Kontinuierliches Ende-zu-Ende Testen von Software Visualisierungen

Bachelorarbeit

Tim Straßburg

Bachelorarbeit
12. April 2023

Software Engineering Group
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
Prof. Dr. Wilhelm Hasselbring
Alexander Krause-Glau M.Sc.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Das Testen von Anwendungen ist ein wichtiger Prozess bei der Entwicklung. Dadurch soll nicht nur die Erkennung von Bugs ermöglicht werden, sondern auch eine Voraussetzung für qualitativ hochwertigen Code. End-to-End Tests spielen dabei eine besondere Rolle, weil sie die meiste Zeit der Tests einnehmen und schwer zu automatisieren sind. Diese Schwierigkeit ergibt sich insbesondere beim Testen von Graphical User Interfaces. Hierzu müssen, nach einer simulierten Interaktion Bilder aufgenommen und verglichen werden.

Diese Arbeit zeigt die Implementierung erster automatisierter End-to-End Tests für ExplorViz. Die Tests finden im Frontend der Anwendung statt und überprüfen die Funktionalität von ausgewählten Nutzerfunktionen, welche in Testszenarien beschrieben werden. Die Entwicklung findet mit dem Framework Gauge statt und wird anschließend über Docker in die Continuous Integration Pipeline von GitHub integriert. Weiterhin werden die End-to-End Tests in zwei exemplarischen Szenarien getestet und die Ergebnisse evaluiert. Aus der Automatisierung der End-to-End Tests resultiert eine kontinuierliche Überprüfung der Anwendung, wodurch die Vorteile einer schnellen Erkennung von Bugs und damit höheren Qualität des Codes entstehen. Weiterhin erfolgt eine Zeitersparnis, da die End-to-End Tests nicht durch eine Person gestartet und durchgeführt werden müssen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1. Motivation

Das Testen einer Software während der Entwicklung spielt eine große Rolle. Die Hauptfunktion besteht darin, zu testen, ob die Ergebnisse mit den Erwartungswerten der Stakeholder des Systems übereinstimmen. Dadurch können beispielsweise Bugs oder fehlende Anforderungen an das System festgestellt werden. Tests können somit Aussagen über die Qualität der entwickelten Software treffen. Es gibt verschiedene Arten von durchführbaren Tests [8], welche in der Testpyramide nach Mike Cohn verdeutlicht werden. Bei dieser (üblichen) Kategorisierung der Tests nach Cohen werden diese nach Granularität, Geschwindigkeit und Isolation voneinander differenziert. Unit-Tests stellen die unterste Ebene der Pyramide dar und testen lediglich isolierte Codeabschnitte [15]. Nach den erfolgreichen Unit-Tests folgen Integrationstests, mit welchen geprüft wird, ob beim Zusammenwirken der einzelnen Komponenten Fehler entstehen. Mit steigender Integration der Tests, also dem Zusammenwirken unterschiedlicher Komponenten, erhöht sich auch die Dauer des Tests In der obersten Stufe der Pyramide, dem letzten Schritt, folgt der Test des gesamten Systems - der End-to-End Test (E2E-Test) [8, 15]

Das sich seit 2013 in der Entwicklung befindende Projekt ExplorViz durchläuft im Rahmen der Forschung und Verbesserung immer wieder unterschiedliche Phasen der Softwareentwicklung [9]. ExplorViz visualisiert Software in 3D basierend auf einer Live-Trace-Analyse. Da Unit- und Integrationstests nicht alle Fehlerquellen abdecken, da sie Visualisierungen nicht vollkommen abtesten können, werden geeignete E2E-Tests für das Projekt entwickelt. Die Schwierigkeit bei der Entwicklung von E2E-Tests für ExplorViz besteht darin, dass die 3D-Modelle verglichen werden müssen und dafür keine Methoden, wie die Assert-Methode, zur verfügung stehen. Die E2E-Tests sollen die Möglichkeit intendieren, unerwartete Fehler in der Visualisierung zu erkennen. Diese können beispielsweise durch Arbeiten im Backend von ExplorViz entstehen.

1.2. Ziele

In diesem Abschnitt werden die Teilziele der Arbeit vorgestellt. Dabei wird auf jedes Ziel kurz eingegangen.

1.2.1. Z1: Identifikation der zu testenden Funktionen

Im ersten Schritt identifizieren wir, welche Funktionen von ExplorViz im Rahmen des E2E-Tests geprüft werden sollen indem unter anderem Gespräche innerhalb des Teams geführt werden.

1.2.2. Z2: Identifikation eines geeigneten Tools für die Tests

Nachdem die zu testenden Funktionen identifiziert wurden, beginnt darauf aufbauend die Suche eines geeigneten Tools zur Durchführung der Tests. Im Fokus der Tests[1] steht die Interaktion der Nutzer mit dem Frontend.

1.2.3. Z3: Implementierung des Tests

Nachdem das geeignete Tool identifiziert wurde, werden die Tests mit dessen Hilfe implementiert. Sobald die Tests funktionsfähig sind, erfolgt die Herstellung der Lauffähigkeit mittels Docker, worauf in Abschnitt 2.2 eingegangen wird. Hierdurch erschaffen wir erstens eine Umgebung, mit der die Tests betriebssystemunabhängig durchgeführt werden können und zweitens werden während des Tests geänderte Umgebungsvariablen und Daten nach dem Durchlauf der Tests wieder zurückgesetzt. Mit der Lauffähigkeit in Docker können wir die Tests mittels einer Continous Integration Pipeline (CI-Pipeline), einer virtuellen Maschine, durchführen.

1.2.4. Z4: Exemplarische Anwendung in der Praxis

Nach der Implementierung der Tests erfolgt eine exemplarische Anwendung. Um dies leisten zu können, werden die Tests über ein GitHub Repository zur Verfügung gestellt. Dort sollen die Tests in die CI-Pipeline eingebunden werden. Durch zwei exemplarische Szenarien wird eine potenzielle Einbindung der Tests in den täglichen Arbeitsablauf geprüft. Daraus könnte sich beispielsweise ergeben, dass eine kontinuierliche Durchführung der E2E-Tests Hinweise auf Bugs liefert.

1.3. Struktur der Arbeit

In Kapitel 2 findet eine nähere Beschreibung der theoretischen Grundlagen verwendeten Technologien statt. Kapitel 3 beschreibt die Vorgehensweise in der Konzeption der E2E-Tests. In Kapitel 4 folgt darauf aufbauend die Implementierung jener. Unter anderem wird dort beschrieben, welche Funktionen von ExplorViz getestet werden. In Kapitel 5 wird die Evaluation durchgeführt. Dort werden zwei exemplarische Szenarien beschrieben und evaluiert, inwiefern sich die Durchführung und Ergebnisse der E2E-Tests zwischen Szenarien unterscheiden.

[1] Zur Vereinfachung des Leseflusses werden an diversen Stelle die Begriffe Test und Tests als Synonym für E2E-Tests verwendet

Kapitel 2

Grundlagen und Technologien

Dieses Kapitel beinhaltet die notwendige theoretischen Grundlagen und Technologien, welche in der Arbeit genutzt wurden. Diese werden vorgestellt und deren Nutzung im Kontext dieser Arbeit kurz begründet.

2.1. ExplorViz

ExplorViz ist ein Ansatz zur Echtzeit-Visualisierung von Softwarelandschaften [6]. Dabei handelt es sich um einen Software Visualisation as a Service (SVaaS) Ansatz, der öffentlich zur Verfügung steht. Das System besteht aus mehreren unterschiedlichen Komponenten (Abbildung 2.1) . Die angewendeten Technologien werden für unterschiedliche Aufgabenbereiche eingesetzt. Dabei wird in Monitoring, Analyse und Visualisierung unterschieden (Abbildung 2.1).

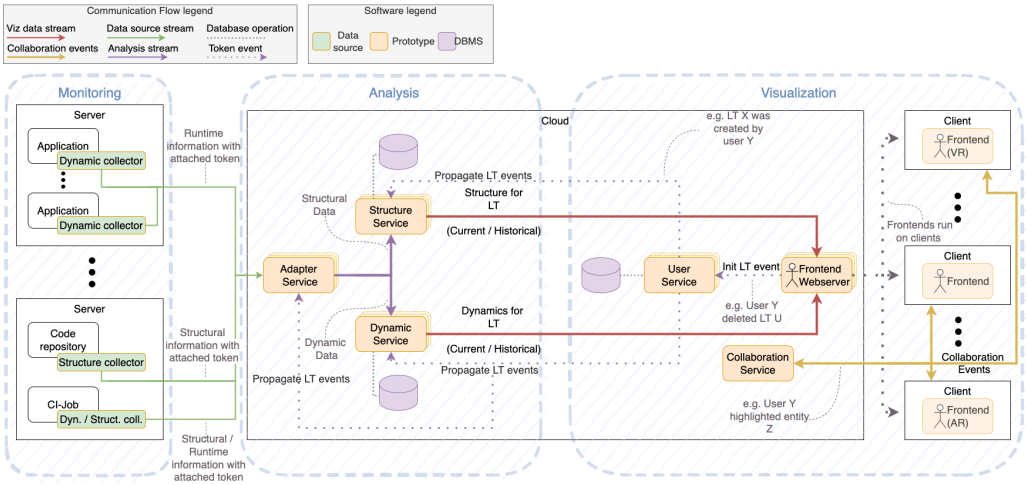

Abbildung 2.1. Architektur ExplorViz[a]

[a] `https://explorviz.dev/3-architecture/`

Durch die Aufteilung in unterschiedliche Komponenten innerhalb der Bereiche soll eine öffentlich erreichbare Cloud-Anwendung zur Verfügung stehen, welche sich durch die gleichzeitige Nutzung mehrerer NutzerInnen auszeichnet [13]. Zudem ermöglicht eine Unterteilung in unterschiedliche Bereiche, dass diese unabhängig voneinander gestaltet werden können. Das trifft nicht nur auf die Bereiche als solche, sondern auch auf die in den Bereichen verwendeten Technologien zu. Das führt zu einer höheren Flexibilität in Bezug auf die Austauschbarkeit der Komponenten.
Der Bereich des Monitorings übernimmt die Aufgabe des Sammelns von Daten den von ExplorViz

überwachten Anwendungen. Die Überwachung der Anwendungen wird mit dem inspectIT Ocelot Java Agent realisiert. Der Ocelot Agent[1] ermöglicht eine genaue Aufzeichnung der während der Nutzungszeit der überwachten Anwendung verwendeten Methoden und sammelt diese Informationen. Der in Ocelot enthaltene *Open Telemetry exporter* leitet die Daten über gRPC[2] mittels des *Ocelot Open Telemetry exporters* an den Analysebereich weiter. Die übertragenen Daten werden auch als *Spans* bezeichnet. Jeder Span ist dabei mit einer einzigartigen Kennung versehen, um diesen der jeweiligen überwachten Anwendung zuordnen zu können [12].
Die *Spans* werden im Analysebereich vom *Open Telemetry Collector*, welcher als Gateway fungiert, an den Adapter Service weitergeleitet. Der Datenfluss innerhalb des Analysebereichs, beispielsweise das Senden und Empfangen von Daten, basiert dabei auf Apache Kafka Streams[3]. Der Adapter Service prüft die empfangenen *Spans* auf Korrektheit. Diese Prüfung wird über *Landscape Tokens* (LT) realisiert, welche vorhanden sein müssen [12]. Das LT ist ein Schlüssel-Wert Paar, welches eine Softwarelandschaft repräsentiert. Bei der Softwarelandschaft handelt es sich um einen virtuellen Raum, der einem individuellen Nutzer gehört und in welchem die Anwendungen visualisiert werden [13]. Der Adapter Service wandelt jeden akzeptierten Span in dynamische und strukturelle Daten (Abbildung 2.1). Strukturelle Daten zeichnen sich dabei durch eine hohe Wiederholungsrate aus. Ein Beispiel dafür ist der Hostname. Dynamische Daten hingegen beinhalten zeitbezogene Informationen über genutzte Methoden der Zielanwendung. Diese werden vom *Dynamic Service* gesammelt für die Visualisierung aggregiert. Die strukturellen Daten werden vom *Structured Service* verwaltet, welcher die gesammelten Informationen in einer Baumstruktur darstellt.
Im Bereich der Visualisierung fließen die aus den überwachten Anwendungen gesammelten Daten zusammen und es folgt die Visualisierung. Die wichtigste Komponente der Visualisierung ist das Frontend von ExplorViz. Das Frontend wird dabei durch einen Webserver bereitgestellt (Abbildung 2.1). Die dargestellte Visualisierung wird alle zehn Sekunden, anhand der zur Verfügung stehenden Informationen, aktualisiert [12]. Die Visualisierung wird über den Browser mittels Three.js, einer Bibliothek zur abstrahierten Nutzung von WebGL, gerendered. Bei den Informationen handelt es sich um das LT der strukturellen und dynamischen Daten. Das Management der LT übernimmt der User Service [12].
Die Darstellung der überwachten Anwendung in der Visualisierung erfolgt über die *Stadtmetapher* [17]. Dies funktioniert, indem die genutzten Methoden konstant überwacht werden. Die gesammelten Daten werden dann in einer Softwarestadt dargestellt. Weiterhin bietet ExplorViz Interaktionsmöglichkeiten mit der dargestellten Visualisierung. Zum einen können detaillierte Informationen abgerufen werden. Dazu muss man den Mauscursor über ein dargestelltes Element bewegen. Daraufhin wird ein Popup dargestellt, welches Informationen darüber ausgibt, wie viele Objekte durch eine Methode instanziiert wurden. Das Popup befindet sich in Abbildung 2.2 in der oberen rechten Ecke des *petclinic-visits-service* Moduls. Zum anderen ist eine weitere Funktion das Markieren von Elementen in der Softwarelandschaft. Hierzu muss man ein beliebiges Element anklicken. Das Element wird dann standardmäßig rot verfärbt [10]. Im Rahmen dieser Arbeit stellt ExplorViz die Software dar, die um E2E-Tests erweitert werden soll.

[1] https://www.inspectit.rocks/
[2] https://grpc.io/
[3] https://kafka.apache.org/documentation/streams/

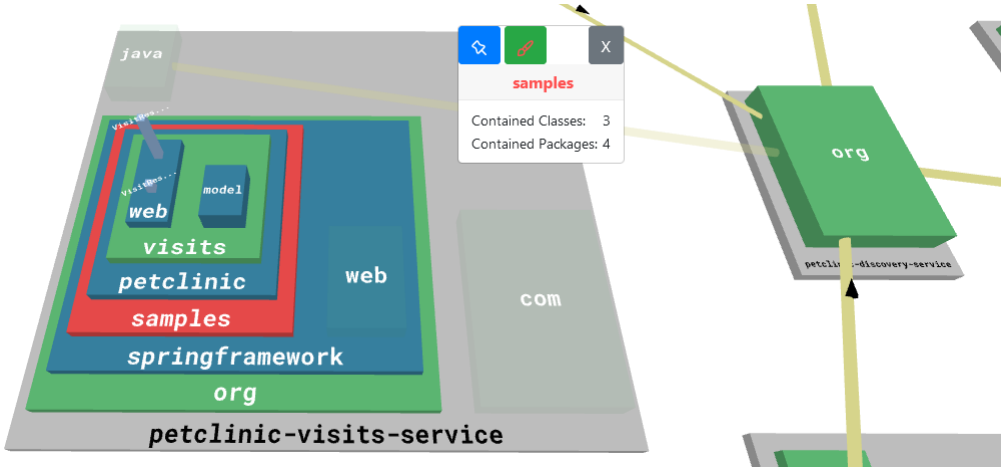

Abbildung 2.2. ExplorViz Beispielfunktionen

2.2. Docker

Bei Docker[4] handelt es sich um eine Technologie, welche Container virtualisiert und diese für die Nutzung und Verbreitung von Softwareanwendungen bereitstellt. Bei einem Container handelt es sich um eine isolierte Umgebung, welche alle für eine Softwareanwendung notwendigen Voraussetzungen enthält (beispielsweise ein lokales Filesystem). Mit diesen Voraussetzungen ist ein Container unabhängig vom Hostsystem. Weiterhin können Container über Netzwerke miteinander kommunizieren oder mit Bind Mount Zugriff auf Inhalte vom Hostsystem erhalten. Mit Bind Mount wird eine Datei oder ein Ordner in den Container gemountet und steht diesem damit zur Verfügung.[5] Wichtig dabei ist, dass, wenn ein Container gelöscht wird, alle Daten, welche dieser beinhaltet, ebenfalls gelöscht werden. Wurden die Daten mit einem Bind Mount auch auf dem Hostsystem gesichert, stehen diese danach weiterhin zur Verfügung. Ein Container wird über ein Image definiert. Die Anweisungen zum Erstellen eines Images werden in ein Dockerfile geschrieben. Bei einem Image handelt es sich um eine Schablone, welche Anweisungen zur Generierung eines Containers beinhaltet. Das Image basiert dabei meistens auf anderen Images, welche dann weiter definiert werden können. Beispielsweise könnte Ubuntu als Basis-Image verwendet werden. [6] Docker wird im Rahmen der Entwicklung als Werkzeug genutzt, mit welchem die Tests auf Befehl innerhalb eines Containers automatisch durchgeführt werden sollen.

2.2.1. Docker Compose

Bei Docker Compose handelt es sich um ein Tool, welches das Management von multi-container Docker Anwendungen ermöglicht. Damit die verschiedenen Container in einer vorgegeben Weise Konfiguriert und gestartet werden, benötigt es eine yml-Datei. Damit wird erreicht, dass die möglichen Abhängigkeit und Schnittstellen zwischen den Container immer gegeben sind und es beim Starten zu keinen Problemen kommt.[7] Die yml-Datei für Docker Compose besteht dabei aus mehreren Komponenten. Diese sind beispielsweise Services, Netzwerke und Volumes. Für die jeweiligen Komponenten gibt es

[4] https://docs.docker.com/get-started/overview/
[5] https://docs.docker.com/storage/bind-mounts/
[6] https://docs.docker.com/get-started/02_our_app/
[7] https://docs.docker.com/compose/

noch weitere Konfigurationsmöglichkeiten. Die Servicekomponente stellt dabei eine abstrakte Form von Computerressource dar, auf welcher Anwendungen laufen können. Diese können unabhängig von anderen Services spezifiziert und ausgetauscht werden. Ein Service kann dabei endlich viele Anwendungen beinhalten. Dabei ist hervorzuheben, dass in einer yml-Datei ein Service bestimmt werden muss, andernfalls ist die Datei nicht funktionsfähig. Die innerhalb der Servicekomponenten definierten unterschiedlichen Anwendungen können Teil eines Netzwerkes sein und so miteinander kommunizieren. Als Defaulteinstellung laufen alle Anwendungen innerhalb eines Services in einem Netzwerk, welches unter dem Alias des Services läuft. Mit Volumes können Speicherorte geteilt werden, dies bezieht sich unter anderem auf die Konfiguration von Bind Mounts, wodurch auf Inhalte des Hostsystems oder Containers zugegriffen werden kann.[8] Die Module von ExplorViz sind auf unterschiedliche Container verteilt. Das bezieht auch den Testcontainer mit ein. Auf Docker Compose wird für die Strukturierung des Starts der Container zurückgegriffen.

2.3. Continuous integration

Continous Integration (CI) ist ein Begriff aus der agilen Softwareentwicklung, um Änderungen des Quellcodes effizient und fehlerfrei zu überführen. Dabei wird überprüft, ob die Änderungen des Codes vordefinierte Anforderungen erfüllen. Zum einem wird überprüft, ob die Software erstellt werden kann und zum anderen, ob alle Tests (Unit-Test, Integrations-Tests und E2E) erfolgreich ausgeführt werden. Sollte es zu einem Fehler kommen, ist es nicht möglich den Code in den Produktivcode zu übertragen und die Fehler müssen zuerst behoben werden. Dadurch können erstens manuelle Tests reduziert und zweitens gewährleistet werden, dass der Code eine hohe Qualität und Robustheit aufweist. Dies ermöglicht eine höhere Frequenz in der Übernahme von Produktivcode. [16]. Der E2E-Test wird als Teil der CI im Projekt implementiert.

2.3.1. GitHub Actions

GitHub Actions ist die CI-Pipeline der Versionsverwaltungsplatform GitHub. Dabei werden durch GitHub Actions virtuelle Maschinen (Runner), basierend auf Linux, MacOS oder Windows, für die Nutzung von CI-Pipelines bereitgestellt. Weiterhin können mehrere Runner bereitgestellt werden und auch simultan arbeiten. Die Runner arbeiten mit Workflows. Der Workflow ist ein automatisierter Prozess, welcher ausgelöst wird, wenn ein bestimmtes Event (beispielsweise ein Pull Request) im Repository ausgelöst wird. Die Konfiguration der Workflows wird über eine yml-Datei realisiert, welche im jeweiligen Repository vorhanden sein muss. Innerhalb der yml-Datei werden Jobs definiert. Dabei handelt es sich um Schritte, welche vom Runner ausgeführt werden, wobei jeder dieser Schritte entweder ein Shell Skript ausführt oder eine Action. Actions sind spezifische Anwendungen, welche Aufgaben wiederholend ausführen. Dies ist ein Tool, welches durch GitHub Actions angeboten wird und unter anderem redundanten Code vermeiden soll. Es können auch eigene Actions geschrieben werden.

Listing 2.1. GitHub Actions Beispiel yml-Datei

```
1  name: Build Docker Container
2  on: push
3  jobs:
4     Test:
```

[8] https://docs.docker.com/compose/compose-file/

```
5         runs-on: ubuntu-latest
6         steps:
7            - uses: actions/checkout@v3
8            - name: Build Container
9              run: docker compose up -d
```

Bei der Erstellung der yml-Datei müssen bestimmte Schritte eingehalten werden. So muss diese innerhalb einer bestimmten Ordnerstruktur liegen. Weiterhin muss mit *on* definiert werden, mit welcher Aktion der Workflow ausgelöst werden soll, dabei könnte es sich um einen *Push* oder *Pull* handeln (Listing 2.1 ist Zeile 2). *Jobs* definiert die Aufgaben innerhalb des Workflows, in dem ihnen unterschiedliche Namen zugeordnet werden (Listing 2.1 Zeile 4). Unterhalb des Namens wird mit *runs-on* definiert, mit welchem System der Runner (Listing 2.1 Zeile 5) die Jobs ausführt. Letztlich werden dann die Steps definiert. Die Steps sind die konkreten Schritte des Tests.[9] GitHub Actions wird als CI-Pipeline für die Testautomatisierung genutzt werden.

2.4. End-to-End Testing

Es wurde bereits erwähnt, dass E2E-Tests in der Testpyramide nach Mike Cohn an oberster Stelle angesiedelt sind und die Testpyramide nach Granularität, Geschwindigkeit und Isolation der Tests angeordnet ist. Von unten nach oben gesehen, nehmen die Geschwindigkeit, Isolation und Granularität der Tests ab. Weiterhin nimmt mit abnehmender Granularität die Sicht auf das Gesamtsystem zu und macht die Tests auf jeder Stufe komplexer. Das lässt den Rückschluss zu, dass E2E-Tests eine geringe Isolation, Geschwindigkeit und Granularität aufweisen, dafür jedoch einen Rückschluss auf das gesamte System zulassen. Weiterhin handelt es sich bei E2E-Tests meist um sehr komplexe und ressourcenintensive Tests, da diese auf Nutzerebene agieren und damit mit dem User Interface interagieren [14]. Dort werden dann die dort zur Verfügung stehenden Funktionen getestet. Der generierte Output kann dabei nicht nur Rückschlüsse über das Zusammenspiel der unterschiedlichen Komponenten der Anwendung geben, sondern auch mit einem Erwartungswert verglichen werden.[10] Der E2E-Test nutzt lediglich die zur Verfügung stehende Funktionalität der Anwendung ohne tiefer gehende Kenntnisse über die interne Struktur der zu testenden Anwendung und fällt dadurch in die Kategorie von Black-Box-Tests. Da der Test aus der Nutzerperspektive und mit den zur Verfügung stehenden Funktionen durchgeführt wird, deckt dieser dadurch einen größeren Bereich von potenziellen Fehlern ab. Steht beispielsweise eine Funktion im Frontend zur Verfügung, welche jedoch im Backend noch nicht implementiert wurde, würde dieser Fehler durch den Test aufgedeckt werden [3]. Nachfolgend erfolgt eine Vorstellung unterschiedlicher Tools, welche zum Erstellen und Durchführen von Testszenarien für E2E-Tests geeignet sind und nach einer Auswahl genutzt werden sollen.

2.4.1. Gauge-Taiko

Gauge[11] ist ein Open Source Framework zur Durchführung von E2E-Tests für Webanwendungen. Die Nutzung von Gauge für E2E-Tests lässt sich wie folgt beschreiben: Erstens wird eine Spezifikationsdatei (spec-Datei) benötigt, in welcher mit der Markdown-Syntax die Schritte innerhalb eines Szenarios beschrieben werden. Ein Szenario ist eine Abfolge von Schritten innerhalb einer spec-Datei.

[9] https://docs.github.com/de/actions/learn-github-actions/understanding-github-actions

[10] https://smartbear.com/learn/automated-testing/how-to-perform-end-to-end-testing/

[11] https://docs.gauge.org/overview

Das Szenario beschreibt einen eigenständigen Workflow eines Tests. Weiterhin muss mindestens ein Szenario innerhalb einer spec-Datei definiert werden. Schritte werden im Rahmen von Gauge als Steps bezeichnet. Bei Steps handelt es sich um ausführbare Komponenten der spec-Datei. Diese können innerhalb und außerhalb eines Szenarios definiert werden.[12]
Zweitens muss jeder Step mit einer ausgewählten Programmiersprache implementiert werden. Dabei unterstützt Gauge unter anderem Python und JavaScript (JS). Letztere wird im Rahmen der Arbeit verwendet. Die Implementierung der Steps findet in einer weiteren Datei statt. Weiterhin gibt es die Möglichkeit Parameter innerhalb der Steps zu definieren und an die Code-Äquivalente der Steps zu übergeben. Dadurch können zum Beispiel Texteingaben aus Textfeldern einer Internetseite an eine Methode übergeben werden. Um die Interaktionsmöglichkeiten von Gauge zu erweitern, kann die Bibliothek Taiko[13] verwendet werden, welche ein Application Programming Interface (API) zur Interaktion mit dem Browser anbietet. Durch die mit Taiko gebotene API kann beispielsweise ein Linksklick simuliert werden.[14] Dies bietet sich bei der Arbeit mit 3D-Objekten an, welche in einem Canvas gerendert werden, da ohne die genannte Funktionalität von Taiko diese durch Gauge nicht testbar wären. Der Gauge-Runner führt dann die gebildeten Szenarien innerhalb der Spezifikationsdatei durch und generiert einen Bericht.[15] Das Besondere hierbei ist, dass die implementierten Steps innerhalb eines Projektes wiederverwendet werden können. Das heißt, dass jeder Step in unterschiedlichen spec-Dateien wiederverwendet werden kann.[16]

2.4.2. Mocha

Die von ExplorViz genutzte Bibliothek Three.js, schlägt als Test-Tool Mocha vor.[17] Mocha[18] arbeitet mit JS und läuft auf Node.js und im Browser. Dazu wird eine *.js* Datei erstellt, in welcher die Tests verfasst werden.Es muss für die Durchführung der Tests kein neues Projekt initialisiert oder erstellt werden. Neben dem arbeitet Mocha mit assertions. [19]. Dabei handelt es sich um Vergleiche mit erwarteten Werten. Es kann dabei mit beliebigen assertion-Bibliotheken gearbeitet werden. Mocha interagiert bei der Benutzung von Three.js direkt mit dessen Objekten. Three.js ist eine Anwendung, mit welcher man 3D Objekte erstellen kann. Diese werden über einen Renderer in das Canvas des Webbrowsers gezeichnet.[20] Es kann so beispielsweise abgeprüft werden, ob ein bestimmtes Element vorhanden ist. Das Schreiben eines Tests erfolgt durch die Erstellung einer test.js Datei. Dabei wird über require auf notwendige Funktionen, wie assert, zugegriffen. Mocha bietet keine Möglichkeit zur Browserinteraktion, wie die Simulation eines Mausklicks durch eine Funktion. Für die Auflösung dieser Problematik kann die Bibliothek three musketeers genutzt werden.

2.4.3. three musketeers

Three musketeers[21] ist eine Bibliothek, mit welcher Three.js Anwendungen getestet werden können. Three musketeers ermöglicht eine Interaktion mit Canvas-Elementen. Der Renderer wird benötigt, da Three-musketeers mit den dort dargestellten Elementen interagiert. Letztendlich werden der Renderer, die Szene und die Kameraressourcen an eine Musketeer-Instanz übergeben, welche mit dem Window Objekt des Browsers über das Alias *$$$* verbunden wird. Über eine JS Konsole des Browsers wird

[12] https://docs.gauge.org/writing-specifications.html?os=windows&language=javascript&ide=vscode
[13] https://docs.taiko.dev/
[14] https://docs.taiko.dev/api/reference/
[15] https://docs.gauge.org/getting_started/view-a-report.html?os=windows&language=javascript&ide=vscode
[16] https://docs.gauge.org/overview.html?os=windows&language=javascript&ide=vscode
[17] https://threejs.org/docs/#manual/en/buildTools/Testing-with-NPM
[18] https://mochajs.org/
[19] https://nodejs.org/api/assert.html
[20] https://threejs.org/manual/#en/fundamentals
[21] https://webgl.github.io/three-musketeers/

dann über den Alias die Interaktion mit den dargestellten Objekten ermöglicht. Beispielsweise mit der Methode *$$$.find('uniqueIdentifier').click()* wird ein Klick auf das Canvas Element simuliert. Somit sind Simulationen von Nutzerinteraktionen durch Three-musketeers möglich. Da es sich bei Three-musketeers um eine Bibliothek handelt, muss ein Test-Framework, wie beispielsweise Mocha für die Ausführung bereitgestellt werden.[22]

[22] https://github.com/webgl/three-musketeers

Kapitel 3

Konzeption der Tests

In diesem Kapitel werden die Tests konzipiert. Für die Entwicklung eines Tests ist es aus strukturellen Gründen wichtig, einen Plan für den Ablauf zu entwickeln. Dabei folgt eine Unterteilung in einer groben Beschreibung des Plans und einer genauen Ausführung mit Instruktionen für den Test [4]. Everett und McLeod Jr schlagen dazu einen grundlegenden punktuellen Ablauf vor, welcher bei der Konzeption eins Plans berücksichtigt werden sollten[4]:

Nr.	Planungsschritt
1	zu testende Anwendung
2	Risiko und Voraussetzungen
3	Umfang und Limitationen
4	Ansprechpartner (AP) für wirtschaftliche Inhalte der Testplanung und Ausführung
5	AP aus der Entwicklung der Software für Inhalte der Testplanung und Ausführung
6	Ausreichende Testdaten
7	Testumgebung und deren Management
8	Teststrategie
9	Entwicklungsphase
10	Allgemeiner Ablauf der Tests

Die Punkte 1-8 aus der Auflistung beziehen sich auf Voraussetzungen, welche geschaffen werden müssen, um die Tests durchzuführen. Element 9 bezieht sich auf die Entwicklung der Tests. Innerhalb dieser Phase findet die Konzeption der Testfälle, anhand der in Elemente 1-8 gesammelten Daten, und deren anschließende Entwicklung statt. Mit der Bearbeitung von Punkten 1-9 kann mit Punkt 10 letztlich der gesamte Testablaufplan gestaltet werden.
Um einen strukturellen Ablauf für die Entwicklung von E2E-Tests für ExplorViz sicherzustellen, orientieren wir uns an dem eingeführten Ablauf von Everett und passen diesen entsprechend unserer Anforderungen an [4]. Dabei fallen unter anderem die Elemente 1, 2, 4, 5, 6, 10 weg, da diese entweder bereits bearbeitet oder im Rahmen der Arbeit nicht notwendig sind. Mit ExplorViz ist die zu testende Anwendung bereits bekannt und ein (unternehmerisches) Risiko ist bei der Konzeption und Entwicklung nicht vorhanden, da diese im Rahmen einer Bachelorarbeit stattfinden. Weiterhin ist mit Alexander Krause-Glau ein Ansprechpartner, als Betreuer der Arbeit, vorhanden. Dieser weist Expertise mit der Anwendung ExplorViz auf. Mit dem Ziel eines Vergleichs von Erwartungswerten mit Ist-Werten einer öffentlich zugänglichen Anwendung müssen keine Testdaten zur Verfügung gestellt werden. Der gesamte Testablauf wird nicht als Punkt aufgeführt, da davon ausgegangen wird, dass die grundlegenden Aspekte von Punkt zehn durch die Inhalte dieser Arbeit abgedeckt sind. Es werden lediglich Erwartungswerte benötigt, welche innerhalb des Teams besprochen und in Form von Screenshots Dokumentiert wurden. Der von uns angestrebte Vorgang bzgl. der Konzeption der Tests sieht wie folgt aus:

Nr.	Vorgang
1	Umfang und Limitationen der Tests
2	Identifikation einer geeigneten Testumgebung
3	Identifikation einer geeigneten Test Strategie
4	Entwicklung der Tests

Zunächst wird definiert, welche Nutzerinteraktionen getestet werden sollen. Daraus werden anschließend die spezifischen Testszenarien konzipiert (Abbildung 3.1). Damit sind dann der Umfang und die Limitation für die E2E-Tests dieser Arbeit festgelegt. Im nächsten Schritt folgt die Identifikation eines geeigneten Tools, welches die Anforderungen der Testszenarien erfüllt. Die Teststrategie ist als E2E-Test, welcher in die Kategorie der Blackbox-Tests einzuordnen ist, bereits mit dem Thema der Arbeit festgelegt. Dieser steht in Wechselwirkung mit der Festlegung des Umfangs, den daraus resultierenden Testszenarien und dem Tool. Die Konzeption der E2E-Tests ist mit den ersten drei Punkten abgeschlossen und anschließend folgt die Entwicklung und Implementierung der Tests. Hierbei werden die Tests im ersten Schritt mit einer Demo von ExplorViz entwickelt und in einem zweiten Schritt auf die Vollversion übertragen. Die Demoversion bietet dieselben Interaktionsmöglichkeiten, ist jedoch technisch reduziert und bietet sich entsprechend für die reine Entwicklung von Testszenerien an.

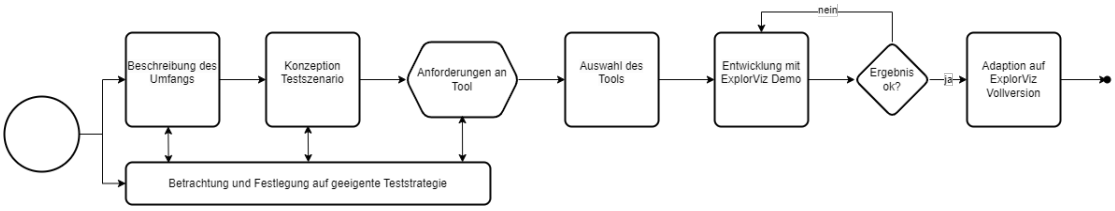

Abbildung 3.1. Ablauf der Konzeption und Implementierung

3.1. Konzeption

In diesem Abschnitt wird die Konzeption der E2E-Tests dargelegt. Dabei werden in Abschnitt 3.1.1 der Umfang und die Limitation der Tests beschrieben und anschließend ein exemplarisches Testszenario ausgewählt. In Abschnitt 3.1.2 folgt die tabellarische Gegenüberstellung der bereits vorgestellten Tools und die Auswahl. Weiterhin wird auf den Punkt *Identifikation einer geeigneten Teststrategie* eingegangen. Hierbei wird beschrieben, was für eine Art von E2E-Test durchgeführt werden soll.

3.1.1. Umfang und Limitation der E2E-Tests

ExplorViz ist architektonisch in die Bereiche Monitoring, Analyse und Visualisierung unterteilt (siehe Abbildung 2.1). Über den Bereich der Visualisierung findet die Interaktion des Nutzers mit ExplorViz statt. Eine Funktion von E2E-Tests ist die Simulation der Anwendung durch einen Nutzer, wodurch die Korrektheit des Interaktionergebnisses des Nutzers überprüft werden kann. Dargestellt wird die Simulation durch den Input eines Nutzers und Output der Anwendung. Infolgedessen bezieht sich der Umfang der zu konzipierenden E2E-Tests auf die Interaktion des Nutzers mit ExplorViz. Dabei soll der Output, der durch die Nutzeraktion generiert wird, mit einem Erwartungswert verglichen werden. Entsprechend müssen Testszenarien ausgewählt werden, welche diese Anforderung widerspiegeln. Weiterhin sollen durch die Tests mögliche Bugs aufgedeckt werden, die bei Arbeiten im Backend oder auch Frontend entstehen können. Ein Beispiel wären Änderung in einer Funktion.

Daraus könnte resultieren, dass die Funktion nicht mehr funktioniert und entsprechend kein Output generiert wird oder dass ein Output generiert wird, der nicht dem Erwartungswert entspricht. Die explizite Kommunikation im Backend oder Funktionen andere Module, welche im Backend laufen, sollen nicht getestet werden. Die E2E-Tests sollen sich lediglich auf den dem Nutzer zur Verfügung stehenden Frontend-Bereich von ExplorViz beziehen und keine Testfälle abbilden, für die Unit- und Integrationstests zuständig wären.
Da ExplorViz bei der Visualisierung der Anwendung unterschiedliche Modi zur Verfügung stellt (Desktop Computer: *on-screen* , Augmented Realitiy, Virtual Reality) [13], wurde zu Beginn im Team darüber gesprochen, für welchen Modus die Tests geschrieben werden sollen. Da der *on-screen* Modus als Gateway für die anderen Modi fungiert und die Rendering-Implementierung bei allen Modi gleich ist [11], wurde sich darauf geeinigt, dass in diesem Modus die Tests geschrieben werden sollen. Damit wird eine grundlegende Abdeckung aller Modi durch die Tests erreicht, da Fehler sich in der Visualisierung *on-screen* auf die anderen Modi übertragen. Es könnte für zukünftige Arbeiten jedoch diskutiert werden, ob Tests innerhalb der anderen Modi sinnvoll wären, da es durch die Nutzung des *on-screen* Modus als Gateway auch zu Übermittlungsfehlern kommen könnte.
Innerhalb einer Softwarelandschaft stehen dem Nutzer unterschiedliche Interaktionsmöglichkeiten zur Verfügung, von welchen einige exemplarisch Möglichkeiten nachfolgend aufgelistet werden. Wie bereits beschrieben, können die unterschiedlichen Komponenten der Softwarestadt markiert (Highlighting) werden. Weiterhin können zusätzliche Informationen, wie zum Beispiel instanziierte Objekte, dargestellt werden. Mit der Nutzung der Softwarestadt wird es auch ermöglicht, die einzelnen Bestandteile dieser näher zu betrachten. Die Bestandteile stellen unterschiedliche Komponenten der Anwendung dar. Ein Beispiel wäre die Repräsentation von Klassen der Anwendung durch Gebäude in der Softwarestadt [10](Abbildung 3.2 (A)). Mit einem Doppelklick werden die unterschiedlichen Subkomponenten sichtbar (Abbildung 3.2 (B)). Durch einen Rechtsklick wird ein Kontextmenü geöffnet (Abbildung 3.2 (C)). Mit diesem kann man beispielsweise die Ansicht wieder auf den Anfangszustand zurücksetzen. In Abbildung 3.2 (D) wird ein kurzlebiger Ping gezeigt, der durch einen Klick auf das Mausrad erzeugt wird. Dieser soll der Koordination dienen [11]. Weiterhin wird im Interface ein Menü-Button (Abbildung 3.2 (E)) angezeigt oder es kann über eine Timeline die Laufzeit der Anwendung eingesehen werden (Abbildung 3.2 (F)). Zudem ist Kameraposition frei beweglich. Das schließt auch ein hinein und hinaus Zoomen mit ein.[1]

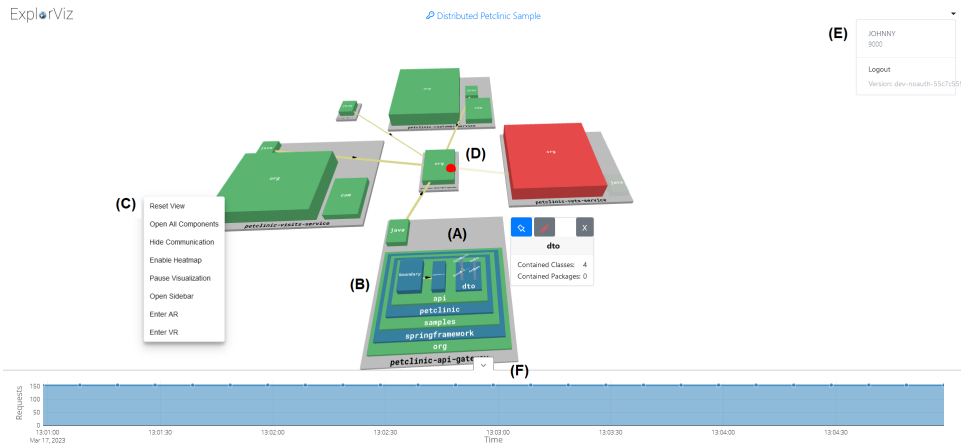

Abbildung 3.2. Beispielfunktionen

[1] https://explorviz.dev/

Eine exemplarische Auswahl der aufgezählten Funktionen gilt es mit den E2E-Tests zu prüfen. Zusammenfassend ergaben sich im Team folgende Funktionen, die in Testszenarien eingebunden werden:

Funktionen
Visualisierung wird angezeigt
Markieren einer Komponenten der CM mittels eines einfachen Klicks
Aufdecken des Popups der CM mittels einer Bewegung auf die Komponente
Aufdecken des Menüs mittels Rechtsklick

Bei den Test wurde darauf geachtet möglichst viele Funktionalitäten abzudecken. So wurde berücksichtigt, dass durch die Funktionen eine mögliche Interaktion mit den gerenderten Elementen stattfindet oder ein neues Element dargestellt wird (in Bezug auf das Aufdecken des Menüs). Da die Elemente in Echtzeit auf Basis der gesammelten Daten im Canvas gezeichnet werden, erfolgt gleichzeitig eine Überprüfung des Datenflusses und Zusammenspiels der Komponenten im Backend. Werden beispielsweise Änderungen im Backend durchgeführt und es kommt zu Bugs in der Visualisierung, hat man dadurch einen Hinweis, ob die Änderungen der Ursprung des Bugs ist. Da Three.js in ExploreViz genutzt wird und dies eine Anbindung an WebGL bedeutet, ist eine Interaktion über DOM-Elemente nicht ohne weiteres möglich. Diese Einschränkung muss letztlich auch bei der Auswahl des Tools zur Entwicklung des E2E-Tests für ExplorViz berücksichtigt werden.
Mit dem Ziel der Konzeption und Entwicklung geeigneter E2E-Tests auf Basis von Nutzerinteraktionen für ExplorViz bietet sich das funktionale Testen von Software an. Dieses hat das Ziel, Workflows der Software zu prüfen. Diese Prüfung bezieht sich auf den Output der Software und überprüft, ob dieser einem Erwartungswert entspricht [7]. Im Rahmen der Entwicklung von ExplorViz wird unter anderem GitHub als Versionierungssoftware genutzt.[2] Damit soll gewährleistet werden, dass unterschiedliche Versionen des Codes gesichert werden und dieser Code qualitativ hochwertig ist. Dies wird unter anderem dadurch erreicht, dass unterschiedliche Unit- und Integrationstests in der CI-Pipeline integriert wurden, welche Code zu prüfen, bevor dieser in den Produktivcode integriert wird. Um nun kontinuierliche E2E-Test zu ermöglichen, muss der in dieser Arbeit konzipierte E2E-Test ebenfalls in die CI-Pipeline integriert werden.
Zusammenfassend sollen funktionale E2E-Tests konzipiert werden, welche die Outputs der oben genannten Funktionen aus dem Bereich der Visualisierung überprüfen und Erwartungswerten gegenüberstellen. Da der Output in Form einer grafischen Oberfläche generiert wird, bietet sich für diese Gegenüberstellung der Vergleich von Screenshots an. Dieser Vorgang soll automatisiert durch die CI-Pipeline durchgeführt werden. Mit den gegebenen Informationen wird nun ein Testszenario exemplarisch beschrieben. Die vier Testszenarien unterscheiden sich lediglich in den eingebauten Funktionen, sodass die Beschreibung eines exemplarischen Testszenarios als Muster für die anderen Funktionen ausreicht. Bei der Erstellung des Szenarios wird sich erneut an dem von Everett beschriebenen Vorschlag zur Erstellung von Testszenarien orientiert[4]. Dadurch soll beschrieben werden, **wie** das Testen für die zu überprüfende Anwendung stattfinden soll. Im ersten Schritt sollen dazu praktische Abläufe gefunden werden, für welche Testszenarien geschrieben werden können. Diese kann man dann Stück für Stück erweitern. Dabei wird der Vorschlag gemacht, dass die Testszenarien folgende Informationen enthalten sollen[4]:

[2] https://github.com/explorviz

Nr.	Informationen zu Testszenarien
1	Eine einzigartige Kennnummer
2	Eine einzigartige Bezeichnung
3	Eine kurze Beschreibung
4	Phase der Entwicklung, in welcher das Testszenario Anwendung finden soll
5	Spezifische Testziele und wie diese gemessen werden sollen
6	Testdaten
7	Vorsätzliches Testtool
8	Beschreibung notwendiger Voraussetzungen
9	Beschreibung zur Unterbrechung des Tests
10	Beschreibung zur Wiederaufnahme des Tests
11	Festlegung der Testschritte

Auch hier finden nicht alle der aufgezählten Elemente Anwendung. Da nur lediglich für vier Funktionen Tests geschrieben werden, wird auf die Vergabe einer einzigartigen Nummer verzichtet. Weiterhin wird auf eine Darstellung der Phase der Entwicklung verzichtet, da sich ExplorViz in einem ständigen Entwicklungs- und Forschungsprozess befindet. Da die Auswahl des Tools im nächsten Kapitel erfolgt, wird auch darauf nicht näher eingegangen. Zudem werden die Elemente 9 und 10 der Aufzählung nicht näher beschrieben, da die Tests über die CI-Pipeline von GitHub abgewickelt werden sollen und dort eine einfache Möglichkeit zur Unterbrechung der Tests besteht.[3] Entsprechend werden die Elemente 2,3,5,6 und 11 inhaltlich im nächsten Abschnitt aufgegriffen und adaptiert. Dies geschieht an Hand des exemplarischen Testszenarios, in welchem das Markieren einer Komponenten durch einen E2E-Test getestet wird.

Exemplarisches Testszenario: Markieren einer Komponente

In diesem exemplarischen Testszenario wird das Markieren einer Komponente getests. Sobald der simulierte Nutzer in den von ExplorViz bereitgestellten Raum eintritt, in welchem die Softwarelandschaft visualisiert wird, soll innerhalb des Tests eine Komponente der dargestellten Module angeglickt werden, um diese zu markieren. Der Raum wird erreicht, indem die Seite des Webservers zur Auswahl der Softwarelandschaften aufgerufen wurde. Nach dem markieren wird ein Screenshot der abgebildeten Visualisierung gemacht und mit einem anderen Screenshot verglichen. Dazu wurde im Vorfeld ein Screenshot der Visualisierung aufgenommen, welcher als erwarteter Output dient. In Abbildung 3.3 folgt ein Ablaufdiagramm zur Verbildlichung der unterschiedlichen Schritte des Tests:

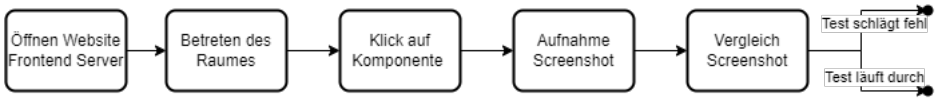

Abbildung 3.3. Ablaufdiagramm

Der erste Schritt ist das Öffnen der Website des Frontend Servers. Der Erwartungswert ist, dass die Website erreicht und geöffnet wird. Darauf folgt das Betreten des Raumes, was durch einen Klick auf einen Button realisiert wird. Der Raum soll daraufhin betreten und die Anwendung visualisiert werden. Anschließend folgt der Klick auf die ausgewählte Komponente der Softwarestadt. Der Erwartungswert ist die markierte Komponenten. Anschließend wird ein Screenshot aufgenommen. Abschließend folgt ein Vergleich der Screenshots, wobei erwartet wird, dass dieser ohne Fehler durchläuft.

[3] https://docs.github.com/en/actions/managing-workflow-runs/canceling-a-workflow

Für die Entwicklung des exemplarischen Szenarios wird im ersten Schritt die Demo von ExplorViz genutzt, da die Räume dort bereits zur Verfügung stehen und keine weitere Interaktion zwischen NutzerInnen und ExplorViz erforderlich ist. Sobald der Test funktioniert und nach Rücksprache im Team zufriedenstellende Ergebnisse liefert, folgt die Adaption des Tests mit der Vollversion von ExplorViz. Dazu wird das Spring Petclinic Example als Zielanwendung genutzt. Bei dem Spring Petclinic Example handelt es sich um eine Spring Boot Webanwendung, eine häufig genutzte Testsoftware[4]. Das Ziel dieses Testszenarios ist die Überprüfung der Visualisierung und der genannten Funktion zum Markieren von Komponenten. Wie bereits beschrieben, wird dies durch einen Soll/Ist-Vergleich dargestellt.

3.1.2. Analyse der Tools unter Berücksichtigung der Anforderungen

In diesem Abschnitt wird dargestellt, welche Anforderungen an die Tools sich aus den erarbeiteten Gegebenheiten der E2E-Tests ergeben. Diese Anforderungen werden dann genutzt, um die in Abschnitt 2.4 vorgestellten Tools auf ihre Brauchbarkeit zu überprüfen und eine Auswahl zu treffen. Wie bereits im vorherigen Kapitel festgestellt wurde, ist das Ziel die Simulation eines Nutzers, welcher eine Auswahl an unterschiedlichen Funktionen durchführt und damit testet. Diese Tests sollen lediglich im Frontend von ExplorViz stattfinden. Eine erste Anforderung an das Tool ergibt sich daraus, dass die E2E-Tests lediglich im Frontend von ExplorViz stattfinden sollen und einen Nutzer simulieren sollen. Erweitert wird diese Anforderung dadurch, dass ExplorViz Three.js nutzt, um 3D-Elemente auf einem Canvas zu zeichnen [11].[5] Daher muss das Tool einen Weg bieten, mit diesen Elementen über den Browser zu interagieren. Ansonsten wäre beispielsweise der Test der Funktion zum Markieren bestimmter Elemente der Softwarestadt nicht möglich. Die zweite Anforderung an das Tool ist das Aufnehmen von Bildern. Das ist notwendig, da Visualisierungen verglichen werden sollen. Weiterhin muss das Tool den Browsers im headless-Modus nutzen können. Bei einem headless-Browser steht das GUI nicht zur Verfügung.[6] Das Tool muss mit dem headless-Modus arbeiten können, da die E2E-Tests innerhalb eines Docker Containers auf einer CI-Pipeline laufen sollen. Hieraus ergibt sich auch, dass das gewählte Tool innerhalb eines Docker Containers laufen können muss. Wir legten in der Konzeption der Tests fest, dass diese den Output in Form von Screenshots mit einem Erwartungswert vergleichen, sodass die Funktion des Aufnehmen und Vergleichen von Screenshots die letzten beiden Anforderungen an das Tool stellt. Insgesamt lassen sich folgende Anforderungen zusammenfassen:

Nr	Anforderung
1	Öffnen des Browsers
2	Interaktion mit HTML-Elementen wie Buttons oder Canvas
3	Öffnen des Browsers im headless Modus inklusive Interaktion
4	Lauffähigkeit mittels Docker
5	Aufnahme von Screenshots
6	Vergleich von Screenshot

Die sechs erarbeiteten Anforderungen an das Tool werden nun genutzt, um zu überprüfen, inwiefern die in Abschnitt 2.4 vorgestellten Tools den Anforderungen entsprechen. Dabei wird *three musketeers* zwar einzeln betrachtet, jedoch in der Betrachtung von Mocha noch einmal aufgegriffen, da es sich lediglich um eine Bibliothek handelt, die mit einem Framework kombiniert werden muss. Weiterhin werden neben den spezifischen Anforderungen auch weiche Faktoren, wie die Übersichtlichkeit in der Dokumentation der Tools und vorhandene Kenntnisse über das Tool, berücksichtigt.

[4] https://spring.io/guides/gs/spring-boot/

[5] https://threejs.org/manual/#en/fundamentals

[6] https://windowsreport.com/headless-browser/

Gauge-Taiko

Da es sich bei Gauge[7] um ein Framework für Tests von Webanwendung handelt, wird eine Interaktion mit einem Browser angeboten. In Verbindung mit Taiko kann der Browser, durch eine Einstellung an der API zum Öffnen des Browsers, auch im headless Modus gestartet werden.[8] Des Weiteren wird durch die Taiko API auch eine Interaktion mit HTML-Elementen ermöglicht, mit welcher einfache Mausklicks simuliert werden können. Dabei kann sich der Mausklick auf ein HTML-Element beziehen, wie zum Beispiel einen Button. Außerdem gibt es die Möglichkeit auf einen Punkt mittels der Nutzung von Koordinaten zu klicken, welche sich an der HTML-Seite ausrichten[9]. Damit wird eine Interaktion mit Elementen ermöglicht, die auf einem Canvas gezeichnet werden. Zudem liefert Gauge mit der Initialisierung des Projektes auch ein Dockerfile. Dieses kann genutzt werden, um das Projekt, in welchem die Tests geschrieben wurden, auf einem Docker Container lauffähig zu machen[10]. Zudem bieten sowohl Gauge[11] als auch Taiko[12] Funktionen an, um Screenshots vom Browser zu machen. Screenshots können allerdings weder durch die Standardfunktionen von Gauge noch von Taiko verglichen werden. Es ergibt sich aus der Analyse, dass Anforderungen 1-5 von Gauge erfüllt werden und lediglich Anforderung 6 *Vergleich von Screenshots* nicht erfüllt ist.
Darüber hinaus bieten Gauge und Taiko eine sehr ausführliche Dokumentation. Diese reicht vom initialisieren des ersten Projektes über die Einbindung von Taiko in Gauge und die Nutzung der dort verfügbaren API bis hin zur Integration in Docker. Des Weiteren wurde bereits ein erster Versuch unternommen, E2E-Tests für ExplorViz über Gauge-Taiko zu realisieren.

three musketeers

Bei three musketeers[13] handelt es sich um eine Bibliothek zur Interaktion mit Three.js-Elementen. Die Anwendung nutzt dabei eine eigene Instanz, um über die Kommandozeile des Browsers mit den Objekten zu interagieren. Somit wird ein Weg angeboten, um mit auf dem Canvas gezeichneten Objekten zu interagieren. So kann beispielsweise ein Klick mit der Maus simuliert werden. Dafür wird lediglich die ID des Nodes benötigt.[14] Da sich three musketeers jedoch lediglich auf die im Canvas dargestellten Objekte bezieht, ist eine Interaktion mit anderen HTML-Elementen, wie zum Beispiel einem Button, nicht möglich. Weiterhin bietet three musketeers kein eigenes Dockerfile für ein Image an. Entsprechend muss dies in Verbindung mit dem genutzten Framework erstellt werden. Selbiges gilt für Screenshots. Hierfür bietet three musketeers keine eigenständige Funktionalität. Da die Interaktion mit den Canvas Objekten über die Kommandozeile stattfindet, ist dies auch im headless-Modus möglich.[15] Three musketeers erfüllt somit nur die Anforderung 3 *Öffnen/Interaktion mit dem Browser im headless Modus* und Anforderung 4 *Lauffähigkeit mittels Docker* und bietet eine bedingte Interaktion mit HTML-Elementen. Die restlichen Anforderungen werden durch Three musketerrs nicht erfüllt.
Three musketeers bietet eine gute Dokumentation. Die Möglichkeiten zur Interaktion mit den Elementen ist zwar begrenzt, so ist ein einfaches Drehen der Kameraperspektive nicht möglich. Die vorhandenen Möglichkeiten reichen jedoch aus, um einen geeigneten Test zu erstellen. Die Nutzung wirkt im ersten Moment ebenfalls nicht komplex. Es liegen jedoch keine Erfahrungswerte vor.

[7] https://docs.gauge.org/overview
[8] https://docs.taiko.dev/api/openbrowser/
[9] https://docs.taiko.dev/api/click/
[10] https://docs.gauge.org/howto/ci_cd/docker.html?os=macos&language=javascript&ide=vscode
[11] https://docs.gauge.org/writing-specifications.html?os=macos&language=javascript&ide=vscode
[12] https://docs.taiko.dev/api/screenshot/
[13] https://github.com/webgl/three-musketeers
[14] https://webgl.github.io/three-musketeers/module-click.html
[15] https://developer.chrome.com/blog/headless-chrome/

Mocha

Das auf Node.js basierte Test-Framework Mocha bietet eine Grundlage für JS basierte Tests. Grundlegend bietet Mocha zwar die Möglichkeit zum Öffnen einen Browsers (auch im headless Modus[16]), jedoch keine eigenständige Interaktion mit diesem. Dazu würde eine Bibliothek benötigt werden, welche diese Funktionalität bietet. Wie eingangs beschrieben kombinieren wir Mocha in der Auswahl mit der Bibliothek three musketeers. So werden die nicht erfüllten Voraussetzungen, wie das Öffnen eines Browsers von three musketeers, ausgeglichen und Mocha um Interaktionsmöglichkeiten mit Canvas-Elementen erweitert. Da weder Mocha noch three musketeers Interaktionsmöglichkeiten mit anderen HTML-Elementen bieten, kann der Klick auf einen Button beispielsweise nicht getestet werden. Die mit Mocha entwickelten Tests können auch über Docker durchgeführt werden.[17] Es wird keine Funktionalität zum Aufnehmen eines Screenshots von Mocha angeboten. Daraus resultiert, dass es kein integriertes Tool gibt, welches Screenshots miteinander vergleicht. Somit kann zusammengefasst werden, dass Mocha den Anforderungen 5 und 6 (Aufnahme und Vergleich von Screenshots) nicht gerecht wird und nur teilweise mit HTML-Elementen oder Canvas interagiert werden kann (Anforderung 2).
Mocha bietet eine mäßig übersichtliche Dokumentation an. So ist es auf den ersten Blick nicht ersichtlich, welche Möglichkeiten zur Interaktion mit Webanwendungen geboten werden. Hierbei wird lediglich auf die JS API verwiesen. Des Weiteren wird Mocha zwar für ein paar Unittests von ExplorViz genutzt, diese bieten jedoch kein Hilfsmaterial für die angestrebten E2E-Tests. Die vielen Möglichkeiten zum Testen sorgen dafür, dass Mocha auf den ersten Blick komplex erscheint.

3.1.3. Auswahl des Tools

Nachdem die Analyse der Tools in Bezug auf die Anforderungen abgeschlossen ist, folgt in diesem Abschnitt die Gegenüberstellung der unterschiedlichen Voraussetzungen in Bezug auf die Anforderungen und ein Vergleich der von uns bestimmten weichen Faktoren. Dazu erfolgt zunächst eine Gegenüberstellung der Erfüllung der Anforderungen der drei Tools:

Anforderung	**Gauge**	**three musketeers**	**Mocha**
Öffnen des Browsers	ja	nein	ja
Interaktion mit HTML-Elementen wie Buttons oder Canvas	ja	teilweise	teilweise
Öffnen/Interaktion mit dem Browsers im headless Modus	ja	ja	ja
Lauffähigkeit mittels Docker	ja	ja	ja
Aufnahme von Screenshots	ja	nein	nein
Vergleich von Screenshot	nein	nein	nein

Der Tabelle ist zu entnehmen, dass keines der vorgestellten Tools allen Anforderungen gerecht wird. So ist ein Vergleich der Screenshots mit keinem der Tools möglich. Es müsste für jedes Tool eine Erweiterung integriert werden, welche einen Vergleich von Screenshots ermöglicht. Eine Möglichkeit dafür wäre die Bibliothek Pixelmatch[18]. Diese ermöglicht einen einfachen Vergleich von Bildern anhand der Pixel. Auffällig ist ebenfalls, dass lediglich Gauge mit der Bibliothek Taiko den restlichen Anforderungen gerecht wird und auch eine problemlose Interaktion mit HTML-Elementen bietet.

[16] https://github.com/mochajs/mocha/wiki
[17] https://docs.docker.com/language/nodejs/run-tests/
[18] https://www.npmjs.com/package/pixelmatch

Außerdem würde eine sehr übersichtliche Dokumentation für eine gute Orientierung bei der Nutzung von Gauge-Taiko sorgen. Three museketeers bietet zwar ebenfalls eine gute Dokumentation an, wurde jedoch zuletzt 2019 aktualisiert. Dadurch besteht die Gefahr, dass Funktionen aus der Bibliothek nicht mehr korrekt funktionieren oder gar keine Kompatibilität mit den von ExplorViz genutzten unterschiedlichen Three.js-Versionen besteht. die schlechte Übersicht der Mocha-Dokumentation hebt die Vorteile von three musketeers an dieser Stelle allerdings wieder auf. Ein weiterer Aspekt sind die bereits vorhandenen Daten aus einem älteren Projektes zur Erstellung eines E2E-Tests mit Gauge-Taiko. Auf diese Daten könnte bei der Wahl von Gauge-Taiko zurückgegriffen werden, was den Entwicklungsprozess der Tests vereinfachen könnte. Abschließend haben wir uns auf Basis der genannten Punkte für die Nutzung von Gauge-Taiko entschieden.

Kapitel 4

Implementierung der Tests

In diesem Kapitel folgt die Beschreibung der Implementierung der Tests mit dem zuvor ausgewähltem Tool Gauge und der dort verwendeten Bibliothek Taiko. Im ersten Schritt folgt die Implementierung der beschriebenen Testfälle. Diese werden im Zusammenhang mit der Demoversion von ExplorViz entwickelt, da dies die technischen Hürden verringert und auf eine Interaktion mit einer durch ExplorViz beobachteten Anwendung verzichtet werden kann. Sind die Tests dort funktionsfähig implementiert, folgt die Adaption der Tests auf die Vollversion. Dazu wird das Spring Petclinic Example, eine Webanwendung, als Zielanwendung verwendet werden. Anschließend folgt die Erstellung eines Docker Images. Dadurch soll die Automatisierung der Tests ermöglicht werden. Dieses Image wird anschließend genutzt, um die Tests auf der CI-Pipeline von GitHub, GitHub Actions, zu integrieren.

4.1. Implementierung mit Gauge-Taiko

Gauge arbeitet bei der Erstellung von Tests mit Projekten. Entsprechend erfolgt im ersten Schritt der Testentwicklung die Initialisierung des Projektes. Dort folgt auch die Auswahl der für die Tests verwendeten Sprache. Nach der Initialisierung des Projektes ergibt sich eine feste Ordnerstruktur (Abbildung 4.1). Die Ordner *spec* und *tests* spielen dabei eine wichtige Rolle, da dort die Dateien abgelegt werden, welche die Tests enthalten. Die Bibliothek Taiko muss für das Projekt nicht importiert werden, da diese bereits bei der Initialisierung des Projektes durch Gauge geliefert wird. Sie ist bereits als Bibliothek in *node_modules* eingebunden.

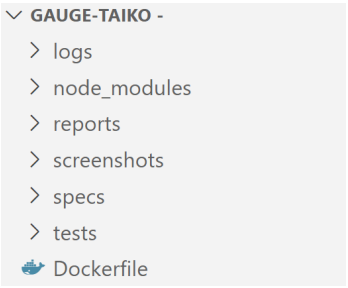

Abbildung 4.1. Gauge Ordnerstruktur

Wie in Abschnitt 2.4.1 bereits beschrieben, sind zwei Kernelemente bei der Implementierung von Tests mit Gauge wichtig. Einmal die spec-Datei, welche die Steps des Tests enthält (Abbildung 4.1 Specs Ordner) und die Realisierung der in der spec-Datei definierten Steps (Abbildung 4.1 tests Ordner). Da die Steps und ihre Implementierung auf zwei Dateien aufgeteilt werden, wird bei der Implementierung wie folgt vorgegangen: Zunächst wird das Testszenario, welches in Abschnitt 3.1.1 beschrieben wurde, auf Gauge übertragen und mit der spec-Datei dargestellt. Innerhalb der spec-

Datei folgt eine Unterteilung der Steps.[1] Diese gehören entweder zu einem Szenario, also dem Test als solchen, einem Kontext, dabei handelt es sich um Voraussetzungen, welche für das Szenario benötigt werden, oder werden als *Tear Down Steps* bezeichnet, welche im Anschluss eines Szenarios durchgeführt werden. Weiterhin benötigt jede spec-Datei eine entsprechende Überschrift, welche mit # gekennzeichnet wird (Listing 4.1 Zeile 1). Innerhalb einer spec-Datei können mehrere Szenarien durchgeführt werden. Dabei besitzt jedes Szenario einen einzigartigen Namen. Dieser wird mit ## kenntlich gemacht (Listing 4.1 Zeile 5). Die einzelnen Steps eines Szenarios werden mit einem * aufgeführt (Listing 4.1 Zeile 6). Die in Abbildung 3.3 dargestellten Schritte, können Listing 4.1 in Form einer spec-Datei entnommen werden.

Listing 4.1. Beispiel Gauge Highlight Spec der Demoversion

```
1  # Hightlighting Test
2  This is a context step that runs before every scenario
3  * Open Frontend-Demo application
4
5  ## Hightlighting Szenario
6  * Click on
7  |Alias|Created|empty|
8  |-----|-------|-----|
9  |Distributed Petclinic Sample|15.11.2017, 16:54:04|button-svg-with-hover|
10 * Click button to dismiss timeline
11 * Click on Element and take screenshot "testHighlight"
12 * Compare "testHighlight" screenshot with "exHighlight" screenshot
```

Gauge bietet die Nutzung von Tabellen an (Listing 4.1 in Zeile 7 bis 9). Diese fungiert dabei als Platzhalter für die Tabelle, welche auf der Softwarelandschaft Seite von ExplorViz zu sehen ist und vereinfacht deren Handhabung. Diese können als Parameter an die implementierten Steps übergeben werden. Parameter können außerhalb von Tabellen durch Hochkommata an die als Funktion implementierten Steps übergeben werden.[2] Für ein klares Bild ohne Laufzeit-Timeline haben wir uns bei der Erstellung der spec-Datei dazu entschieden, einen Step zur Minimierung der Laufzeit-Timeline hinzuzufügen. Dieser wurde in Zeile 10 realisiert. Schritte drei und vier aus der aus Abschnitt 3.1.1 dargestellten Tabelle wurden in einem Step zusammengefasst, (Listing 4.1 Zeile 11).
Die in der spec-Datei beschriebenen Steps müssen als Funktionen implementiert werden. Dazu wird innerhalb des Ordners *tests*, welcher auf der Abbildung 4.1 abgebildet ist, eine *.js* Datei mit dem Namen step_implementation angelegt. Für jeden der in der spec-Datei angelegten Steps muss eine Funktion implementiert werden. Die grundlegende Syntax kann in Listing 4.2 eingesehen werden.

Listing 4.2. Beispiel Step Implementation

```
1  const{
2     openBrowser, closedBrowser, goto, screenshot, click, evaluate, button, $,
3     mouseAction, righClick, title, openTab} = require('taiko');
4
5  step("This␣is␣a␣example␣step␣implementation␣with␣parameter␣<param>", async
6  function(parm){
7     // some code;
8     await click(param);
9  })
```

[1] https://docs.gauge.org/writing-specifications.html?os=macos&language=javascript&ide=vscode

[2] https://docs.gauge.org/writing-specifications.html?os=macos&language=javascript&ide=vscode#parameters-1

Bei der Implementierung der Schritte kann auf die Taiko-Bibliothek zurückgegriffen werden. Dazu muss mit dem Schlüsselwort *require('taiko')* die gewünschte API einer Konstante zugewiesen werden. Danach ist eine Nutzung innerhalb der Funktion möglich (Listing 4.2 Zeile 7). Der Step als solcher findet sich in der Beschreibung der Funktion wieder (Zeile 5 Listing 4.2). Bei der Implementierung der Steps aus dem Szenario zum Markieren einer Komponente stellte sich die Problematik heraus, dass die Taiko API nicht auf die Canvas-Elemente zugreifen konnte. Aus diesem Grund haben wir mit der Durchführung mehrerer Testläufe die Koordinaten eines Elements herausgefiltert. Wie Listing 4.3 darstellt, beziehen wir uns innerhalb der Funktion auf das Canvas Objekt. Damit wollen wir sicherstellten, dass sich bei den Tests immer am Canvas orientiert wird und nicht am Browser.

Listing 4.3. Realisierung Screenshot

```
1  step("Click␣on␣Element␣and␣take␣screenshot␣<screenshotFileName>",
2  async (screenshotFileName) => {
3     let pos_array = await evaluate($("canvas"), (canvas) => {
4        const canvasBCR = canvas.getBoundingClientRect();
5        return [canvasBCR.top + 850, canvasBCR.left + 450];
6     });
7
8     let canvas_x = pos_array[0];
9     let canvas_y = pos_array[1];
10    await click({ x: canvas_x, y: canvas_y});
11    await screenshot({path: "screenshots/test/" + screenshotFileName + ".png"});
12    }
13 );
```

Die Wahl auf das in Abbildung 4.2 (A) abgebildete rote Element fiel willkürlich. Da Gauge-Taiko die Anforderung des Vergleichs von Screenshots nicht erfüllt, wurde die Bibliothekt Pixelmatch genutzt. Pixelmatch färbt die nicht übereinstimmenden Pixel der Vergleichsbilder farbig ein.[3] Dieser Prozess wird in Abbildung 4.2 ersichtlich, in welcher (A) und (B) in (C) verglichen werden, wobei (C) das Vergleichsbild darstellt. Dieses Bild wird in einen Bericht eingebunden, welcher am Ende des Tests durch Gauge generiert wird. Weiterhin wird eine Nummer an Pixeln zurückgegeben, welche nicht übereinstimmen. Dadurch kann ein Vergleich erzeugt werden. Es werden die falschen Pixel aus dem Abgleich vom gemachten Bild mit dem Erwartungswert mit den Pixeln des Erwartungswertes verglichen. Überschreiten die falschen Pixel eine Grenze auf Basis des Gesamtwerts der Pixel, wird der Test als negativ gewertet. Bei dem Gesamtwert handelt es sich um 5% der Pixel des Erwartungswertes, welcher nicht mit falschen Pixeln überschritten werden darf. Der Wert von 5% ergab sich aus einer Reihe von Annäherungstests.

[3] https://github.com/mapbox/pixelmatch

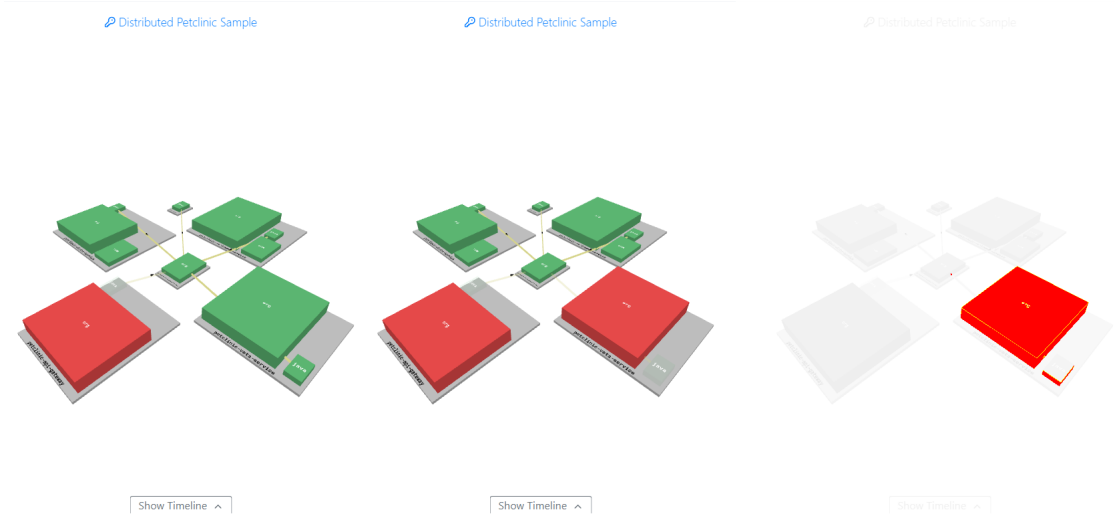

Abbildung 4.2. Beispiel Pixelmatch

Die Abbildung 4.2 wurde mit der Demoversion von ExplorViz aufgenommen. Bei dieser handelt es sich um das Frontend von ExplorViz, welches mit einem gemockten Backend und mehreren Zielanwendungen gestartet wird.[4] Dadurch wird eine weniger komplexe Umgebung angeboten, um sich mit ExplorViz vertraut zu machen. Aufgrund dessen findet erste Implementierung der Tests auf Basis der Demo statt, da sich durch die Reduzierung der technischen Komponente auf die Erstellung der Tests konzentriert werden kann. Die Demo bietet mehrere Landscapes zur Interaktion an, unter anderem auch das Petclinic Example. Da wir uns im späteren Verlauf bei der Vollversion auf das Spring Petclinic Example[5] als Zielanwendung geeinigt haben, nutzen wir dieses auch bei der Demoversion für die Tests. Damit wird die spätere Adaption der Tests auf die Vollversion erleichtert.
Mit der Implementierung der Steps als Funktionen und dem erfolgreichen Testen des in Abschnitt 3.1.1 beschriebenen Testszenarios wurden auch die anderen Szenarien implementiert. Somit standen im Zusammenhang mit der ExplorViz Demo folgende Testszenarien aus dem Abschnitt 3.1.1 für eine Besprechung im Team zur Verfügung:

Szenarien
Visualisierung wird angezeigt
Markieren einer Komponenten der CM mittels eines einfachen Klicks
Aufdecken des Popups der CM mittels einer Bewegung auf die Komponente
Aufdecken des Menüs mittels Rechtsklick

Bis auf kleinere technische Anforderungen, welche zum Zeitpunkt der Vorstellung der Tests noch nicht klar waren, wurden die Szenarien und die Implementierung dieser mit Gauge-Taiko als erfolgreich gewertet. Bei einer der technischen Anforderungen handelte es sich beispielsweise um die Durchführung und Darstellung der Tests über die Konsole. Dieser Punkt konnte bereits bei der Besprechung durch eine gemeinsame Recherche gelöst werden. Ein weiterer Punkt war die Durchführung der Tests im headless-Modus des Browsers. Bei der Erstellung war die Durchführung der Tests im headless-Modus noch nicht getestet worden. Auch diese Problematik konnte bei der Besprechung und Vorführung im Team gelöst werden. So wurde lediglich eine Option im Code umgestellt.

[4] https://explorviz.dev/2-getting-started/
[5] https://github.com/spring-projects/spring-petclinic

Bei der Adaptierung zur Vollversion kamen mehrere Herausforderungen auf. So musste zum einen ein weiterer Kontext-Step eingeführt werden. Bei der Nutzung der Vollversion von ExplorViz steht keine gemockte Zielanwendung zur Verfügung. Wie im Verlauf des Kapitels bereits beschrieben, nutzen wir im Zusammenhang mit der Vollversion von ExplorViz die Spring Petclinic Webanwendung als Zielanwendung. Da ExplorViz derzeit lediglich eine dynamische Überwachung einer Anwendung ermöglicht [12] (entsprechend muss die Zielanwendung gestartet und genutzt werden), erfolgt über den Kontext-Step die Interaktion mit der Spring Petclinic Webanwendung als Voraussetzung für die Tests. Der Gegenstand der Tests sind die Funktionen, die ExplorViz im Frontend bietet. Daher beinhaltet die Interaktion mit der Petclinic Webanwendung lediglich das Aufrufen der Seite und keine weiteren Interaktionen mit der Anwendung. Es wäre zwar möglich noch weitere Interaktionen mit der Webanwendung durchzuführen, was die beanspruchte Zeit des Tests jedoch verlängern würde. Dadurch erfolgt die Darstellung einer Komponente im Frontend (Abbildung 4.3).

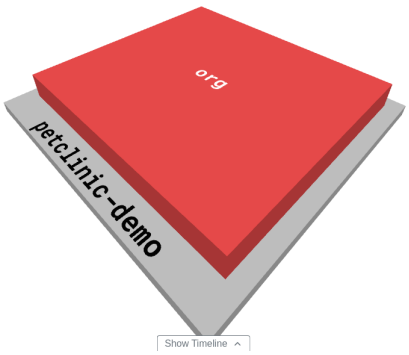

Abbildung 4.3. Softwarestadt der Spring Petlinic nach Öffnen der Website

Die Komponente auf der Darstellung ist bereits durch die Nutzung Funktion zum Markieren von Komponenten markiert. Bei der Implementierung der Kontext-Steps entschieden wir uns dafür, einen Browser und einen Tab innerhalb des Browsers zu öffnen (Listing 4.4).

Listing 4.4. Implementierung Kontext Steps

```
1  step("Open petclinic-example page", async function () {
2     await goto("http://host.docker.internal:18080");
3  });
4
5  step("Open Explorviz landscape", async function () {
6     await openTab("http://explorviz-frontend/landscapes");
7  });
```

Eine weitere Schwierigkeit war die Auswahl der Erwartungswerte. Bei den Erwartungswerten handelt es sich um Screenshots der erwarteten Visualisierung. Nachdem diese aufgenommen und eingefügt wurden ist aufgefallen, dass die Visualisierung der Zielanwendung nicht konsistent ist. So wird das in Abbildung 4.3 dargestellte Modul in unterschiedlichen Größen visualisiert. Nach Rücksprache im Team stellte sich heraus, dass die Ursache dafür im Backend von ExplorViz liegt. Auf Basis dessen wurde sich für eine Auswahl an Erwartungswerten im Team entschieden, welche die Visualisierung korrekt darstellen. Die Lösung der Problematik im Backend wird separat in einer anderen Arbeit

behandelt. Die Implementierung der Vollversion unterscheidet sich von der Demoversion lediglich in den eingeführten Kontext-Steps. Die Szenarien unterscheiden sich in der durchzuführenden Aktion und somit in der genutzten Taiko API. Beim Szenario *Markieren einer Komponente der Softwarestadt mittels eines einfachen Klicks* wurde die API zum Klick verwendet. In Listing 4.3 in Zeile 10 sieht man, dass die aus dem Canvas gezogenen Koordinaten für die genaue Bestimmung der Position des Klicks verwendet wurden. Für die Szenarien *Markieren einer Komponente der Softwarestadt mittels eines einfachen Klicks* und *Aufdecken des Pop-ups der Softwarestadt mittels einer Bewegung auf die Komponente* wurden dieselben Koordinaten verwendet. Bei dem Szenario *Aufdecken des Menüs mittels Rechtsklick* mussten die Koordinaten so angepasst werden, dass beim Rechtsklick keine Komponente ausgewählt wurde. Das Menü lässt sich nur öffnen, wenn der Rechtsklick auf weißem Untergrund erfolgt. Bei dem Szenario *Visualisierung wird angezeigt* muss nur ein Screenshot aufgenommen werden, da nur geprüft werden soll, ob die Visualisierung auch vorhanden ist. Mit der Implementierung der Szenarien als Tests wurden diese erneut im Team besprochen. Bei der Vorstellung gab es keine Probleme und somit konnte der nächste Arbeitsschritt begonnen werden. Dieser Arbeitsschritt besteht aus der Lauffähigkeit der Tests in Docker, welche im folgenden Abschnitt 4.2 dargestellt wird.

4.2. Lauffähigkeit in Docker

Nach der erfolgreichen Implementierung der Tests mittels Gauge-Taiko müssen diese nun in Docker integriert werden. Das ist notwendig, da die Tests automatisiert durchgeführt werden sollen. Entsprechend soll ein Container gestartet werden, welcher mit den anderen Containern innerhalb des ExplorViz-Netzwerks kommuniziert und die in Abschnitt 4.1 implementierten Tests durchführt.

Dazu muss im ersten Schritt geklärt werden, wie ExplorViz in Verbindung mit Docker aufgebaut ist. In Abschnitt 2.2.1 wurde dargestellt, dass die Aufgaben, die durch die in Abschnitt 2.2 genannten Module erledigt werden, durch diverse Container erfüllt werden. Um dies zu ermöglichen, wurde eine Docker-Compose-Datei erstellt. Diese beinhaltet unterschiedliche Services, welche die unterschiedlichen Module darstellen, welche wiederum die Container abbilden. Da Docker Container ohne weitere Konfiguration weitestgehend isoliert sind, wurde ein Netzwerk mit dem Namen *explorviz* erstellt. Alle innerhalb der durch die Docker-Compose-Datei erstellten Container befinden sich im benannten Netzwerk. Über dieses kommunizieren die Container miteinander, um den Datenfluss zu ermöglichen, der auf der Abbildung 2.1 zur Architektur von ExplorViz abgebildet ist. Durch diese Aufteilung der Module auf die Container wird es ermöglicht, ExplorViz auf verschiedenen Endgeräten zu nutzen. Die zur Verfügung gestellte Docker-Compose-Datei enthält alle notwendigen Services für die Nutzung von ExplorViz. Das beinhaltet jedoch nicht die Zielanwendung, die überwacht werden soll. Daraus folgt das Ziel für diesen Abschnitt. Dieses ist die Erstellung eines Images, welches die Tests enthält, wodurch ein Container gebaut werden kann, der die Tests automatisiert durchführt. Außerdem werden Services benötigt, welche die Spring Petclinic Anwendung und die Tests startet. Die dazugehörigen Container bettet man zur Ermöglichung der Kommunikation in das Netzwerk *explorviz* ein.
Für die Erstellung von Docker Images ist ein Dockerfile[6] notwendig. Dieses wird, wie Abbildung 4.1 entnommen werden kann, bei der Initialisierung des Gauge Projektes automatisch durch Gauge generiert. Als Basisimage wird Node vorgeschlagen, welches jedoch auf *node:18-bullseye* abgeändert werden muss, da der Bau des Image ansonsten nicht funktioniert. Für die Nutzung von ExplorViz als Webanwendung und die Durchführung der Tests mit Gauge-Taiko ist die Installation eines Browsers notwendig. Da es sich bei Gauge um ein Framework zum Test von Webanwendungen handelt, wird Chrome als Browser zur Installation und Nutzung vorgeschlagen und von uns genutzt. Weiterhin

[6] `https://docs.docker.com/engine/reference/builder/`

wird ein Nutzer angelegt, welcher die Tests durchführt. Zudem folgt die Installation notwendiger Anwendungen. Das Image wird im ersten Schritt lokal unter dem Namen *gauge-taiko* gespeichert und nicht auf Dockerhub[7] hochgeladen. Bei Dockerhub handelt es sich um eine Plattform zum Verteilen von Containerimages. Dazu stellt Docker ein Repository zur Verfügung, welches das Pushen und Pullen von Images ermöglicht. Der Schritt des Hochladens folgt nach der erfolgreichen Umsetzung der Tests mit Docker, damit das Image für den CI-Runner von GitHub Actions zur Verfügung steht und ggf. einfacher ausgetauscht werden kann. Außerdem wurde die zur Verfügung stehende Docker-Compose-Datei von ExplorViz angepasst und es wurde ein Service *e2e-testing* eingefügt. Dieser ist auf der **??** abgebildet und zeigt u.a. die Nutzung des zuvor erstellten Gauge-Taiko Images.

[7] `https://docs.docker.com/docker-hub/`

Listing 4.5. E2E-Test-Service in Docker-Compose-Datei

```
1  ## E2E Test ##
2  e2e-testing:
3     container_name: gauge-taiko
4     image: gauge-taiko
5     depends_on:
6        spring-petclinic:
7           condition: service_healthy
```

Weiterhin ist dem auf der Listing 4.5 abgebildeten *depends on* zu entnehmen, dass der Docker-Compose-Datei weitere Services hinzugefügt wurden. Bei diesen handelt es sich um die Services aus der Docker-Compose-Datei, welche zum Start der Spring Petclinic genutzt werden. Die Datei wird beim Pullen von ExplorViz als Demoanwendung zur Verfügung gestellt.[8] Bei der Docker-Compose-Datei werden mehrere Container gebaut und gestartet. Der Start der Container beruht auf einer innerhalb der Datei abgebildeten Reihenfolge. So fußt der Start des Adapter Service von ExplorViz unter anderem auf den Status *service_healthy* des Kafka Service Containers. Zum Erreichen des Healthy-Status wird zuvor ein Healthcheck[9] durchgeführt. Dieser führt einen Befehl innerhalb des Containers aus. Kann der Befehl der Kondition des Healthchecks entsprechend durchgeführt werden, erhält der Container den Status Healthy. Mit dieser Methode kann ein kontrollierter Start von Containern durchgeführt werden. Um die Tests korrekt durchführen zu können, ist es notwendig, dass ExplorViz und die Zielanwendung simultan ausgeführt werden. Zur Umsetzung des gewünschten Ablaufes wurde für die Spring Petclinic ein Healthcheck verwendet. Im Vorfeld müssen jedoch diverse Services gestartet werden und erfolgreich laufen. Zur Darstellung der Abhängigkeiten haben wir ein Ablaufdiagramm verwendet. Abbildung 4.4 zeigt den Weg der Abhängigkeiten zur Durchführung der Tests. So müssen im ersten Schritt die ExplorViz-Services erfolgreich gestartet werden. Dies beinhaltet unter anderem den frontend-dev Service. Läuft dieser, starten die Spring Petclinic Services mit dem Start des Petclinic Ocelot Agents. Läuft dieser ohne Probleme, so wird der Service Spring-Petclinic gestartet, für den ein Healthcheck geschrieben wurde. Dieser ruft die Webanwendung innerhalb des Containers auf und gibt *true* zurück, falls die Seite erfolgreich aufgerufen wurde. Andernfalls wird *false* zurückgegeben und der Healthcheck wird nach fünf Sekunden erneut durchgeführt. Diese Wiederholung kann sich bis zu zehn Mal wiederholen. Wird die Webanwendung erfolgreich aufgerufen, erhält der Container den Status Healhty und der e2e-testing Service wird gestartet.

[8] https://github.com/ExplorViz/deployment

[9] https://docs.docker.com/engine/reference/builder/#healthcheck

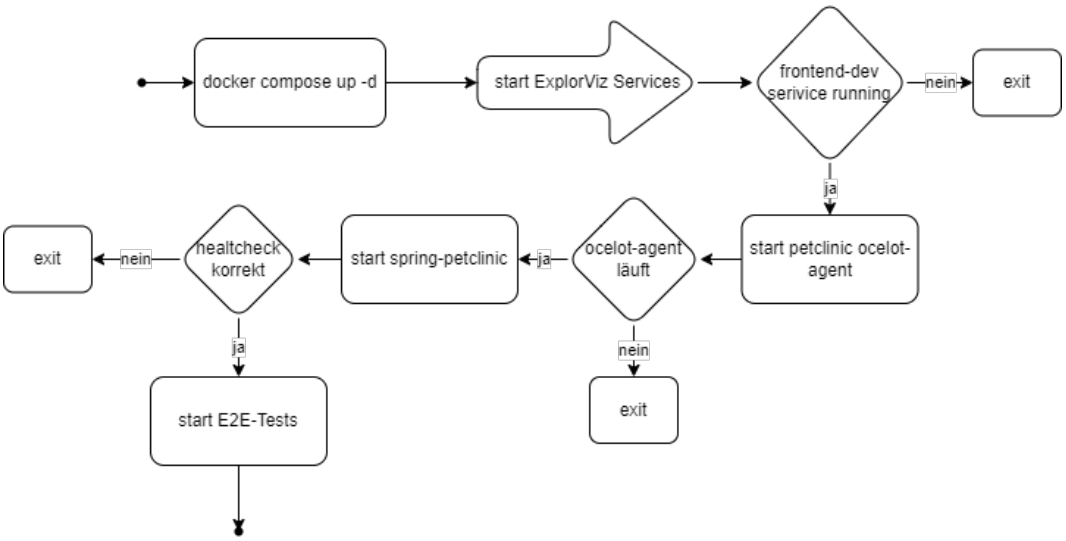

Abbildung 4.4. Ablaufdiagramm Docker Compose

Mit der Nutzung von Docker zur Durchführung der Tests musste der Zugriff auf die Landscapes, durchgeführt mit den Kontext-Steps, geändert werden. So konnte lokal auf dem Endgerät *http://localhost:8080* genutzt werden, um die Softwarelandschaften Seite des Frontend Servers im Browser aufzurufen. Die Seite bildet die unterschiedlichen Landscapes ab, mit denen interagiert werden kann. Da mit *localhost* das lokale Endgerät gemeint ist, welchem Zugriff auf die Seite gewährt wird, gilt dies nicht für den isolierten Docker-Container, der die Tests ausführt. Dieser befindet sich zwar im selben Netzwerk, wie die anderen Container und der lokale Host, bildet mit der Namensauflösung von localhost jedoch sich selbst als Host ab. Für diesen stehen jedoch keine Port zur Verfügung. Entsprechend wird der *host.docker.internal* Befehl genutzt, welcher zur internen IP des Hosts aufgelöst wird und damit einen Zugriff auf die Services anderer Container unter der IP des Docker-Hosts ermöglicht.[10] Mit dem Host ist jenes System gemeint, auf welchem Docker durchgeführt wird. Somit kann es sich dabei um das lokale Endgerät oder auch eine virtuelle Maschine in Form einer CI-Pipeline handeln. Mit der Änderung der Docker-Compose-Datei und der entsprechenden Steps, durch welche die Anwendungen geöffnet werden, sind die Tests auf Docker lauffähig.
Mit der Lauffähigkeit der Tests wird das Image auf DockerHub hochgeladen. Damit wird eine einfachere Austauschbarkeit des Tests gewährleistet und es ergibt sich eine Zeitersparnis. So muss nicht bei jedem Durchlauf das Image neu gebaut werden. Dazu wird dem Image mit *docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]*[11] der Tag *gauge-taiko1_0* zugewiesen, sodass schlussendlich mit dem Ausdruck *image: sousagod/gauge-taiko:gauge-taiko1_0* das Image in der Docker-Compose-Datei genutzt werden kann.

4.3. Integration des Docker Image in GitHub Actions

Wir setzten in Abschnitt 1.2.4 das Ziel, dass die Tests mittels einer CI-Pipeline durchgeführt werden sollen. Hierzu wird GitHub Actions, die CI-Pipeline von GitHub, genutzt. Es wurde sich im Vorfeld

[10] https://docs.docker.com/desktop/networking/#i-want-to-connect-from-a-container-to-a-service-on-the-host
[11] https://docs.docker.com/engine/reference/commandline/tag/

auf GitHub geeinigt, da dies als Mirrow der GitLab Repositories genutzt wird und somit für den Entwicklungsprozess von Tests eine sichere Umgebung zur Verfügung steht.
Für GitHub Actions ist es notwendig eine yml-Datei anzulegen, welche die Schritte zum durchzuführenden Workflow beinhaltet. Für diese muss, nachdem das dazugehörige Repository erstellt wurde, die Ordnerstruktur *.github/workflow* angelegt werden. Dort wird die Datei abgelegt.[12] Die yml-Datei wurde entsprechend der in Abschnitt 2.3.1 beschriebenen Anforderungen erstellt.

Listing 4.6. GitHub Workflow

```
1  name: E2E-Test with Petclinic Example
2  on: push
3  jobs:
4     test:
5        runs-on: ubuntu-latest
6        steps:
7           - uses: actions/checkout@v3
8           - name: Build and test
9           run: COMPOSE_PROFILES=env,jvm docker compose up --attach gauge-taiko
10          - name: get result
11          id: exit-value
12          run: |
13          docker wait gauge-taiko
14          docker logs gauge-taiko
15          t=$(docker inspect --format='{{.State.ExitCode}}' gauge-taiko)
16          echo "EXIT_CODE=$t" >> $GITHUB_ENV
17          - name: Test succeded
18          if: ${{ env.EXIT_CODE == 0 }}
19          run: echo "Test succeded!"
20          - name: Test failed
21          if: ${{ env.EXIT_CODE > 0 }}
22          run: |
23          echo "Test failed with exit code ${{ env.EXIT_CODE }}"
24          docker compose down -v
25          exit 1
26          - name: Stop containers
27          if: ${{ success() }}
28          run: docker compose down -v
```

In Zeile 2 auf Listing 4.6 ist zu erkennen, dass die CI-Pipeline den Workflow bei einem Push in das Repository durchführt. Der Job mit der Bezeichnung *test* definiert. Dort haben wir definiert dass die CI-Pipeline die benötigten Container, inklusive des Testcontainers *gauge-taiko*, bauen und starten soll. Innerhalb des Jobs haben wir vier Steps festgelegt, welche von der CI-Pipeline durchgeführt werden. Der erste Step in Zeile 7 *actions/checkout@v3* ermöglicht der CI-Pipeline den Zugriff auf die Daten innerhalb des Repositorys.[13] Der in Zeile 8 definierte Step *Build and test* baut die Container und sorgt entsprechend für die Durchführung des Tests. Für den Bau und Start der Container nutzen wir den *COMPOSE_PROFILES=env,native docker compose up* Befehl, da dadurch jene Container, welche für das Monitoring von ExplorViz zu Forschungszwecken genutzt werden, nicht gestartet werden.

[12] https://docs.github.com/en/actions/quickstart
[13] https://github.com/actions/checkout

Dieses Vorgehen führt zu einer Zeitersparnis, da weniger Container gepullt und gebaut werden müssen. Die Option *–attach gauge-taiko* haben wir gewählt, um den Docker Compose Befehl an den Service Output des Testcontainers *gauge-taiko* zu binden. Das war notwendig, da das Terminal der CI-Pipeline mit der Nutzung des *docker compose up -d* Befehls den Output der unterschiedlichen Container angezeigt hat. Da die Container jedoch die ganze Zeit laufen und Output generieren, hängt sich die CI-Pipeline damit auf. Der Testconainer fährt sich nach der Durchführung der Tests selbstständig wieder herunter. Aus diesem Grund umgingen wir mit dem beschriebenen Workaround das Aufhängen der CI-Pipeline. Mit dem Step *get result* in Zeile zehn weist die CI-Pipeline den ExitCode des Testcontainers *gauge-taiko* einer Variablen zu. Mit Zeile 13 wird erreicht, dass die CI-Pipeline die nächsten Schritte erst ausführt, nachdem der Testcontainer seinen Durchlauf beendet hat. Zeile 14 zeigt die entsprechenden Logs der ausgeführten Tests auf der Konsole. In Zeile 16 wird der Ergebniswert des Befehls *docker inspect –format='.State.ExitCode' gauge-taiko*, der der Variabel *t* zugewiesen wurde, auf eine Umgebungsvariable übertragen. Auf diese kann innerhalb des Workflows durch nachfolgende Steps zugegriffen werden.[14] Danach unterteilt sich der Job in zwei mögliche Resultate. Entweder wird mit der If-Bedingung[15] in Zeile 18 festgestellt, dass die E2E-Tests erfolgreich waren, dann wird "Test succededäusgegeben (Zeile 19) oder die Tests sind fehlgeschlagen und es wird eine entsprechende Fehlermeldung ausgegeben (Zeile 23). Die Feststellung, ob die Tests erfolgreich waren, erfolgt über den *EXIT CODE* des Testcontainers.[16] Unabhängig vom Testergebnis führt der Step *Stop containers* in Zeile 26 den Befehl *docker compose down -v* zum Herunterfahren der Container aus.

Listing 4.7. Finaler Service für das E2E-Testing in der Docker-Compose-Datei

```
1  e2e-testing:
2     container_name: gauge-taiko
3     image: gauge-taiko
4     depends_on:
5        spring-petclinic:
6           condition: service_healthy
7     extra_hosts:
8        - "host.docker.internal:host-gateway"
```

Neben der Erstellung der yml-Datei haben wir die Docker-Compose-Datei erneut angepasst. Da wir uns bei dem Runner für Ubuntu entschieden haben und bei Linux basierten Systemen *host.docker.internal* nicht ohne weiteres zur Auflösung der IP-Adresse des Docker-Hosts zur Verfügung steht, mussten wir mit *extra_host* das interne Gateway des Testcontainers mappen (Listing 4.7 Zeile 8). Damit der Code innerhalb der Tests nicht geändert werden muss, haben wir uns für den Hostnamen *host.docker.internal* entschieden. Mit *extra_host*[17] wird ein Eintrag in der Netzwerkinterface Konfiguration des Containers generiert, welcher den Hostname auf das Gateway des Containers mapped.[18] Neben den eben beschriebenen Erweiterungen des Services *e2e-testing* mussten wir auch weitere Einträge in der Docker-Compose-Datei ändern.

[14] https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions#setting-an-environment-variable

[15] https://docs.github.com/de/actions/learn-github-actions/expressions

[16] https://komodor.com/learn/exit-codes-in-containers-and-kubernetes-the-complete-guide/

[17] https://github.com/compose-spec/compose-spec/blob/master/spec.md#extra_hosts

[18] https://medium.com/@TimvanBaarsen/how-to-connect-to-the-docker-host-from-inside-a-docker-container-112b4c71bc66

Listing 4.8. Bind Mount

```
1  ## kurze Schreibweise ##
2  volumes:
3     - kafka-data:/var/lib/kafka/data
4     - kafka-secrets:/etc/kafka/secrets
5     - ./generic/kafka/kafka_kraft_init.sh:/tmp/update_run.sh
6
7  ## lange Schreibweise ##
8  volumes:
9  - kafka-data:/var/lib/kafka/data
10 - kafka-secrets:/etc/kafka/secrets
11 - type: bind
12   source: /home/runner/work/E2E-Test/E2E-Test/generic/kafka/kafka_kraft_init.sh
13   target: /tmp/update_run.sh
```

So ist es der CI-Pipeline von GitHub nicht möglich, die Kurzschreibweise von bind mounts zu erkennen (Listing 4.8 Zeile 1). Bei der Nutzung dieser Schreibweise erkennt die CI-Pipeline die Datei als Ordner und nicht als Datei. Bei der langen Schreibweise[19] (Listing 4.8 Zeile 7) nutzt man die Schlüsselwörter *bind, source* und *target*. Weiterhin haben wir mit *chmod +x* die Zugriffsrechte auf die in **??** (A) in Zeile 54 abgebildete Datei *update_run.sh* geändert. Dies war notwendig, da die Tests auf einer Windowsmaschine entwickelt wurden, der Runner der CI-Pipeline jedoch auf Ubuntu und damit einem Linux-System basiert. Dieses benötigt die entsprechenden Zugriffsrechte auf die Datei, um diese innerhalb des Bind Mounts ausführen zu können. Anschließend konnte die CI-Pipeline den E2E-Test, welcher die Prüfung der Szenarien beinhaltet, erfolgreich durchführen, was die lange Schreibweise rechtfertigt (Listing 4.8 Zeile 7).
Abschließend folgt in Abbildung 4.5 die Darstellung eines erfolgreich durchgeführten Testlaufes. Dieser nimmt vier bis fünf Minuten in Anspruch.

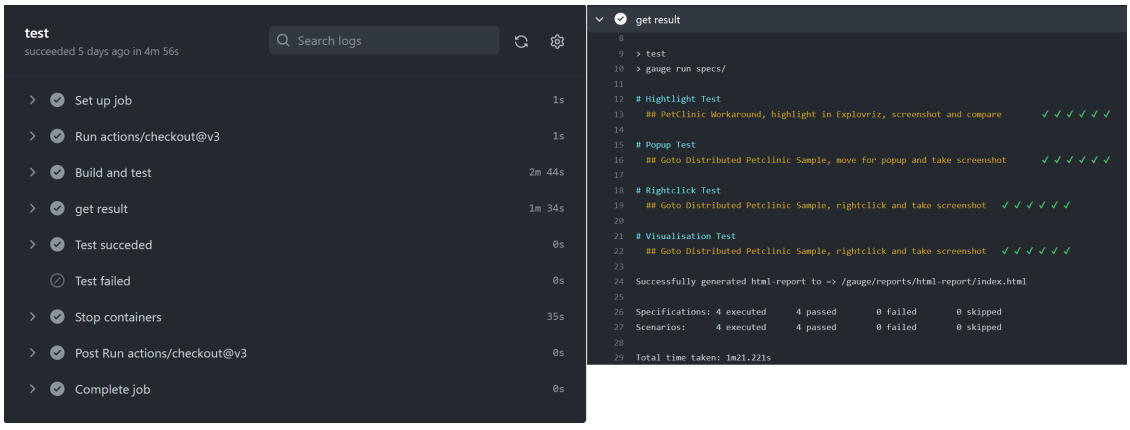

Abbildung 4.5. Erfolgreicher Testlauf

[19] https://docs.docker.com/compose/compose-file/compose-file-v3/#volumes

Kapitel 5

Evaluation

In diesem Kapitel folgt die Evaluation unseres Ansatzes. Entsprechend greifen wir das letzte Ziel der Arbeit, die exemplarische Anwendung in der Praxis, auf. Dazu werden zwei exemplarische Szenarien gegenübergestellt.

5.1. Exemplarische Anwendung in der Praxis

In Abschnitt 5.1.1 werden wir die E2E-Tests auf einem lokalen Gerät durchführen. Anschließend werden wir in Abschnitt 5.1.2 die Implementierung in GitHub Actions aufgreifen und die Tests automatisiert durchführen.

5.1.1. Szenario 1: Durchführung der E2E-Tests nach Bedarf

In diesem Szenario wird die manuelle Durchführung von E2E-Tests beschrieben. Dieser Testablauf zeichnet sich dadurch aus, dass nach Bedarf die mit Gauge entwickelten E2E-Tests lokal auf dem Gerät durchgeführt werden. Dazu startet die Testperson ExplorViz inklusive der Spring Petclinic als Zielanwendung. Danach folgt der Start der E2E-Tests mittels des *npm test* Befehls in der Konsole. Die Tests werden durch direkt Gauge ausgeführt , somit ist Gauge hier nicht in einem Container integriert und ein Bericht wird generiert. Schlägt der Test fehl, so können in dem Bericht die Ursachen in Form eines Bildes eingesehen werden.
Wir führten entsprechend des beschriebenen Szenarios einen E2E-Test durch. Bei diesem wollten wir konkret überprüfen, ob die Popup Funktion reagiert und korrekt dargestellt wird. Dazu starteten wir ExplorViz und die Spring Petclinic mit Docker Desktop über die Konsole. Laufen alle Container wird mit *npm test* der Testlauf mittels Gauge gestartet. Der Headless Modus des Browsers ist dazu ausgeschaltet, damit wir den Testablauf verfolgen können (dieser kann jedoch auch aktiv bleiben). Nach erfolgreicher Beendigung der Tests fahren wir ExplorViz und die Spring Petclinic wieder herunter. Da alle Tests als erfolgreich gewertete wurden, schließt dies auch die Popup Funktion mit ein. Ein Blick in die Berichte ist somit nicht notwendig.

5.1.2. Szenario 2: Durchführung automatisierter E2E-Tests

Es wird ein Szenario konzipiert, bei welchem die Tests exemplarisch in der Praxis angewandt werden. Hierbei werden die E2E-Tests automatisch durchgeführt. Das heißt, dass nach jedem Push in das Repository die Anweisung an die CI-Pipeline erfolgt, den entsprechenden Workflow zur Durchführung der Tests zu starten. Da die Anweisung zur Durchführung der Tests durch einen Push-Befehl in das Repository gegeben wird, startet der/die EntwicklerIn mit diesem Befehl die Tests. Bei einem fehlgeschlagenen Testlauf folgt eine Information in Form einer Mail an den/die EntwicklerIn, wohingegen bei einem erfolgreichen Testlauf keine Information versandt wird. Die Durchführung der Tests erfolgt zeitgleich mit der Durchführung der Unit- und Integrationstests. Weiterhin sind die Interaktionen

bei jedem Testdurchlauf gleichbleibend und überprüfen die Anwendung immer mit den gleichen Grundbedingungen. Die Tests führen bei jedem Durchlauf dieselben Schritte aus. Entsprechend ist die Visualisierung in Form der dargestellten Module homogen.
Entsprechend des beschriebenen Szenarios wurde eine Änderung des Repository vorgenommen. Der von uns durchgeführte Push löste den Start des Workflows im GitHub Repository aus. Anschließend warteten wir ca. vier Minuten bis der Workflow durch die CI-Pipeline beendet wurde. Da wir keine Informationsmail erhielten, schlossen wir, dass der Test keine Fehler enthielt. Für eine detailliertere Betrachtung prüften wir dennoch die gegebenen Informationen des Workflows, was die Vermutung des korrekt durchgelaufenen Tests bestätigte.

5.2. Evaluation der Ergebnisse

Beide Szenarien, also die lokale Ausführung und die automatisierte in GitHub, waren sowohl in der Durchführung als auch in den Ergebnissen erfolgreich. In Bezug auf die Handhabung muss für Szenario 1 vor jedem Testlauf ExplorViz und die Zielanwendung gestartet werden. Diese können zwar über ein Docker Compose gestartet werden, was jedoch einen weiteren Schritt zum Arbeitsablauf hinzufügt und ihn damit auch fehleranfälliger macht. Dieser Schritt ist in Szenario 2 nicht notwendig, da die Container in der CI-Pipeline selbstständig gebaut und gestartet werden. Weiterhin spielen die verfügbaren Ressourcen, auf welcher der Test durchgeführt wird, eine Rolle. Weist das lokale Gerät in Szenario 1 nicht genügend Arbeitsspeicher auf, so wird die Verwendung von Docker erschwert. Im schlimmsten Fall ist bei zu geringem Arbeitsspeicher gar keine Verwendung möglich. Auch diese Problematik wird in Szenario 2 durch die CI-Pipeline gelöst, welche die notwendigen Ressourcen zur Verfügung stellt. Weiterhin werden die Container in GitHub Actions bei jedem Durchlauf neu gebaut, wohingegen die Container lokal nur einmal gebaut werden müssen (wobei sie für einen sauberen Testlauf auch jedes mal neu gebaut werden müssten). Dadurch ist der Testablauf in Szenario 1 zwar schneller, hat jedoch nicht immer denselben Ausgangspunkt wie das Szenario 2. Die Anzahl der getesteten Funktionen spielt in beiden Szenarien gleichermaßen eine Rolle. Bei einer höheren Anzahl an zu testenden Funktionen erhöht sich die Testzeit (durch Gauge). Während der Entwicklungsphase dauerte die reine Ausführung der E2E-Tests (also ohne den Start der Docker Container) mit Überprüfung von nur einer Funktion unter einer Minuten. Die Dauer von ca. ein bis zwei Minuten beim Testen von vier Funktionen sorgte lediglich für einen marginalen Anstieg der Testdauer.
Von besonderer Bedeutsamkeit ist die Genauigkeit der E2E-Tests in beiden Szenarien. So würde ein minimaler Unterschied in der Visualisierung (vgl. Abbildung 5.1 (A) und (B)), der mit bloßem Auge nur schwer zu erfassen wäre, durch die Tests auffallen. Damit wird in beiden Szenarien eine sehr hohe Genauigkeit in dem Vergleich von Ist- mit Erwartungswerten erreicht.

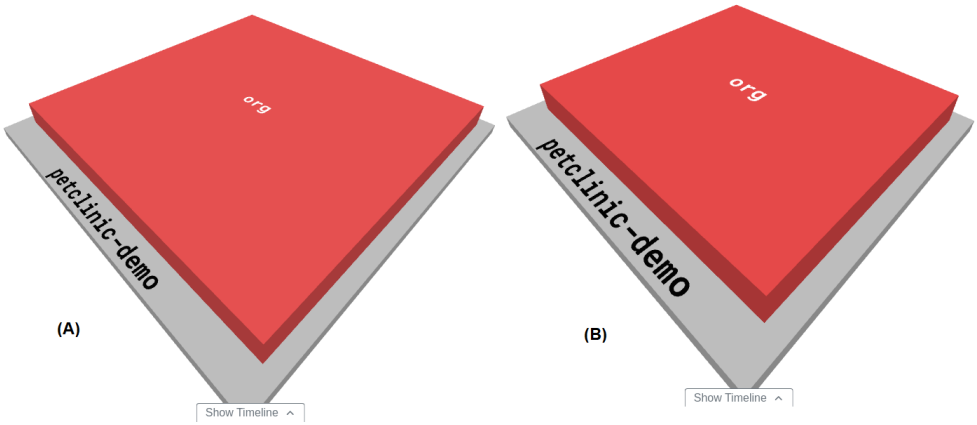

Abbildung 5.1. Beispieldarstellung

Zudem wurde bei der Durchführung der Tests in beiden Szenarien festgestellt, dass die Visualisierung der Spring Petclinic Anwendung nicht konsistent ist. So ist das Modul *org* aus Abbildung 5.1 (A) größer, als in (B). Dieser Unterschied wurde durch die Tests festgestellt, und zeigt, dass auch kleine Unterschiede erkannt werden. Die Inkonsistenz in der Visualisierung hängt dabei mit einem Fehler im Backend von ExplorViz zusammen. Ein Nachteil, der sich in Szenario 1 zeigte, ist die Abhängigkeit vom Betriebssystem des Hosts. So wurden die Tests auf einem Windows System funktional entwickelt. Als die Tests jedoch auf einem anderen Betriebssystem durchgeführt werden sollten, traten Fehler auf. Die in Szenario 2 genutzte CI-Pipeline von GitHub sorgt hingegen dafür, dass die Testung unabhängig vom Betriebssystem des Hosts ist. Dieser Vorteil sorgt dafür, dass in Abschnitt 5.1.2 die Tests mit wenig Aufwand durch alle am Projekt Beteiligten ausgeführt werden können. Weiterhin werden Arbeitsschritte zur Anpassung an das Betriebssystem vermieden, was den Testprozess in Abschnitt 5.1.2 effizienter gestaltet. Der letzte zu erwähnende große Unterschied ist das kontinuierliche Testen der Anwendung durch die Automatisierung. Wohingegen zum jetzigen Zeitpunkt die Tests lediglich nach Bedarf durch die Testperson durchgeführt werden, erfolgt die Testung der Anwendung in Abschnitt 5.1.2 kontinuierlich nach jedem Push. Somit ist die Frequenz der Tests sehr viel höher als in Abschnitt 5.1.2. Dieser Umstand kann dazu führen, dass Bugs schneller erkannt werden und der Code eine höhere Qualität erreicht. Außerdem wird die Möglichkeit des Vergessens von Tests und damit auch eine unsaubere Entwicklung negiert.

5.3. Threats to validity

Auch nach der mehrfachen Ausführung der Tests in exemplarischen Szenarien können wir nicht sicher sein, dass die generierten Daten im Form von Testergebnissen frei von Fehlern sind. So könnten die generierten Daten durch lokale Faktoren der Endgeräte (Hardware, Software) beeinflusst worden sein. Dies bezieht sich auch auf die Erwartungswerte. Diese wurden zwar aus ExplorViz heraus generiert, könnten aber in ihrer Entstehung fehlerhaft gewesen sein. So können die Szenarien zwar einen Einblick in das Verhalten der Tests geben, jedoch keine endgültige Aussage über die kontinuierliche Nutzbarkeit treffen. Diese Annahme kann noch dadurch gestützt werden, dass der Umfang der Tests innerhalb der Szenarien beschränkt war. So ergab sich keine hohe Varianz in den Endgeräten, auf welchen die Tests durchgeführt wurden.

Kapitel 6

Verwandte Arbeiten

Für das Ziel der Entwicklung automatisierter E2E-Tests für Softwarevisualisierungen mussten eigene Testszenarien konzipiert und entwickelt werden. Entsprechend werden wir uns auf Arbeiten fokussieren, welche sich mit einer ähnlichen Thematik beschäftigt haben.
So stellen Azimi et al. [2] in ihrer Arbeit AdaptTV, einen Ansatz für E2E-Tests von Benutzeroberflächen von Smart TVs, vor. Dabei wird eine Art Blackbox Test verwendet, welcher das dargestellte Bild interpretiert. Der Ansatz nutzt zwei SmartTVs (eine alte und eine neue Version) und eine Test Suite, welche auf der alten TV-Version läuft. Danach werden die Testfälle der Test Suite auf der alten TV-Version durchgeführt, wodurch das User Interface (UI) durchquert wird. Danach erfolgt eine semantisch äquivalente Durchquerung des UI auf dem neuen Gerät, wobei Feedback generiert wird.
In der Arbeit von Fasolino et al. [5] werden Capture and Replay (CaR) Techniken als Variante von E2E-Tests vorgestellt. CaR-Techniken zeichnen Nutzerinteraktion, wie das Klicken eines Buttons, auf und wandeln jene in Testskripte um. Diese Technik weist jedoch viele Lücken auf, so können kleine Veränderung an der GUI bereits zu einem Fehlschlag der Tests führen. Das Thema der Arbeit behandelt die Möglichkeit, Testfälle robuster und anpassbarer zu gestalten. Die vorgestellte Technik bewegt sich im Bereich der Definition robuster Testfälle und Locator von DOM-Objekten, welche dort als Hooks bezeichnet werden. Diese sollen dafür sorgen, dass auch bei Veränderung der GUI in der Entwicklung die Elemente eine konsistente Bezeichnung erhalten. Dazu wird ebenfalls ein Tool vorgestellt, welches Hooks in der HTML-Seite implementiert.
In der Arbeit von Wetzlmaier und Ramler [18] wird die Methode des *Monkey testings* aufgegriffen. Dabei werden bereits vorhandene automatisierte GUI Tests, in Form von Regressionstests, um Monkeys erweitert. Diese sollen für eine zufällige Komponente bei Tests sorgen und damit eine größere Abdeckung schaffen. Das Ziel ist somit die Findung neuer Bugs, welche nicht durch die Regressionstests gefunden werden.
Mit der obersten Stufe in der Testpyramide sind E2E-Test sehr ressourcenintensiv. Augusto [1] beschreibt in seiner Arbeit einen Ansatz zum ressourcenschonenden Einsatz von E2E-Tests. Die Testfälle werden in unterschiedliche Gruppen kategorisiert, um effizient gestaltete Test Suits zu schaffen. In der Arbeit wird auf einen vorgestellten Ansatz zur Orchestrierung zurückgegriffen, welcher in drei Prozesse unterteilt ist: im ersten folgt die Identifikation der benötigten Ressourcen, zweitens werden die Testfälle nach der benötigten Ressource gruppiert und drittens folgt der Verteilungsprozess der Testfälle, wobei der Fokus auf der Reduzierung der Ausführungszeit der Tests liegt.

Kapitel 7

Fazit und Ausblick

7.1. Fazit

Das übergeordnete Ziel der Arbeit war die Konzeption und Entwicklung von E2E-Tests zum kontinuierlichem Testen der Softwarevisualisierung ExplorViz. Entsprechend wurden mit Gauge-Taiko zuvor konzipierte Testszenarien entwickelt. Für die entwickelten E2E-Tests wurde ein Docker Image gebaut und anschließend in GitHub Actions integriert. Dort wurde ein Workflow zur Durchführung automatisierter Tests nach einem Push in das Repository erstellt.
Anschließend folgte zur Evaluation die Nutzung der entwickelten Tests in zwei exemplarischen Szenarien. In dem ersten Szenario wurden die Tests lokal auf einem Gerät nach Bedarf und in einem zweiten Szenario mit Hilfe von GitHub Actions automatisiert und damit kontinuierlich durchgeführt. Anschließend wurde festgestellt, dass das zweite Szenario und damit die automatisierte Durchführung der E2E-Tests mehr Vorteile aufwies als das erste Szenario. Darunter fällt die einfachere Handhabung der E2E-Tests. Diese drückt sich dadurch aus, dass die Tests unabhängig vom Betriebssystem der Testperson sind und keine Anpassungen vorgenommen werden müssen. Weiterhin ist ein manueller Start von ExplorViz und der Spring Petclinic nicht für jeden Testlauf erforderlich. Daraus kann man schließen, dass die Zeit, die die Testperson aktiv mit dem Test verbringt, reduziert wird. Durch eine automatisierte und damit kontinuierliche Durchführung der Tests, erhöht sich die Frequenz der Tests, was zur Folge hat, dass Bugs schneller erkannt werden und die Qualität des Codes sich erhöht. Weiterhin wird der menschliche Fehler des "Vergessens"der Testdurchführung damit ausgeschlossen.

7.2. Ausblick

Auf Basis der Ergebnisse dieser Arbeit identifizierten wir noch weitere Aspekte, welche für zukünftige Arbeiten in dieser Thematik wertvoll sein könnten. So wurde lediglich eine kleine Auswahl an Testszenarien getroffen. Diese könnten bei zukünftigen Arbeiten sinnvoll erweitert werden. Beispielsweise erfolgt keine Betrachtung der Website zur Auswahl der Softwarelandschaften, auf welcher die Zusammenfassung der unterschiedlichen durch ExplorViz erstellten Softwarelandschaften erfolgt. Weiterhin gibt es noch immer Funktionen, wie das Öffnen der Module für Einblicke in die Stadtmetapher, welche nicht durch die Tests abgedeckt werden. Diesbezüglich könnte man die Testabdeckung sinnvoll erweitern.
Ein weiterer Aspekt ist der derzeitige Stand der Entwicklung. So gibt es zurzeit keine Möglichkeit, die Testfälle zu erweitern oder zu ändern, ohne das Image neu bauen zu müssen. Es könnte in einer zukünftigen Arbeit entsprechend ein generisches Docker-Image geschaffen werden, was die Möglichkeit zum Austausch von Code-Bausteinen ermöglicht und den Container damit dynamischer macht. Ermöglichen könnte dies die Nutzung von Volumes in diesem Kontext.
Der nächste zu nennende Aspekt betrifft die Zielanwendung. Im Rahmen der Tests wird die Spring Petclinic als Zielanwendung für die Tests genutzt. Zum einen könnten die Interaktionen mit der Spring Petclinic erweitert werden oder eine andere Anwendung als Zielanwendung genutzt werden, um

eine größere Varianz der Testabdeckung schaffen. In diesem Zusammenhang kann auch die Nutzung mehrerer Nutzer durch Gauge simuliert werden. Dies würde eine weitere Kernfunktion von ExplorViz, die Darstellung der gleichen Softwarelandschaft bei mehreren Nutzern, welche sich im selben Raum befinden, testen.
Als letzter Aspekt ist eine mögliche Erweiterung der Tests um einen nicht deterministischen Ansatz. Es könnte geprüft werden, ob der bereits erwähnte *Monkey testing* Ansatz möglich wäre. Dadurch könnte eine zufällige Testkomponente in die E2E-Tests einfließen und dafür sorgen, dass mehr Bugs entdeckt werden. Weiterhin könnten dadurch auch nicht gängige Nutzerinteraktionen getestet werden.

Literaturverzeichnis

[1] Cristian Augusto. „‚Efficient Test Execution in End to End Testing: Resource Optimization in End to End Testing through a Smart Resource Characterization and Orchestration‘“. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, Seiten 152–154. DOI: 10.1145/3377812.3382177. URL: https://doi.org/10.1145/3377812.3382177 (siehe Seite 37).

[2] Mohammad Yusaf Azimi u. a. „‚AdapTV: A Model-Based Test Adaptation Approach for End-to-End User Interface Testing of Smart TVs‘“. In: *IEEE Access* 11 (2023), Seiten 32095–32118. DOI: 10.1109/ACCESS.2023.3262746 (siehe Seite 37).

[3] James Bach. „‚Testen von Software‘“. In: *Softwarequalität in PHP-Projekten*. 2013, Seiten 17–52. DOI: 10.3139/9783446435827.002. URL: https://www.hanser-elibrary.com/doi/abs/10.3139/9783446435827.002 (siehe Seite 7).

[4] Gerald D Everett und Raymond McLeod Jr. „‚Software testing‘“. In: *Testing Across the Entire Software Development Life Cycle*. John Wiley und Sons, 2007 (siehe Seiten 11, 14).

[5] Anna Rita Fasolino und Porfirio Tramontana. „‚Towards the Generation of Robust E2E Test Cases in Template-based Web Applications‘“. In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022, Seiten 104–111. DOI: 10.1109/SEAA56994.2022.00024 (siehe Seite 37).

[6] Wilhelm Hasselbring, Alexander Krause und Christian Zirkelbach. „‚ExplorViz: Research on software visualization, comprehension and collaboration‘“. In: *Software Impacts* 6 (2020), Seite 100034. DOI: https://doi.org/10.1016/j.simpa.2020.100034. URL: https://www.sciencedirect.com/science/article/pii/S2665963820300257 (siehe Seite 3).

[7] Itti Hooda und Rajender Singh Chhillar. „‚Software test process, testing types and techniques‘“. In: *International Journal of Computer Applications* 111.13 (2015) (siehe Seite 14).

[8] Muhammad Abid Jamil u. a. „‚Software Testing Techniques: A Literature Review‘“. In: *Proceedings of the 6th International Conference on Information and Communication Technology for The Muslim World(ICT4M 2016)*. IEEE Comnputer Soceity, Nov. 2016, Seiten 177–182 (siehe Seite 1).

[9] Rasneet Kaur Chauhann und Iqbal Singh. „‚Latest Research and Development on Software Testing Techniques and Tools‘“. In: *International Journal of Current Engineering and Technology*. INPRESSCO, Aug. 2014, Seiten 2368–2372 (siehe Seite 1).

[10] Alexander Krause, Malte Hansen und Wilhelm Hasselbring. „‚Live Visualization of Dynamic Software Cities with Heat Map Overlays‘“. In: *2021 Working Conference on Software Visualization (VISSOFT)*. 2021, Seiten 125–129. DOI: 10.1109/VISSOFT52517.2021.00024 (siehe Seiten 4, 13).

[11] Alexander Krause-Glau, Marcel Bader und Wilhelm Hasselbring. „‚Collaborative Software Visualization for Program Comprehension‘“. In: *2022 Working Conference on Software Visualization (VISSOFT)*. 2022, Seiten 75–86. DOI: 10.1109/VISSOFT55257.2022.00016 (siehe Seiten 13, 16).

[12] Alexander Krause-Glau, Malte Hansen und Wilhelm Hasselbring. „‚Collaborative program comprehension via software visualization in extended reality‘“. In: *Information and Software Technology* 151 (2022), Seite 107007. DOI: https://doi.org/10.1016/j.infsof.2022.107007. URL: https://www.sciencedirect.com/science/article/pii/S095058492200132X (siehe Seiten 4, 25).

[13] Alexander Krause-Glau und Wilhelm Hasselbring. „‚Scalable Collaborative Software Visualization as a Service: Short Industry and Experience Paper‘“. In: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E 2022)*. Sep. 2022, Seiten 182–187. DOI: 10.1109/IC2E55432.2022.00026 (siehe Seiten 3, 4, 13).

[14] Vadym Mukhin u. a. „‚Improved method of testing distributed system interfaces using simulation tests‘“. In: *TASK Quarterly: scientific bulletin of Academic Computer Centre in Gdansk* 25 (2021), Seiten 261–270. DOI: https://doi.org/10.34808/tq2021/25.2/f (siehe Seite 7).

[15] Vadym Mukhin u. a. „‚The Testing Mechanism for Software and Services Based on Mike Cohn's Testing Pyramid Modification‘“. In: *Proceedings of the 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS 2021)*. Band 1. Sep. 2021, Seiten 589–595. DOI: 10.1109/IDAACS53288.2021.9660999 (siehe Seite 1).

[16] Sean Stolberg. „‚Enabling Agile Testing Through Continuous Integration‘“. In: *2009 Agile Conference*. IEEE Computer Society, 2009, Seiten 369–374 (siehe Seite 6).

[17] Richard Wettel und Michele Lanza. „‚Program Comprehension through Software Habitability‘“. In: *15th IEEE International Conference on Program Comprehension (ICPC '07)*. 2007, Seiten 231–240. DOI: 10.1109/ICPC.2007.30 (siehe Seite 4).

[18] Thomas Wetzlmaier und Rudolf Ramler. „‚Hybrid Monkey Testing: Enhancing Automated GUI Tests with Random Test Generation‘“. In: A-TEST 2017. Paderborn, Germany: Association for Computing Machinery, 2017, Seiten 5–10. DOI: 10.1145/3121245.3121247. URL: https://doi.org/10.1145/3121245.3121247 (siehe Seite 37).