“* (vgl. RFC3696, Sektion 3)

Der RFC beschreibt E-Mail-Adressen mit einer maximalen Länge von 320 Zeichen. Jedoch existiert eine Beschränkung in RFC 2821, wonach E-Mail-Adressen, die nicht in MAIL oder RCPT passen, als wenig nützlich erachtet werden [35, Sektion 4.5.3.1]. Die Obergrenze sollte deshalb normalerweise bei 256 Zeichen liegen. Zudem sind drei der im RFC aufgelisteten E-Mail-Adressen, die als gültige Variante vorgeschlagen werden, keine gültigen Varianten [45].

7 Testdurchführung

Die Implementierung muss auch auf Funktionalität geprüft werden, sodass sicher gestellt wird, dass die generierten Zufallswerte auch den Spezifikationen entsprechen. Das Kapitel untersucht die Fragestellung, wie dieses Problem gelöst werden kann. Prinzipiell ergeben sich dafür zwei unterschiedliche Wege. Zum einen ist die Verwendung bereits existenter Bibliotheken zur Validierung eine Möglichkeit. Zum anderen bietet CsCheck bereits Erweiterungen für bekannte Test-Frameworks, mit denen zufallsgeneriertes Testen durchgeführt werden kann. Dafür ist die Definition von Properties notwendig.

7.1 Prüfung durch CsCheck

Die Prüfung mit Hilfe von CsCheck selbst ist eine Möglichkeit, da es sich hierbei bereits um eine Bibliothek zum zufallsgenerierten Testen handelt. Mit „Sample()“ stellen Generatoren bereits eine Methode bereit. Diese nimmt eine Reihe von Parameter entgegen. Der erste ist dabei eine Action, die als generischer Typ den jeweiligen generierten Datentyp aufnimmt. Diese stellt die Assertion dar und liefert im Erfolgsfall „true“ zurück. Schlägt die Annahme fehl, so wird eine Ausnahme ausgelöst und die Testdurchführung wird abgebrochen. Der Test gilt dann als fehlgeschlagen.

Die Methode „Sample()“ kann außerdem mit weiteren Parameter definiert werden. So ist die Angabe eines Seeds möglich, um einen fehlgeschlagenen Test im Zweifelsfall noch einmal mit den gleichen zufällig erzeugten Werten wiederholen zu können. Der Test lässt sich so reproduzieren. Die Zahl der Durchläufe eines Tests lässt sich zudem ändern. Ohne Angabe wird ein Test standardmäßig 100 Mal durchgeführt. Mit jedem dieser Durchläufe wird auch ein neuer zufälliger Wert berechnet.

```
1  public class EmailGenTests
2  {
3      [Fact]
4      public void LocalPart_Max_Length_Is_Less_Or_Equal_64()
5      {
6          var gen = new GenEmail();
7          gen.Sample(email => GetLocalPart(email).Length <= 64);
8      }
9
10     [Fact]
11     public void
12 Unquoted_LocalPart_Consists_Of_Alphanumerical_And_SpecialChars()
13     {
14         var allowCharacters = "!#$%&'*+-\\/=?^_`{|}~";
15         var gen = new GenEmail();
16         gen.Sample(email => CharIsValid(GetLocalPart(email),
17 allowCharacters));
18     }
19     // ...
20 }
```

Listing 38: Tests des E-Mail-Generators

Der Aufbau der Teststruktur sollte so realisiert werden, dass sich Tests problemlos organisieren lassen. Jeder Generator besitzt dazu seine eigene Testklasse, für den spezifische Tests geschrieben werden. Jede Testklasse stellt damit eine Testsuite dar. Jede Methode innerhalb dieser Testklasse repräsentiert dementsprechend, sofern diese mit einer Annotation „Fact“ versehen wurde, einen Test. Ziel dieser Tests soll es sein, das zufallsgenerierte Werte ihren Spezifikationen entsprechen und der Generator demnach seinem Standardverhalten entspricht. Außerdem muss mit den Tests sichergestellt sein, dass die erzeugten Werte, durch Änderungen an den Konfigurationsparametern des Generators, den Erwartungen entsprechen.

Auf Basis der in Kapitel 5 untersuchten Spezifikationen und den daraus abgeleiteten Richtlinien besteht ein wesentlicher Punkt für die Prüfung darin, diese gezielt in einzelnen Tests auf Einhaltung zu prüfen. Das erleichtert die Fehlerfindung, wenn ein Test fehlgeschlagen ist. Tests werden damit auch wesentlich schlanker im Umfang des Quellcodes.

```
1  public class EmailGenTests
2  {
3      [Fact]
4      public void Email_Contains_At_Least_One_At_Sign()
5      {
6          var regex = new Regex("^.*[@].*$");
7          GenBuilder.Email.Build().Sample(regex.IsMatch);
8      }
9
10     [Fact]
11     public void Max_Length_Is_Less_Then_256()
12     {
13         GenBuilder.Email.Build().Sample(email => email.Length < 256);
14     }
15 }
```

Listing 39: Testfälle unter Verwendung von RegEx und Properties

Wie eine Prüfung aussehen kann, verdeutlicht das Beispiel in Listing 39. Die aufgestellte Annahme im ersten Test „Email_Contains_At_Least_One_At_Sign()“ ist, dass die generierte E-Mail-Adresse wenigstens ein „@“-Zeichen enthalten muss. Für die Umsetzung dieser Annahme wird auf die Prüfung mittels eines regulären Ausdruckes zurückgegriffen. Durch die Aufteilung der Richtlinien in unterschiedliche Tests bleiben diese klein und übersichtlich.

Dementsprechend können auch die notwendigen regulären Ausdrücke sehr einfach gehalten werden. Auch bei einem regulären Ausdruck muss in irgendeiner Form sichergestellt werden, dass dieser den Erwartungen entsprechend arbeitet. Je komplexer dieser daher ausfällt, desto größer besteht das Risiko, dass durch den Ausdruck geprüfte E-Mail-Adressen nicht dem erwarteten Ergebnis entsprechen.

Anstelle eines regulären Ausdrucks können Eingabedaten aber auch mithilfe spezifischer String-Methoden überprüft werden. Dies kann durch die Prüfung auf das Vorhandensein verschiedener Teile eines Strings an unterschiedlichen Positionen erreicht werden. Beispielsweise könnten Methoden wie „StartsWith()“, „EndsWith()“ oder „Contains()“ verwendet werden, um zu ermitteln, ob der gegebene String mit einem bestimmten Muster übereinstimmt oder bestimmte Zeichenketten enthält.

```
Fact]
public void LocalPart_Do_Not_Start_Or_End_With_Dot_Sign()
{
    GenBuilder.Email
        .AllowQuotedLocalPart()
        .Build()
        .Sample(email =>
        {
            var local = GetLocalPart(email);
            return !local.StartsWith(".") || !local.EndsWith(".");
        });
}
```

Listing 40: Test mit Prüfung durch String-Funktionen

Eine Prüfung mit Hilfe von String-Funktionen wie in Listing 40 zu sehen, beinhaltet eine Fehleranfälligkeit, wenn bspw. das Ausrufezeichen versehentlich vergessen wurde und damit der Ausdruck eine andere Bedeutung erhält. Im Gegenzug haben reguläre Ausdrücke stets das Problem, dass diese schlecht lesbar sein können.

```
^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9]
[0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|
[01]?[0-9][0-9]?)$
```

Listing 41: Regulärer Ausdruck zur Prüfung einer IPv4-Adresse

Die Verwendung eines regulären Ausdrucks wie in Listing 41 bringt demzufolge gleichermaßen Probleme mit sich. Die genaue Funktion des Ausdrucks lässt sich je nach Komplexität nicht ohne größeren Aufwand erkennen. Das gleiche gilt, wenn der Ausdruck Fehler beinhaltet. Im Zweifelsfall muss je nach Situation abgewägt werden, welche Variante besser geeignet ist.

7.2 Validierungsbibliotheken

Eine weitere Möglichkeit zur Validierung bietet die Nutzung bereits vorhandener Bibliotheken. Diese haben den Vorteil bereits einen hohen Funktionsumfang an Prüfungen von unterschiedlichen Datentypen anbieten zu können. Abhängig von ihrer Popularität und ihrem Alter können sie so bereits oft bewährte Lösungen für häufig auftretende Probleme anbieten.

FluentValidation ist eine der am häufigsten eingesetzten Bibliotheken im .NET-Umfeld, wenn es um die Validierung von Eingabeparametern geht. Neben Validatoren für gängige Überprüfungsmuster existiert bereits eine Implementierung zur Überprüfung einer E-Mail-Adresse.

```
1  public class EmailValidator : AbstractValidator<string>
2  {
3      public EmailValidator()
4      {
5          RuleFor(email => email).EmailAddress();
6      }
7  }
8
9  // ...
10
11 var validator = new EmailValidator();
12 var result = validator.Validate("demo.user@example.com");
```

Listing 42: Implementierung und Ausführung eines E-Mail-Validators mittels FluentValidation

Ein Validator in FluentValidation besteht aus einer Reihe von Regeln, die definiert werden. Listing 43 zeigt eine solche Implementierung, die als Regel die Build-In-Funktionalität der Bibliothek verwendet. Der Aufruf erfolgt dann im Anschluss durch Instanziierung der Klasse und dem Aufruf der Methode „Valide()“ inklusive dem Wert, der überprüft werden soll.

Eine weitere Möglichkeit stellt die Klasse MailAddress dar. Die Klasse ist in .NET Teil des „System.Net.Mail“-Namensraums und dient dazu E-Mail-Adressen darzustellen. Die primäre Aufgabe dieser Klasse ist die Speicherung von Sender- und Empfänger-E-Mail-Adressen bei der Arbeit mit E-Mail-Funktionalitäten innerhalb von .NET-Anwendungen. Sie kann jedoch auch dafür verwendet werden, um die Syntax einer E-Mail-Adresse zu überprüfen.

```
static void Main(string[] args)
{
    var email = "demo@example";

    try
    {
        var mailAddress = new MailAddress(email);
        Console.WriteLine(email + " is valid");
    }
    catch
    {
        Console.WriteLine(email + " is invalid");
    }
}
```

Listing 43: Prüfung einer E-Mail mit Hilfe der MailAddress-Klasse

Die Überprüfung einer E-Mail-Adresse mit Hilfe der MailAddress-Klasse wird durch die Instanziierung der Klasse durchgeführt. Die Klasse erwartet bei der Konstruktion eine E-Mail-Adresse als Eingabeparameter. Ist die Instanziierung erfolgreich bedeutet dies, die E-Mail-Adresse war nach den Standards der Klasse syntaktisch richtig. Wird während des Instanziierungsprozesses eine Ausnahme ausgelöst, deutet dies auf eine ungültige E-Mail-Adresse hin.

Um zu überprüfen, inwiefern diese Möglichkeiten eine Option zur Validierung darstellen, sollte im Vorfeld geprüft werden, inwiefern beide Varianten sowohl FluentValidation, als auch die MailAddress-Klasse, auf verschiedene Syntax-Varianten reagieren. Die folgenden zwei Tabellen zeigen sowohl gültige, als auch ungültige E-Mail-Adressen und jeweils das Verhalten beider Optionen:

Szenario	Fluent Validation	MailAddress
Simple E-Mail-Adresse: demo@example.com	Test bestanden	Test bestanden
Lokalanteil ist ein Zeichen lang d@example.com	Test bestanden	Test bestanden
Lokale Domain demo@example	Test bestanden	Test bestanden
Minuszeichen am Ende des Lokalanteils demo-@example.com	Test bestanden	Test bestanden
IP-Adresse als Domainanteil demo@[192.168.178.1]	Test bestanden	Test bestanden
Leerzeichen zwischen Anführungszeichen „ “@example.com	Test bestanden	Test bestanden
Doppelte Punkte zwischen Anführungszeichen „demo..user“@example.com	Test bestanden	Test bestanden
Anführungszeichen und Backslashes werden innerhalb von Anführungszeichen maskiert „demo\“.user\\.extern“@example.com	Test bestanden	Test bestanden

Tabelle 6: Prüfung gültiger E-Mail-Adressen Richtlinien

Szenario	Fluent Validation	MailAddress
E-Mail besitzt mehr als 256 Zeichen	Test bestanden	Test bestanden
Lokalanteil besitzt mehr als 64 Zeichen	Test bestanden	Test bestanden
Kein @-Zeichen demo.example.com	Test fehlgeschlagen	Test fehlgeschlagen
Punkte am Anfang oder Ende .demo.@example.com	Test bestanden	Test fehlgeschlagen
Ungültige Zeichen im Lokalanteil demo“user@example.com	Test bestanden	Test bestanden
Unterstrich im Domainanteil demo@example_com	Test bestanden	Test bestanden

Tabelle 7: Prüfung ungültiger E-Mail-Adressen Richtlinien

Für eine Verwendung als Alternative zu selbstdefinierten regulären Ausdrücken oder Properties sind diese Bibliotheken nicht geeignet. Zufallsgenerierte E-Mail-Adressen würden in jedem Fall, in denen es sich um ungültige Werte handelt, dafür sorgen, dass der Test fehlschlägt. Eine eindeutige Bewertung lässt sich auf Grundlage des

Testergebnisses nicht mehr durchführen. Schließlich könnte der Test auch nur erfolgreich durchgelaufen sein, weil ausschließlich E-Mail-Adressen generiert wurden, die auf die entsprechenden Szenarien passen und nur dadurch den Test erfolgreich passieren konnten. Demzufolge bleibt nur die Verwendung zur Aufstellung eigener Properties als Testmethodik über.

Die fehlende Eignung der getesteten Bibliotheken bezieht sich auf den konkreten Anwendungsfall. Die getesteten Bibliotheken und Implementierungen besitzen ein strikteres Regelwerk, welches sich näher an der Praxis orientiert. Schließlich sind einige der Spezifikationen in der Praxis so nicht anwendbar oder von einer Verwendung wird abgeraten. Abgesehen davon ist es ratsam, auf die Verwendung vorhandener Bibliotheken und Implementierungen zuzugreifen.

7.3 Testauswertung

Die erstellten Tests prüften die Richtlinien, welche in Kapitel 6.2 und folgende Unterkapitel zur Generierung von Testdaten behandelt wurden. Dabei ergab sich auch die Erkenntnis, dass sich die Prüfung des Generators und dessen erzeugten Werte, anders gestalten als es bei einem EBT der Fall ist. Die Zufälligkeit sorgt dafür, dass keine exakte Annahme getroffen werden kann, was das erwartete Ergebnis betrifft. Demzufolge können Werte nur gegen Eigenschaften geprüft werden. Das macht es jedoch an einigen Stellen schwierig sicherzustellen, dass die erzeugten Werte dem erwarteten Ergebnis entsprechen. So arbeitet der implementierte IPv4-Generator bspw. im Kontext von reservierten Adressbereichen nach dem Ausschlussprinzip.

Sollen Teile dieser Adressbereiche in die Generierung mit einbezogen werden, muss der Generator entsprechend konfiguriert werden. Die Überprüfung dieses Verhaltens erweist sich jedoch als Problem. Wird dem IPv4-Adressgenerator zum Beispiel die Erzeugung von IP-Adressen aus privaten Netzbereichen erlaubt, kann keine Sicherstellung erfolgen, dass dieser auch tatsächlich IPv4-Adressen aus diesem Bereich erzeugt. Zwar ließe sich die Anzahl der Testdurchläufe und damit die Wahrscheinlichkeit erhöhen, dass eine IPv4-Adresse aus diesem Bereich erzeugt wurde, schlussendlich bietet dies keine Garantie, dass der Fall auch tatsächlich eintritt. Umgekehrt lässt sich jedoch prüfen, ob IPv4-Adressen generiert werden, die einem nicht erlaubten Adressbereich zugeordnet sind.

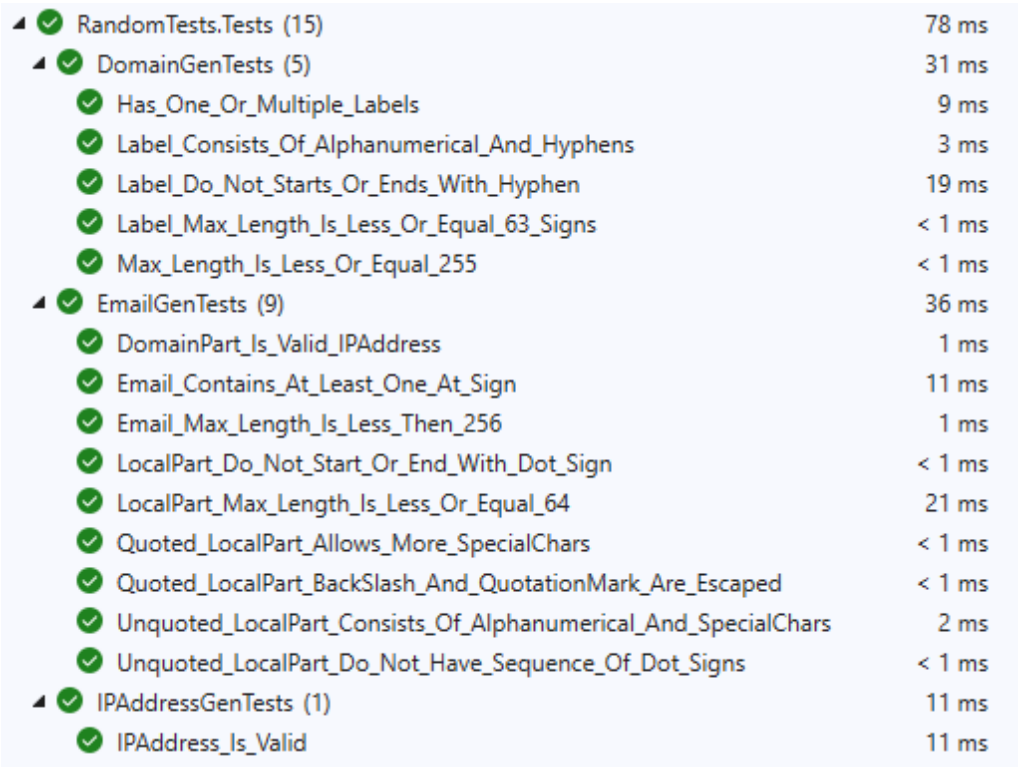

Abbildung 6: Ergebnisse der Generatortests

Bei den Testdurchläufen zur Prüfung der erzeugten Zufallswerte mit den Standardwerten von 100 Iterationen pro Test ist es zu keinerlei Fehlern gekommen. Insgesamt kann der Test damit als Erfolg bezeichnet werden. Es gilt jedoch zu bedenken, dass die Ergebnisse schwierig einzuschätzen bleiben. Zwar können die Tests auch mit einer weitaus höheren Anzahl an Iterationen durchgeführt werden, jedoch gibt es keine Garantie, dass der Generator seinem Verhalten entsprechend gültige Werte erzeugt.

8 Zusammenfassung

Im Abschlusskapitel dieser Arbeit wird zusammengefasst, wie die Untersuchung und Entwicklung im Bereich von Property Based Testing zu einem tieferen Verständnis dieses Themas geführt hat.

8.1 Fazit

Die Arbeit hat dazu beigetragen einen tieferen Einblick in das Konzept von Property Based Tests zu erhalten. Es wurden sowohl die Vorteile als auch die Herausforderungen identifiziert, die mit der Anwendung von PBT im Unterschied zu Example Based Tests verbunden sind. Tests mit zufallsgenerierten Werten tragen durch das Testen eines breiteren Wertebereiches dazu bei Fehler aufzudecken, die durch klassische Example Based Tests möglicherweise unentdeckt geblieben wären.

Die Recherche nach vorhandenen Bibliotheken hat gezeigt, dass die Thematik des Testens mit zufallsgenerierten Werten zwar bereits diskutiert und angewendet wird, die Auswahl verfügbarer Bibliotheken jedoch im Vergleich zu Example Based Tests deutlich geringer ausfällt. Auch die Definition, was genau eine Property definiert, stellt sich als Herausforderung dar. Viele der Bibliotheken stellen für das Generieren von Zufallsdaten lediglich Generatoren für primitive Datentypen zur Verfügung. Darunter zählen oft Integer, Boolean, aber auch Strings. In einigen Fällen war die Unterstützung von komplexeren Datentypen wie Datum oder Zeitstempel ebenfalls gegeben. Basierend auf den Erkenntnissen wurde eine Bibliothek ausgewählt, die anhand der vorliegenden Bewertung als geeignet zum Weiterarbeiten eingestuft wurde.

Die Implementierung von weiteren Generatoren setzte eine vorangegangene Auseinandersetzung mit den dazu notwendigen Spezifikationen und Standards voraus. Dies führte zu der Erkenntnis, dass ein Datentyp trotz seiner vermeintlich unscheinbar einfachen Darstellung, so wie es in dem in der Arbeit behandelten Datentyp E-Mail-Adresse der Fall gewesen ist, dennoch umfangreichen Richtlinien unterliegt.

Auf Basis der Auswahl und den Erkenntnissen aus den Recherchen zu den Datentypen wurde die ausgewählte Testbibliothek um eine Reihe von Generatoren erweitert. Damit es nun möglich zufallsgenerierte Werte für die Datentypen E-Mail-Adresse, IPv4-Adresse und Domainnamen zu erzeugen. Die Generatoren sind konfigurierbar,

sodass sich je nach Datentyp auch „fast“ gültige Werte erzeugen lassen. Schließlich kann auch das Testen auf ungültige Werte Bestandteil einer Testsuite sein.

Die Erzeugung und Konfigurierung von Generatoren wurde außerdem durch eine Abstraktionsebene in Form eines Fluent Interfaces erweitert. Dies erleichtert die Handhabung, indem der Nutzer bei der Erzeugung eines Generators bis hin zur Testdurchführung durch die IDE unterstützt wird. In der Testautomatisierung wurden darüber hinaus Erkenntnisse gesammelt und Methoden untersucht, mit denen Generatoren hinsichtlich ihrer Funktionalität hin getestet werden können. Die Erweiterung ist zudem als Projekt auf GitHub veröffentlicht worden und ist somit anderen Entwickler frei zugänglich. Das Git-Repository kann unter der folgenden URL abgerufen werden: https://github.com/Back2Roots/CsCheck.Extension.git.

8.2 Ausblick

Die Erweiterung legt mit den implementierten Generatoren für E-Mail-/ IPv4-Adressen und Domainnamen den Grundstein. Dies gilt auch für die Konfigurationsmöglichkeiten, bei denen zusätzliche Optionsparameter in Betracht gezogen werden können, um das Generatorverhalten noch gezielter in bestimmte Bahnen lenken zu können. Der Quellcode bietet zudem Raum für Optimierungen. So könnten weitere Konfigurationsparameter zu bestehenden Generatoren hinzugefügt werden. Die Evaluierung weiterer Datentypen wäre ein möglicher nächster Schritt. Mit dem Hinzufügen neuer Datentypen und -strukturen würde sich das Anwendungsfeld weiter vergrößern.

Literaturverzeichnis

[1] Khan, R., Srivastava, A. K. & Pandey, D.: *Agile approach for Software Testing process*, International Conference System Modeling & Advancement in Research Trends (SMART), Moradabad, 2016.

[2] Cohn, M.: *Succeeding with Agile: Software, Development Using Scrum*, Boston: Addison-Wesley 2009

[3] Fowler, M.: *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley 2002.

[4] Vocke, H.: *The Practical Test Pyramid*. https://martinfowler.com/articles/practical-test-pyramid.html. 2018, abgerufen am 21.05.2024

[5] Kröger, A.: *Methoden automatisierter Softwaretests in modernen DevOps Umgebungen – Eine Literaturanalyse*. Osnabrück: Hochschule Osnabrück 2022

[6] Jorgensen, P. C.: *Software testing. A Craftsman's Approach.* Auflage 4. Boca Raton: CRC Press, 2014.

[7] Bender, J., McWherter, J.: *Professional Test Driven Development with C#. Developing Real World Applications with TDD*. Hoboken: John Wiley & Sons. 2011

[8] Khorikov, V.: *Unit Testing. Principles, Practices, and Patterns*, Shelter Island: Manning Publications 2020

[9] Hall, G. M.: *Agile Softwareentwicklung mit C#. Best Practices und Patterns für flexiblen und adaptiven C#-Code*. Heidelberg: dpunkt.verlag GmbH 2015.

[10] Astels, D.: *Shift Your Paradigm!. One Assertion Per Test*. https://www.artima.com/weblogs/viewpost.jsp?thread=35578. 23.02.2004, abgerufen am 19.05.2024

[11] Fowler, M.: *Mocks aren't stubs*. https://martinfowler.com/articles/mocksArentStubs.html. 2007, abgerufen am 29.06.2024

[12] Spillner A., Linz, T.: *Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified Tester*. Auflage 1. Heidelberg: dpunkt.verlag GmbH 2003.

[13] Claessen, K., Hughes, J.: QuickCheck: *A Lightweight Tool For Random Testing Of Haskell Programs*. Göteborg: Chalmers University of Technology. 2000

[14] Antoy, S., Hamlet, D.: *Automatically Checking an Implementation against Its Formal Specification*. In: IEEE Transactions on Software Engineering, 26/2000, Seite 55 - 69

[15] Papadakis, M. & Sagonas, K.: A *PropEr integration of types and function specifications with property-based testing*. New York: Association for Computing Machinery 2011

[16] Rushby, J.: *Automated Test Generation and Verified Software,* VSTTE, Zürich 2005, *Verified Software: Theories, Tools, Experiments*. Lecture Notes in Computer Science. Band 4171. Auflage 1. Heidelberg: Springer Berlin

[17] Rebert, A.: *What is Property-based Testing?* https://www.mayhem.security/blog/what-is-property-based-testing. 2021, abgerufen am 30.05.2024

[18] Goldstein, H., W. Cutler, J., Dickstein, D., C. Pierce, B., Head, A.: *Property-Based testing in practice*, ICSE 2024 Lisbon 2024

[19] Warren, G.: *System.Random class,* 2024, https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-random, abgerufen am 12.06.2024

[20] Code Maze: *Random Class in C#*, https://code-maze.com/csharp-random-class/. abgerufen am 12.06.2024

[21] O'Neill, M.: *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Claremont: Harvey Mudd College 2014

[22] O'Neill, M.: *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. https://www.pcg-random.org/rng-basics.html. 2018, abgerufen am 11.06.2024

[23] Oragui, D.: *Software Documentation Best Practices [With Examples]*. https://helpjuice.com/blog/software-documentation. 2024, abgerufen am 13.06.2024

[24] FsCheck (o. V.): *GitHub - fscheck/FsCheck: Random Testing for .NET*. https://github.com/fscheck/FsCheck, abgerufen am 29.04.2024

[25] FsCheck (o. V.): *Random Testing for .NET*. https://fscheck.github.io/FsCheck/, abgerufen am 29.04.2024

[26] o. V.: *Newest „FSCheck“ questions*. https://stackoverflow.com/questions/tagged/fscheck, abgerufen am 03.05.2024

[27] Lloyd, A.: *GitHub - AnthonyLloyd/CsCheck: Random testing library for C#*. https://github.com/AnthonyLloyd/CsCheck, abgerufen am 27.04.2024

[28] Lloyd, A.: *Comparison with other random testing libraries*. https://github.com/AnthonyLloyd/CsCheck/blob/master/Comparison.md, abgerufen am 27.04.2024

[29] Resnick, P.: *Internet Message Format*. https://www.ietf.org/rfc/rfc2822.txt. April 2001, abgerufen am 06.05.2024

[30] Klensin, J.: *Application Techniques for Checking and Transformation of Names*. https://www.ietf.org/rfc/rfc3696.txt. Februar 2004, abgerufen am 06.05.2024

[31] Crocker, D.: *STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES*. https://www.ietf.org/rfc/rfc822.txt. August 1982, abgerufen am 17.06.2024

[32] Yao, J., Mao, W.: *SMTP Extension for Internationalized Email*. https://www.ietf.org/rfc/rfc6531.txt. Februar 2012, abgerufen am 18.05.2024

[33] Yang, A., Steele, S., Freed, N.: *Internationalized Email Headers*. https://www.ietf.org/rfc/rfc6532.txt. Februar 2012, abgerufen am 17.06.2024

[34] Resnick, P.: *Internet Message Format*. https://www.ietf.org/rfc/rfc5322.txt. Oktober 2008, abgerufen am 04.05.2024

[35] Klensin, J.: *Simple Mail Transfer Protocol*. https://www.ietf.org/rfc/rfc2821.txt. April 2001, abgerufen am 10.05.2024

[36] *Simple Mail Transfer Protocol*. https://www.ietf.org/rfc/rfc5321.txt. Oktober 2008, abgerufen am 04.05.2024

[37] Bradner, S.: *The Internet Standards Process -- Revision 3*. https://www.ietf.org/rfc/rfc2026.txt. Oktober 1996, abgerufen am 06.05.2024

[38] Postel, J.: *Internet Protocol*. https://www.ietf.org/rfc/rfc791.txt. September 1981, abgerufen 13.06.2024

[39] Cotton, M., Vegoda, L.: *Special Use IPv4 Addresses*. https://www.ietf.org/rfc/rfc5735.txt. Januar 2010, abgerufen 14.06.2024

[40] Cotton, M., Vegoda, L., Bonica, R., Haberman , B.: *Special-Purpose IP Address Registries*. https://www.ietf.org/rfc/rfc6890.txt. April 2013, abgerufen am 20.06.2024

[41] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., Lear , E.: *Address Allocation for Private Internets*. https://www.ietf.org/rfc/rfc1918.txt. Februar 1996, abgerufen am 14.06.2024

[42] Mainiero, N.: *Property Based Testing - eine Einführung*. In: Java aktuell, 2/2018, Seite 43 - 47

[43] Mockapetris , P.: *Domain names - implementation and specification*. https://www.ietf.org/rfc/rfc1035.txt. November 1987, abgerufen am 06.06.2024

[44] Gamma, E., Helm, R., Johnson, R. & Vlissides, J.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Auflage 37. Indianapolis: Pearson Education 2007

[45] Klensin, J.: *RFC Errata*. https://www.rfc-editor.org/errata/eid10032005. Juli 2005, abgerufen am 17.06.2024

Selbstständigkeitserklärung

Ich versichere, dass ich die Arbeit selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Wörtlich oder sinngemäß aus anderen Quellen übernommene Textstellen, Bilder, Tabellen u. a. sind unter Angabe der Herkunft kenntlich gemacht.

Weiterhin versichere ich, dass diese Arbeit noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Kamenz, 26.07.2024 _____

Ort, Datum Unterschrift