

Ruhr-Universität Bochum

Bachelorarbeit

Entwicklung eines erweiterbaren Kommandozeileninterfaces zur Generierung von Web-Applikationen

Schriftliche Prüfungsarbeit
für die Bachelor-Prüfung des Studiengangs Angewandte Informatik an der
Ruhr-Universität Bochum

vorgelegt von
Michael David Kuckuk

am
Lehrstuhl für Informatik im Bauwesen
Prof. Dr.-Ing. Markus König

Abgabedatum:	15. September 2021
Matrikelnummer:	108 017 207 503
1. Prüfer:	Prof. Dr.-Ing. Markus König
2. Prüfer:	Stephan Embers, M. Sc.

Abstract

The ABSTRACT is to be a fully-justified text following the title page. The text will be formatted in 12 pt, single-spaced Computer Modern. The title is “Abstract”, set in 12pt Computer Modern Sans Serif, centered, boldface type, and initially capitalized. Writing the abstract in English is mandatory even if the thesis itself is written in German. The length of the abstract should be roughly about 200 words but must not exceed 250 words.

As usual, the abstract should clearly summarize the aim, the background and the results of your thesis so that an interested reading can decide if he or she wants to further read your (interesting) written report. Also, the abstract should be kept simple. That is, do not include tables, figures or cross references in the abstract. You will have plenty of time in your thesis to explain things more clearly.

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für eine andere Prüfung eingereichten Arbeit.

Ich erkläre weiterhin, dass ich die Arbeit nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Statement

I hereby declare that except where the specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university.

This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

Datum

Unterschrift

Inhaltsverzeichnis

1	Einleitung	1
2	Stand der Technik	2
2.1	Einführung in Frontend-Frameworks für Webapplikationen	3
2.2	Kommandozeileninterfaces	4
2.3	Ähnliche Projektgeneratoren	4
2.4	Funktionale Programmierung	7
2.5	Testgetriebene Entwicklung	7
3	Konzeptionierung	7
3.1	Auswahl der zu vergleichenden Installationsprozesse	8
3.2	Vergleich der Installationsprozesse	12
3.3	Bestimmung der erwünschten Features	16
3.4	Abwägung über Sonderstellungen für TypeScript und Frameworks	17
3.5	Planung der Erweiterungen	18
3.6	Zu verwendende Technologien	19
4	Implementierung	20
4.1	Umsetzung von Erweiterungen im Allgemeinen	20
4.2	Umsetzung des Dialogs	27
4.3	Analyse der CLI-Argumente	27
4.4	Installation von Erweiterungen	30
4.5	Probleme	33
4.6	Vorgehen nach TDD	35
5	Evaluierung	36
5.1	Evaluierung der umgesetzten Features	36
5.2	Sicherstellung der Funktionalität	38
6	Ausblick	39
7	Fazit	41

1 Einleitung

Entwickelnde haben zu Beginn eines neuen Projektes immer erst die Aufgabe, sich eine geeignete Architektur für das Projekt zu überlegen, die sie dann möglichst von Anfang an verfolgen können. In den meisten Programmiersprachen und Umfelden, insbesondere in der Webentwicklung, ist ein Teil dieser Aufgabe die Auswahl von Bibliotheken und Werkzeugen, die bei der Programmierung von Hilfe sein können.

Unter diesen Bibliotheken und Werkzeugen gibt es viele, die besonders häufig verwendet werden oder in bestimmten Kombinationen empfohlen werden. Beispielsweise verwenden 60,4% der von „State of JS 2020“ Befragten die Bibliothek Axios¹ (ein HTTP-Client, die für Browser und Node.js dasselbe Interface bietet)[15]. Unter den Befragten nutzen 89,6% das Werkzeug Node Package Manager (NPM)², 82,3% nutzen ESLint³ (ein Werkzeug zur statischen Codeanalyse) und 70,9% nutzen Prettier⁴ (ein Codeformatierer).

Die genaue Konstellation, in der diese Bibliotheken und Werkzeuge verwendet werden, unterscheiden sich jedoch von Projekt zu Projekt. Sie hängen primär von den Anforderungen des Projekts, aber auch von den persönlichen Präferenzen der beteiligten Entwickelnden ab.

Nachdem entschieden wurde, welche Konstellation von Bibliotheken und Werkzeugen zu verwenden ist, müssen diese installiert werden und miteinander verknüpft werden, so dass sie alle zusammen funktionieren. Dieser Schritt unterscheidet sich von klassischen Entwicklungsaufgaben, da hierbei Probleme wie Laufzeitanalyse oder Graphentheorie keine Rolle spielen. Stattdessen müssen Konfigurationsdateien erzeugt werden, Objekte müssen in gewissen Reihenfolgen initialisiert werden und Werkzeuge müssen in den Kompilierprozess eingebunden werden, der je nach Konstellation der Werkzeuge zunächst eingerichtet werden muss.

Ziel dieser Arbeit ist es, diese initiale Projektgenerierung zu automatisieren. Über ein Command Line Interface (CLI) sollen zunächst einige Fragen gestellt werden, durch die festgestellt wird, welche genaue Konstellation von Bibliotheken und Werkzeugen installiert werden soll. Entsprechend dieser Spezifikation soll dann ein Projekt erzeugt werden, in dem keine weitere Verdrahtung mehr notwendig ist und in dem sofort alle Bibliotheken und Werkzeuge verwendet werden können.

Es gibt bereits mehrere Projekte, die sich ebenfalls dieser Aufgabe widmen und auf die im späteren Verlauf der Arbeit näher eingegangen wird. Diese haben jedoch den Nachteil, dass sie nur begrenzte Unterstützung für Bibliotheken und Werkzeuge liefern. Manche der Projekte bieten eine Möglichkeit zur Erweiterung um neue Bibliotheken und Werkzeuge an, jedoch sind derartige Erweiterungen nicht schon bei der Projektinitia-

¹<https://www.npmjs.com/package/axios>

²NPM genießt jedoch als offizieller Node.js-Paketmanager eine Sonderstellung. Daher ist diese Zahl nur begrenzt mit anderen Verwendungsquoten vergleichbar.

³<https://www.npmjs.com/package/eslint>

⁴<https://www.npmjs.com/package/prettier>

lisierung verwendbar und müssen daher in weiteren Schritten nach der Initialisierung ausgeführt werden.

Neben der beschriebenen Erleichterung bieten derartige CLIs auch den Vorteil, dass sie Einsteigenden bei der Entdeckung neuer Werkzeuge und Bibliotheken helfen können. Im Jahr 2019 gab es bereits über 1,3 Millionen NPM-Pakete [7], die einen Überblick insbesondere am Anfang erschweren.

Durch ein CLI kann eine Vorauswahl dieser Pakete getroffen werden, die leichter von Einsteigenden überblickt werden kann. Zudem können der Vorauswahl weitere Informationen wie Abhängigkeiten zwischen Bibliotheken oder Links zu zugehörigen Dokumentationsseiten angehängt werden, die das initiale Erstellen eines Projektes zusätzlich erleichtern.

Im Rahmen dieser Arbeit wird daher ein solches CLI namens *generate-web-app* (GWA) entwickelt. Es soll in der Lage sein, leicht verständliche Fragen zu stellen und basierend auf den Antworten ein funktionsfähiges Projekt zu erzeugen. Bei der Konzeptionierung und Entwicklung wird ferner darauf geachtet, dass GWA leicht erweitert werden kann, um die Einbeziehung zukünftiger oder zum Entwicklungszeitpunkt nicht berücksichtigter Bibliotheken und Werkzeuge zu ermöglichen.

Bevor mit der Konzeptionierung von GWA begonnen wird, wird der Stand der Technik erläutert. Dies umfasst neben einigen Grundlagen, die zum besseren Verständnis der Arbeit beitragen, eine Vorstellung der bereits erwähnten Projekte mit ähnlicher Zielsetzung.

Daraufhin wird analysiert, welche Anforderungen genau von GWA erfüllt werden sollen. Hierfür werden zunächst die Vor- und Nachteile der bereits existierenden Projektgeneratoren analysiert. Daraufhin wird eine Auswahl der zu verwendenden Technologien getroffen und es werden erste Pläne zur Gestaltung des Ablaufs und des Datenmodells angefertigt.

Der Fokus der Arbeit liegt auf der Implementierung von GWA. In diesem Zusammenhang wird das allgemeine Vorgehen bei der Entwicklung erläutert. Es werden einige zentrale Probleme untersucht, für die verschiedene Lösungsoptionen entwickelt und diskutiert werden und von denen anschließend eine ausgewählt und umgesetzt wird.

Abschließend wird evaluiert, ob die gesetzten Ziele erfüllt werden konnten. Es wird diskutiert, welche Probleme weiterhin bestehen und womit sich zukünftige Forschung auseinandersetzen könnte.

2 Stand der Technik

Um die Implementierung der Arbeit vollständig nachvollziehen zu können, ist es wichtig, einige Konzepte verstanden zu haben und zu wissen, worauf die Entwicklung aufbauen wird. Diese Grundlagen werden im Folgenden eingeführt und erläutert.

2.1 Einführung in Frontend-Frameworks für Webapplikationen

Um interaktive Webapplikationen zu entwickeln, müssen diese mit HTML, CSS und JavaScript programmiert werden. Browser stellen den mit HTML definierten Inhalt dar, manipulieren das Aussehen der Seite anhand der durch CSS definierten Regeln und verfügen zudem über einen JavaScript Interpreter, der den übermittelten JavaScript Code ausführen kann und darüber eine Manipulation der Seiteninhalte erlauben.

Durch diese Möglichkeit ergibt sich das Problem, dass die Daten, die das Programm berechnet (oder auf andere Weise erlangt) Usern korrekt dargestellt werden und umgekehrt Eingaben von Usern sich in den Daten widerspiegeln sollen.

Beim Entwickeln solcher Funktionalität wiederholen sich häufig bestimmte Probleme wie die Synchronisierung von Daten mit der Anzeige oder die Wiederverwendung von Teilen der Nutzeroberfläche. Diese Probleme können mit Hilfe von Frontend-Frameworks gelöst werden.

Zur Zeit gibt es vor allem drei Frameworks, die häufig verwendet werden: React, Angular und Vue [15]. Angular baut dabei auf dem klassischen Model View Controller (MVC)-Pattern auf. In einer Variation von HTML, in der bestimmte TypeScript-Ausdrücke interpretiert werden können, kann die View aufgebaut werden. Die Funktionalität des Controllers wird in einer TypeScript-Klasse implementiert und das Model kann über Services dargestellt werden, die dann automatisch instanziiert und in die Controller injected werden. **ZITAT FEHLT**

Vue und React verfolgen nicht so strikt das MVC-Pattern. Vue verfügt (ähnlich wie Angular) über eine Variation von HTML, in der auch bestimmte Ausdrücke evaluiert werden können, sodass so die View definiert werden kann. Allerdings ist es bei Vue üblich, dass sich noch in derselben Datei wie die View (und das CSS) auch der Controller befindet. Vue gibt im Gegensatz zu Angular keine klare Vorgabe, wie das Model umzusetzen ist.

React hingegen verfügt über einen anderen Ansatz. Hier wird es ermöglicht, eine Variante von HTML-Ausdrücken in JavaScript bzw. TypeScript einzufügen, die dann wie andere Werte auch in Variablen gespeichert werden können. Auf diesem Weg wird parallel zum eigentlichen Document Object Model (DOM) ein sogenanntes Virtual DOM generiert, das dann in das eigentliche DOM überführt wird. Somit ist es in React möglich, View und Controller komplett miteinander zu verbinden. Zudem verfügt auch React nicht über einen klaren Ansatz um das Model umzusetzen.

Alle drei dieser Frameworks ermöglichen eine komponentenbasierte Entwicklung. Das bedeutet, dass mit sehr geringem Aufwand die View in Komponenten unterteilt werden kann, die dann wiederverwendet werden können.

Aus diesen und weiteren Gründen werden aktuelle Webapplikationen in der Regel auf Grundlage eines Frontend-Frameworks aufgebaut **ZITAT FEHLT**. Da jedes Frontend aber mehr oder weniger klare Strukturen, Datenflüsse, Syntaxen und vieles mehr vorgibt, ist die Entscheidung eines Frameworks in Bezug auf die Suche weiterer Abhängigkeiten eine sehr folgeschwere Entscheidung.

Beispielsweise ist es im Rahmen der empfohlenen Installationsprozesse nicht möglich, Angular ohne TypeScript oder React ohne ESLint zu verwenden **ZITAT FEHLT** (Angular und React quick start guides, vlt auch einfach die jeweiligen CLIs?). Für alle drei der genannten Frameworks gibt es Komponentenbibliotheken⁵, die Google's Material Design System folgen; allerdings sind diese Bibliotheken sehr verschieden, haben unterschiedliche Features und gehen nicht aus demselben Code hervor.

Die Entscheidung eines Frameworks ist daher eine der ersten Entscheidungen, die Entwickelnde beim Einrichten ihrer Projekte treffen müssen. Sie schränkt die weitere Auswahl an Werkzeugen und Bibliotheken ein. Außerdem können, bis auf in besonderen Kontexten wie Micro-Frontends oder WebComponents **ZITAT FEHLT**, nur mit hohem Aufwand oder gar nicht zwei Frameworks zusammen verwendet werden.

2.2 Kommandozeileninterfaces

Die erste Art von Programmen, die auf Computern ausgeführt werden konnten, waren sogenannte Kommandozeileninterfaces (CLIs). Über die Kommandozeile konnten diese Programme über ihren Namen bzw. ihren Speicherort aufgerufen werden, wobei zusätzliche Argumente übergeben werden konnten, um den Ablauf des Programms zu beeinflussen[18].

Diese Benutzweise birgt verschiedene Vor- und Nachteile. A priori bietet sie keine Entdeckbarkeit der Funktionalität. Erst durch die Einrichtung eines weiteren Befehls bzw. einer Konfigurationsoptionen kann Nutzenden eine Liste aller Nutzungsmöglichkeiten des Programms angezeigt werden. Nutzende müssen aber auch diesen Befehl kennen, um ihn ausführen zu können. Dafür lassen sich derartige Befehle sehr leicht automatisiert ausführen, da sie ohne weiteres in Skripte eingebunden werden können. Um eine automatische Ausführung grafischer Benutzungsoberflächen zu ermöglichen, müsste dahingegen die Benutzung einer Computermouse simuliert werden [17].

Seit diesen Anfängen von CLIs ist es jedoch möglich geworden, im Rahmen eines CLIs weitere Nutzereingaben wie die Verwendung von Tastatur und Maus zu verarbeiten. So können Nutzerinteraktionen ermöglicht werden, die einer grafischen Benutzeroberfläche gleich kommen. Beispielsweise ist es möglich, in der Kommandozeile sich eine Weltkarte anzuzeigen, die mit Maus und Tastatur bedienbar ist [11].

Heutzutage bieten CLIs daher den Vorteil, dass sie weiterhin leicht automatisierbar sind und dennoch Interaktivität und damit Entdeckbarkeit der Funktionen ermöglichen. Um derartige Interaktivität leichter umzusetzen, gibt es zahlreiche Bibliotheken.

2.3 Ähnliche Projektgeneratoren

Um das Problem der initialen Projektkonfiguration zu automatisieren oder zumindest zu erleichtern, bieten die drei oben erwähnten Frameworks jeweils ein Programm an,

⁵Material-UI für React: <https://material-ui.com/>; Angular Material: <https://material.angular.io/>; Vuetify für Vue: <https://vuetifyjs.com/en/>

was zur initialen Erstellung von Projekten mit dem jeweiligen Framework empfohlen wird. Einige dieser Programme sind darüber hinaus in der Lage, auch nach der initialen Erstellung Änderungen am Code vorzunehmen. Diese Features sind jedoch für GWA nicht geplant und werden daher nicht näher betrachtet.

2.3.1 React

In der Dokumentation für React wird empfohlen, neue Projekte über das CLI `create-react-app` (CRA) zu erstellen. Dieses kann per NPM installiert werden und ist dann in der Lage, den Inhalt eines angegebenen Templates (oder des Standardtemplates) in ein spezifiziertes Verzeichnis zu kopieren. Daraufhin werden die benötigten Abhängigkeiten per NPM oder mittels eines anderen installierten Paketmanagers (wie z.B. Yarn) installiert.

Dritten ist es möglich, eigene Templates für CRA zu erzeugen, die dann wie Erstanbietertemplates zur Erzeugung des neuen Projekts genutzt werden können. Wenn also mittels CRA ein Projekt erzeugt werden soll, das bereits mit gewissen Bibliotheken oder Werkzeugen ausgestattet ist, muss dafür zuvor ein entsprechendes Template erstellt worden sein. Die Wahrscheinlichkeit, dass das jedoch passiert ist, sinkt mit der Spezifität der Wünsche.

Die Erstanbietertemplates geben bereits die empfohlene Ordner- und Dateistruktur für React-Projekte vor; durch Drittanbietertemplates kann hiervon aber abgewichen werden. Außerdem installieren die Erstanbietertemplates bereits einige weitere Werkzeuge wie ESLint, Jest (ein Werkzeug zum Schreiben und Ausführen von automatisierten Tests, in der Regel Unit Tests) und die sogenannten `react-scripts`. Diese sind eine Sammlung von kleinen Skripten und Abhängigkeiten, die Prozesse automatisieren, die für die Entwicklung notwendig sind. Beispielsweise wird so ein Skript zur Verfügung gestellt, welches einen lokalen Server startet, der den aktuellen Stand der Webapplikation hostet und bei jeder Codeänderung automatisch aktualisiert. Dieser Server gibt zudem auch die Warnungen aus, die ESLint erzeugt.

Außerdem ist es bei der Verwendung dieser Templates mit geringem Aufwand möglich, bestimmte empfohlene Bibliotheken und Werkzeuge nachzurüsten. So gibt es zum Beispiel Anleitungen für die Installation von Prettier oder von CSS Präprozessoren, die aus jeweils einem in der Kommandozeile auszuführenden Befehl bestehen. Für Prettier muss zudem noch eine bestimmte, bereits existierende Datei um einige Zeilen ergänzt werden.

2.3.2 Angular

Das Angular-Team stellt das sog. Angular-CLI zur Verfügung, das Entwickelnden beim Programmieren verschiedene wiederkehrende und repetitive Aufgaben abnehmen soll (wie z.B. die Erstellung und Einbindung neuer Komponenten). Neben diesen Aufgaben wird das Tool auch zur Erstellung neuer Angular-Projekte genutzt.

Der Befehl `ng new <project-name>` führt Nutzende in einen Dialog, bei dem zwei Fragen zur gewünschten Konfiguration des Projektes gestellt werden. Über explizit gesetzte Kommandozeilenoptionen können dann insgesamt Features konfiguriert werden, mit denen das Projekt eingerichtet wird. Diese Features sind jedoch allesamt Angular-intern, d.h. es gibt keine Möglichkeit weitere Libraries oder ähnliches installieren zu lassen, die nicht direkt vom Angular Team kommen. Selbst einige Libraries (z.B. Angular Material), die vom Angular-Team direkt entwickelt werden, sind nicht in diese initiale Projektgenerierung eingebunden. Eine Ausnahme hiervon bildet die Auswählbarkeit der CSS-Präprozessoren SCSS, Sass und Less.

Im Gegensatz zu CRA sind hier die Möglichkeiten, die Nutzenden bei der initialen Projekterstellung geboten werden, sehr eingeschränkt. Es gibt die Möglichkeit, bei der Einrichtung eine sogenannte „Collection“ anzugeben, die dann die Erstellung des eigentlichen Projektes übernimmt. Es könnte also eine Collection entwickelt werden, die weitere Libraries je nach Eingabe einbindet. Jedoch scheint die Umsetzung dieses Features abzunehmen: von den überprüften Collections für ESLint, Prettier und Apollo-Angular hat nur eine dieses Feature (also die Installation bei der Initialisierung statt hinterher) jemals unterstützt, hat es aber mittlerweile wieder eingestellt [4] [6] [16].

Im Gegensatz zu der React-Lösung können sich Nutzende hier (im Rahmen der beschränkten angebotenen Optionen) eine beliebige Konfiguration aussuchen und sind nicht darauf angewiesen, dass jemand vor ihnen schon denselben Wunsch hatte. Außerdem wird durch die gestellten auch Anfänger:innen die Entdeckung und der Einstieg in Angular erleichtert, da gewisse Features automatisch an sie herangetragen werden.

2.3.3 Vue

Von den drei Frameworks bietet Vue in Bezug auf die Projekterstellung das umfangreichste CLI. Zu aller erst besteht die Möglichkeit, ein Preset auszuwählen. Dies kann eines von zwei Standardpresets sein oder eines, was zuvor auf dem lokalen Computer erstellt und gespeichert wurde. Es ist auch möglich, ein bereits existierendes Preset oder ein komplett neues Preset als Kommandozeilenparameter anzugeben.

Falls kein Preset ausgewählt wird, ist zunächst im Rahmen einer Multiple-Choice-Frage eine Vorauswahl von zehn Features zu treffen, die installiert werden sollen nicht installiert werden sollen. Daraufhin werden zu den ausgewählten Features detailliertere Fragen gestellt. Insgesamt stehen durch dieses Tool über 20 verschiedene Bibliotheken bzw. Libraries ohne weiteren Konfigurationsaufwand zur Verfügung. Die soeben erstellte Konfiguration kann anschließend als neues Preset gespeichert werden.

Wie im Angular-CLI können auch hier CSS-Präprozessoren und ein Framework-spezifischer Router in das Projekt eingebunden werden. Darüber hinaus besteht jedoch eine Auswahl aus verschiedenen Werkzeugen zur statischen Codeanalyse, verschiedenen Bibliotheken und Werkzeugen sowohl für Unit- als auch für Ende-zu-Ende-Tests und weiteren vereinzelt Bibliotheken und Features.

2.4 Funktionale Programmierung

2.5 Testgetriebene Entwicklung

Der Begriff der Testgetriebenen Entwicklung, im Englischen auch Test-driven Development (TDD) genannt, bezeichnet ein bestimmtes Vorgehen beim Programmieren, das zu zuverlässigerem Code führen soll und das dazu beitragen soll, dass weniger ungebrauchter Code geschrieben wird [3].

Während der Entwicklung werden wiederholt drei Phasen durchlaufen. Zunächst wird ein automatisierter Test geschrieben, der eine noch nicht implementierte Funktionalität testet. Diese erste Phase wird erst beendet, wenn der Test fehlschlägt. So wird garantiert, dass der Test nicht versehentlich immer funktioniert (z.B. weil er versehentlich eine andere, bereits implementierte Funktionalität testet).

In der zweiten Phase wird dann die gewünschte Funktionalität implementiert. Hierbei wird jedoch darauf geachtet, dass nur so viel Code geschrieben wird, wie notwendig ist, um den zuvor geschriebenen Test bestehen zu lassen. Hierbei wird jeder Anspruch auf Codequalität außer Acht gelassen, sodass der Fokus darauf liegt, mit minimalem Aufwand den Test zu beheben.

Im Anschluss an die zweite Phase hat man eine Lösung, von der man weiß, dass sie funktioniert (da sie den zuvor geschriebenen Test und alle präexistenten Tests besteht). Diese Lösung muss zunächst gesichert werden, bevor in der dritten Phase der Code aufgeräumt werden kann. In diesem Schritt können Verbesserungen der Codequalität, der Laufzeit oder sonstiger Kriterien vorgenommen werden. Es ist jedoch bei allen Änderungen stets darauf zu achten, dass weiterhin kein Test fehlschlägt. Sollte dies dennoch geschehen, so kann jederzeit der Code auf den Stand von Phase 2 zurückgesetzt und erneut mit Phase 3 begonnen werden.

Nach der dritten Phase ist somit Code entstanden, der sämtlichen Qualitätsansprüchen genügt. Er ist außerdem durch automatische Tests abgedeckt, von denen bekannt ist, dass sie bei Entfernung des Features fehlschlagen würden. Dieser Stand ist nun zu sichern und die Entwicklung des nächsten Features beginnt erneut mit der ersten Phase.

3 Konzeptionierung

Angesichts dessen, dass das geplante CLI verschiedenste Bibliotheken und Werkzeuge unterstützen können soll und dabei auch zukunftssicher bleiben soll, ist es unabdingbar, die Entwicklung im Vorhinein zu planen. Hierfür muss zunächst untersucht werden, welche Gemeinsamkeiten und welche Unterschiede bei verschiedenen Installationsprozessen vorliegen. Daraufhin kann dann ein System erarbeitet werden, was die Gemeinsamkeiten fördert und gleichzeitig Raum für die bekannten, aber auch möglichst für erwartbare unbekannte Unterschiede lässt.

3.1 Auswahl der zu vergleichenden Installationsprozesse

Um zu bestimmen, welche Schritte bei der Einrichtung von neuen Projekten durchführbar sein müssen, wurden zunächst verschiedenste Projekte erstellt, zu denen dann möglichst verschiedene Abhängigkeiten und Werkzeuge hinzugefügt wurden.

Aufgrund der bereits erläuterten Beliebtheit von Frontend-Frameworks wurde für jedes der drei bekanntesten Framework ein Projekt erstellt, in dem dann die entsprechenden Installationen durchgeführt wurden. Für diese initiale Erstellung wurde jeweils das bereits vorgestellte CLI verwendet.

Bei der Auswahl dessen, was zu diesen Projekten hinzugefügt wurde, lag der Fokus zum einen auf Bekanntheit und Verbreitung und zum anderen auf Vielseitigkeit. Da sich ähnliche Werkzeuge oft auch in ihrem Installationsprozess ähneln (beispielsweise waren die Schritte zur Installation von ESLint fast identisch zu den Schritten zur Installation von TSLint⁶), ist es für diese Vorbereitung also nicht nötig, eine Abhängigkeit hinzuzufügen, wenn bereits eine andere ausprobiert wurde, die einen Ähnlichen Funktionsumfang hat.

Aus diesen Gründen wurden folgende Werkzeuge installiert:

- **ESLint** ist ein Werkzeug zur statischen Codeanalyse für JavaScript. Es gibt Warnungen aus, wenn bestimmte Ausdrücke verwendet wurden, die häufig zu Bugs führen (z.B. die Verwendung von einem doppelten Gleichheitszeichen anstelle eines dreifachen Gleichheitszeichens, d.h. die typunsichere Gleichheitsüberprüfung anstelle Gleichheitsprüfung mit Prüfung auf Typgleichheit).

ESLint verfügt auch über die Möglichkeit, Formatierungspräferenzen anzugeben. Diese werden in die Analyse mit einbezogen und darüber können beispielsweise Warnungen ausgegeben werden, wenn Zeilen falsch eingerückt sind. Einige Warnungen, darunter die meisten Warnungen zur Formatierung, können von ESLint auf Wunsch automatisch behoben werden. Standardmäßig ist ESLint jedoch nur passiv, also idempotent.

ESLint wird stellvertretend für alle Werkzeuge zur statischen Codeanalyse überprüft. Insbesondere entfällt deshalb die Überprüfung von TSLint; zumal das Projekt veraltet ist und die Entwicklung zugunsten von ESLint aufgegeben wurde [2] [1]. Da es aber noch automatisch mit Angular zusammen installiert wird, hätte man seine Überprüfung dennoch in Betracht ziehen können.

- **Prettier** ist ein Codeformatierer, der unter freiwilliger Angabe von Konfigurationseigenschaften Code einheitlich formatiert (d.h. Einrückungen werden korrigiert, Leerzeichen vor oder hinter Klammern eingefügt oder entfernt etc.). Gegenüber ESLint hat Prettier den Vorteil, nicht nur JavaScript-basierte Dateien formatieren zu können, sondern es kann auch mit anderen Dateiformaten wie Markdown oder CSS umgehen.

⁶<https://palantir.github.io/tslint/>

Darüber hinaus kann Prettier auch ohne Konfiguration (d.h. ohne Festlegung von Präferenzen) aufgerufen werden um Code zu formatieren. ESLint hingegen muss ausgiebig konfiguriert werden, damit der Code formatiert wird. Zudem hat Prettier mehr Einstellmöglichkeiten als ESLint (zumindest in Bezug auf Formatierung).

- **SCSS / Sass**⁷ sind zwei Dialekte eines CSS-Präprozessors. Dieser Präprozessor erweitert die CSS-Syntax um eine weitere Art von Variablen, Verschachtelung, sogenannten "Mixins" und weitere Features. Zur Kompilierzeit wird daraus normales CSS erzeugt. Dank der zusätzlichen Funktionen kann kompakteres, wiederholungsfreieres und oft auch leserlicheres CSS geschrieben werden.

Zugunsten von SCSS / Sass wurde auf die weitere Überprüfung anderer Präprozessoren wie Less⁸ verzichtet.

- **Jest** ist eine Bibliothek zum Schreiben und Ausführen automatisierter Tests. Diese werden in Node.js ausgeführt und verfügen daher a priori nicht über Browserspezifische APIs (wie z.B. Zugriff auf Canvas-Elemente). Viele dieser Funktionalitäten sind aber ggf. über weitere Bibliotheken nachrüstbar. In der Regel wird Jest für Unittests verwendet.

Im Vergleich zu anderen Werkzeugen vereint Jest mehrere Funktionalitäten, die sonst über mehrere Abhängigkeiten aufgeteilt sind. Wo sonst der Ausführer der Tests von der Bibliothek zur Aufstellung und Überprüfung von Annahmen (Engl.: „Assertion Library“) und der Mockingbibliothek getrennt ist und zur Erzeugung eines Berichtes über die Testabdeckung eine weitere Bibliothek notwendig ist, bietet Jest alle diese Features gleichzeitig an.

Seit 2019 ist Jest das meist verbreitetste Werkzeug zum Schreiben von JavaScript-Tests [15] und unterstützt dabei alle drei Frameworks. Außerdem ist seine Installation vergleichsweise sehr simpel, da hier nicht mehrere Werkzeuge zusammen installiert und miteinander eingerichtet werden müssen. Da diese höhere Komplexität also vermeidbar ist, werden andere Bibliothekskombinationen für Unittests nicht näher betrachtet.

- **Cypress** ist ein weiteres Werkzeug zum Schreiben automatisierter Tests, allerdings mit dem wichtigen Unterschied zu Jest, dass die Tests im Browser ausgeführt werden. Aktuell ist Cypress in der Lage, Chromium-basierte Browser und Firefox zu kontrollieren, um das Laden und die Interaktion mit einer Seite nicht nur per Node.js zu simulieren, sondern entsprechende Ereignisse in einem Browser so auszulösen, dass sie nicht (in relevantem Ausmaß) von tatsächlichen Userinteraktionen unterschieden werden können. Außerdem stehen in diesen Tests die Browser-APIs in vollem Umfang zur Verfügung.

⁷<https://sass-lang.com/>

⁸<https://lesscss.org/>

Seit 2020 wird Cypress öfter als vergleichbare Projekte wie Puppeteer oder WebdriverIO verwendet [15], weshalb diese zugunsten von Cypress nicht weiter berücksichtigt werden.

- **Husky und lint-staged** ist eine Kombination von Werkzeugen, die die Ausführung anderer Werkzeuge als Reaktion auf bestimmte Ereignisse ermöglichen.

Einige der zuvor aufgeführten Werkzeuge bieten Leistungen an, die dabei helfen können, sicherzustellen, dass in ein Versionskontrollsystem bzw. Engl.: Version Control System (VCS) eingepflegter Code immer gewissen Standards entspricht. Die automatischen Tests können beispielsweise garantieren, dass kein zuvor existierendes (und von Tests abgedecktes) Feature kaputt gemacht worden ist, und Prettier und ESLint könne garantieren, dass der Code einem gewissen Stil entspricht. Andere Werkzeuge wie CSS-Präprozessoren hingegen bieten in dieser Hinsicht keine weiteren Vorteile.

Um die Qualität des Codes auf einem möglichst hohen Niveau zu halten, kann es also sinnvoll sein, diese entsprechenden Werkzeuge automatisch vor der Einpflegung des Codes in ein VCS ausführen zu lassen. Genau diese Möglichkeit bietet Husky, sofern das VCS Git verwendet wird. Diese Annahme lässt sich durchaus treffen, da Git mit einem Abstand von ??? das am weitesten verbreitete VCS ist **ZITAT FEHLT**.

Mithilfe von Husky lassen sich mit wenig Konfigurationsaufwand Skripte festlegen, die vor dem Speichern von Code in Git ausgeführt werden sollen. Scheitert eines dieser Skripte, so wird der Speichervorgang abgebrochen. Lint-staged erleichtert hierbei das Überprüfen der geänderten Dateien, sodass nicht bei jeder Änderung der gesamte Code geprüft werden muss.

Aufgrund dessen, dass diese beiden Werkzeuge empfohlen werden, um Prettier automatisch vor entsprechenden Vorgängen laufen zu lassen, und da ihre Verwendung sich leicht auf zusätzliche Werkzeuge erweitern lässt, werden Husky und lint-staged vergleichbaren Alternativen vorgezogen.

Außerdem wurden die folgenden Bibliotheken installiert:

- **Router** - Jedes der drei Frameworks verfügt über eine Unterbibliothek um verschiedene Routen einzurichten. So kann man ohne ein Neuladen der Seite die URL wechseln, diese Änderung auch im Browserverlauf widerspiegeln lassen und auch schon beim Aufruf einer Unter-URL direkt die entsprechende Komponente anzeigen lassen. Schon die Tatsache, dass die Installation eines Routers in Angular- und Vueprojekten schon bei der initialen Einrichtung ausgelöst werden kann, zeigt, dass Router sehr häufig verwendet werden. Für ein jeweiliges Framework gibt es außerdem in der Regel genau einen geläufigen Router, sodass sich hier keine besondere Bevorzugung stattgefunden hat. Stattdessen wurden alle drei Router gleichermaßen untersucht.

- **Redux** - Wie bereits erläutert, verfügt React nicht über eine besonders empfohlene Methode zum zentralen Statemanagement. Diese Lücke kann jedoch von Redux gefüllt werden. Mithilfe von Redux kann ein zentraler sogenannter SStore erzeugt werden. Dieser kann modifiziert werden, indem man an anderer Stelle eine sogenannte Action erzeugt, die an den Store weitergeleitet wird. Daraufhin werden sogenannte "Reducer aufgerufen, die den aktuellen Wert des Stores und die aktuelle Action entgegennehmen und einen neuen, entsprechend der Action aktualisierten Store zurückgeben.

Diese Reducer sind stets pure Funktionen⁹ und sind daher (da sie keinen Initialisierungsprozess voraussetzen und nichts anderes beeinflussen) sehr leicht testbar. Außerdem ermöglicht Redux in Kombination mit einer Browsererweiterung mehrere interessante Features, darunter ein Feature namens SZeitreise", mit dem man den Store zu einem beliebigen vorherigen Stand zurücksetzen kann.

Redux lässt sich als eine Implementierung des Kommandopatterns **ZITAT FEHLT** betrachten. Hierbei stellen die Actions die Kommandos dar und der Store ist der Ausführer, der auch die Kommandohistorie verwaltet. Entsprechend stellt Redux auch automatisch eine Aufwandslose Undo-Redo-Funktionalität zur Verfügung.

Da Redux die meistverwendete Bibliothek für Statemanagement ist [15], wird ihre Installation stellvertretend für Alternativen wie MobX betrachtet. Ausnahme hiervon bildet jedoch Vuex, da es eine sehr hohe Ähnlichkeit zu Redux aufweist, sich mit besonders wenig Aufwand in einem Vue-Projekt installieren lässt und es für Vue-Projekte anstelle von Redux empfohlen wird [13].

- **Komponentenbibliotheken** - Gerade für kleine Projekte, in denen keine eigenen Designer angestellt sind, kann es sich sehr lohnen, eine Komponentenbibliothek einzubinden. Da diese jeweils für ein Framework Komponenten bereitstellen muss, gibt es in der Regel für verschiedene Design Systeme jeweils eine Bibliothek pro Framework.

Stellvertretend für andere Komponentenbibliotheken wurden drei Material-Design-Bibliotheken untersucht: Material UI für React, Angular Material für Angular und Vuetify für Vue.

- **Paper.js** - Um abschließend noch eine Bibliothek zu untersuchen, die an sich keinerlei Bindung an Frameworks hat aber trotzdem in die Applikation eingebunden werden muss, wurde willkürlich als Beispiel paper.js ausgewählt. Diese Bibliothek erleichtert den Umgang mit dem HTML Canvas.

⁹D.h. sie sind seiteneffektfrei und ergeben bei gleicher Eingabe immer die gleiche Ausgabe.

3.2 Vergleich der Installationsprozesse

Beim manuellen Installieren der verschiedenen Werkzeugen und Bibliotheken sind einige Gemeinsamkeiten und vor allem in den Details viele Unterschiede aufgefallen. Auf einer allgemeineren, konzeptuellen Ebene lässt sich aber vor allem (wie bereits geschehen) zwischen Werkzeugen und Bibliotheken unterscheiden.

Allgemeinhin müssen für beides zunächst NPM-Pakete installiert werden. Daraufhin benötigen aber Werkzeuge meist die Änderung von Konfigurationsdateien, während zur Einbindung von Bibliotheken eher der produktive Code modifiziert werden muss.

3.2.1 Installationen von Werkzeugen

Zur Installation von Prettier ist beispielsweise lediglich noch die Erstellung einer Konfigurationsdatei für Prettier notwendig. Theoretisch kann auch auf diese Verzichtet werden, aber da ein Ziel des Projektes ist, dass Nutzende alles Wichtige für die meiste Arbeit eingerichtet bekommen, ist es sinnvoll, die Datei bereits zu erstellen und mit sinnvollen Standardwerten zu füllen. Darüber hinaus ist es ratsam, zur leichteren Ausführung von Prettier (wie bei den meisten anderen Werkzeugen auch) ein NPM-Script zu erzeugen. Hierfür muss die package.json-Datei modifiziert werden können.

Ähnlich ist bei der Installation von ESLint vorzugehen. In der Regel bedarf es lediglich der Installation des Paketes, der Einrichtung eines entsprechenden Scripts und der Erzeugung der Konfigurationsdatei. Leider wird der Installationsprozess bei der Benutzung von Angular dadurch verkompliziert, dass hier standardmäßig zunächst das veraltete TSLint eingebunden ist. Allerdings gibt es bereits ein NPM-Paket¹⁰, was den Wechsel auf ESLint automatisch durchführen kann.

Mit überdurchschnittlich wenig Aufwand lässt sich Sass / SCSS installieren. Sowohl Angular als auch Vue verfügen hierzu über einen Parameter bei der initialen Installation. Bei React muss man zwar das node-sass Paket installieren, aber weitere Einrichtung muss nicht vorgenommen werden.

Überdurchschnittlich viel Aufwand verursachen dahingegen Jest und Cypress. In Kombination mit React wird Jest zwar automatisch installiert und im Vue-CLI gibt es hierfür eine entsprechende Option. Allerdings setzt Angular bisher auf Karma als Testausführer und Mocha als Assertion Library, weshalb hier nicht nur eine Installation von Jest sondern auch eine Anpassung bestehender Scripte und alter Tests notwendig ist. Wie auch schon bei ESLint gibt es ein Paket, was den Umstieg bis auf die Aktualisierung der Tests selber automatisch durchführen kann. Da im Kontext dieses Projekts bei der Ausführung noch keine speziellen Tests entstanden sein können, kann diese Aufgabe hier durch ein schlichtes Austauschen von festgelegten, vorgefertigten Dateien erledigt werden.

Die Installation von Cypress ist vergleichsweise zwar simpel und benötigt lediglich das Installieren neuer NPM-Pakete, das erstellen eines Scripts in der package.json-Datei und

¹⁰<https://github.com/angular-eslint/angular-eslint>

das Erzeugen neuer Dateien. Dieser Prozess kann im Kontext von Vue sogar von der initialen Projekterstellung abgenommen werden und auch in Angular-Applikationen kann er mithilfe eines NPM-Pakets automatisiert werden. Allerdings ergibt sich in genau der Kombination (d.h. in Angular-Projekten) eine Komplikation, wenn auch Jest installiert wurde. Dieses Problem lässt sich durch Modifikation der `jest.config.js`-Datei beheben [5].

Da das Schreiben von Test sowohl für Jest als auch für Cypress sowohl in JavaScript als auch in TypeScript erfolgen kann, sollten die automatisch eingerichteten Tests in derjenigen der beiden Sprachen sein, die auch für den Produktivcode verwendet wird. Es ist also bei der Installation notwendig zu wissen, ob TypeScript verwendet wird oder nicht.

Eine letzte Besonderheit ergibt sich bei der Installation von Husky und lint-staged. Hier sind zwar ähnliche Schritte wie bei anderen Werkzeugen notwendig, aber zusätzlich wird hier eine Liste von Befehlen bzw. Skripten benötigt, die vor bestimmten Git-Prozessen ausgeführt werden sollen.

Zusammenfassend ist für die Installation von Werkzeugen also eine Kenntnis über die Verwendung von TypeScript nötig. Es muss auch bekannt sein, ob unter den anderen ausgewählten Werkzeugen solche dabei sind, die vor Git-Prozessen ausgeführt werden sollen. Darüber hinaus ist die Installation von NPM-Paketen sowie die Modifikation einiger Dateien notwendig. Diese Dateien sind entweder im JavaScript Object Notation (JSON)-Format oder sind JavaScript- / TypeScript-Dateien. Außerdem muss es möglich sein, neue Dateien mit vorbestimmtem Inhalt zu erzeugen, der sich an den JavaScript-Dialekt anpassen können muss.

3.2.2 Installationen von Bibliotheken

Auch bei der Installation von Bibliotheken gibt es viele Gemeinsamkeiten, aber hier sind deutlich größere Unterschiede zwischen den Frameworks zu vermerken. Dies ist vorallem dadurch zu begründen, dass die Benutzung der Bibliotheken innerhalb bestimmter Komponenten geschieht und daher entweder bestehende Komponenten modifiziert oder neu erstellte Komponenten eingebunden werden müssen. Aufgrund dieser Bindung an Komponenten muss das entsprechende Vorgehen an jedes Framework einzeln angepasst werden.

Ein gutes Beispiel hierfür ist die Installation von `paper.js`. Da es sich hier nur um eine spezielle Bibliothek handelt, ist sie die einzige, die in keinem Framework vorinstallierbar ist, und eignet sich daher besonders gut für diesen Vergleich. Zur sinnvollen Benutzung von `paper.js` ist es notwendig, einen Canvas auf der Webseite anzuzeigen. Sobald dieser Canvas existiert, kann `paper.js` initialisiert werden und etwas auf dem Canvas zeichnen (zu Demonstrationszwecken genügt ein einfaches Rechteck).

Hierfür lässt sich in allen drei Frameworks eine Komponente schreiben, die dann in die Hauptkomponente eingebettet werden muss. Die dafür erforderlichen Dateiveränderungen sind jedoch sehr verschieden, was am Beispiel von Vue in Listing 1 dargestellt wird.

```

1  // HelloWorld.vue
2  <template>
3    <div class="hello">
4      <!-- ... -->
5
6  +    <PaperJsExample />
7
8      <h3>Installed CLI Plugins</h3>
9      <!-- ... -->
10   </div>
11 </template>
12
13 <script lang="ts">
14   import { defineComponent } from "vue";
15 + import PaperJsExample from "@/components/PaperJsExample.vue";
16
17   export default defineComponent({
18     name: "HelloWorld",
19     props: {
20       msg: String,
21     },
22 +   components: {
23 +     PaperJsExample,
24 +   },
25   });
26 </script>
27
28 <!-- ... -->

```

Listing 1: In Vue notwendige Änderungen, um eine Komponente hinzuzufügen

Wie man hier sieht, muss die zu verwendende Komponente zum einen dort eingefügt werden, wo sie anschließend im DOM erscheinen soll. Zum anderen muss sie im Script-Block importiert werden. Danach folgt eine Besonderheit von Vue: es muss deklariert werden, dass die Komponente verwendet werden wird (vgl. Zeilen 22 - 24 in Listing 1). Dieses Attribut des Konfigurationsobjektes muss bei der ersten Komponente, die eingefügt werden soll, neu erzeugt werden, während es bei allen weiteren neuen Komponenten nur ergänzt werden muss.

Dieses eine Codebeispiel zeigt bereits, dass es schon für simple Installationen von Bibliotheken notwendig ist, framework-spezifischen Code modifizieren zu können. Hierfür wäre es hilfreich, ein tiefes Verständnis des zu modifizierenden Codes zu haben. Da aber die Struktur des vorher existierenden Codes bekannt ist (unter der Annahme, dass sich die initialen Setups nicht auf relevante Art und Weise verändern werden) und da die vorzunehmenden Änderungen sich innerhalb eines Frameworks immer ähneln, kann diese geringe Menge von Änderungen bzw. Änderungsarten auch ohne ein solches tiefes

Verständnis umgesetzt werden. Genauer wird hierauf in Kapitel 4 eingegangen.

Die Komponentenbibliotheken lassen sich mehrheitlich ohne bereits bekannte Probleme einbinden; häufig sogar automatisch bei der initialen Projekterzeugung mittels des Framework-spezifischen CLI's.

Neue Probleme weisen jedoch die Einbindung von Redux und Routern auf. Diese Bibliotheken bauen beide darauf auf, dass gewisse Informationen global verfügbar sind (in Redux der Store und in Routern die aktuelle Route).

In React ist das umsetzbar, indem die oberste Komponente von sogenannten "Providern" umgeben wird, die einen Wert und einen zugehörigen Schlüssel entgegennehmen und dann jeder ihnen unterliegenden Komponente den Wert unter dem jeweiligen Schlüssel zur Verfügung stellen. Das Umgeben der obersten Komponente ist also notwendig, damit alle Komponenten der gesamten Applikation Zugriff auf diese Werte haben. Falls dies nicht gewünscht ist, kann der zugehörige Provider verschoben werden.

In Angular ist es für eine sinnvolle Verwendung von Redux sinnvoll, einen Angular-Service einzurichten. Dies ermöglicht die einfache Erstellung eines Singleton **ZITAT FEHLT** Store-Objektes und erlaubt es gleichzeitig, mit Angular-typischen Mitteln wie RxJS¹¹-Methoden auf den Store zuzugreifen. Im Grunde genommen handelt es sich hierbei aber erneut nur um das erstellen einer Datei mit vorbestimmtem Inhalt.

Im Zusammenhang mit Vue fallen keine weiteren Probleme auf, da hier für Redux Vuex substituiert wird und sowohl Vuex als auch der Vue Router bereits automatisch installierbar sind.

3.2.3 Zusammenfassung der notwendigen Installationsschritte

Über die Werkzeuge und Bibliotheken hinweg zeigt sich, dass sich vieles über einfache Befehle installieren lässt, ohne, dass ein weiterer Aufwand betrieben werden muss. Für andere Installationen ist es ausnahmslos notwendig, gegebene Pakete von NPM installieren zu können. Wünschenswert wäre es auch, hierfür einen anderen Installer wie Yarn¹² verwenden zu können, der aber dieselben Pakete installiert. Es muss außerdem möglich sein, neue Dateien an bestimmten Orten zu erzeugen und diese mit Inhalten zu füllen, die sich nach der getroffenen Auswahl von Werkzeugen und Bibliotheken richten können.

Außerdem muss die Modifikation bestimmter Dateien ermöglicht werden. Zum einen muss es möglich sein, verschiedene JSON-Dateien zu verändern. Zum anderen muss frameworkspezifischer Code ergänzt und modifiziert werden können. Außerdem muss die Reihenfolge der Installation der Werkzeuge bzw. Bibliotheken festlegbar sein, da beispielsweise Husky und lint-staged als letzte Werkzeuge installiert werden sollten.

¹¹<https://rxjs.dev/>

¹²<https://yarnpkg.com/>

3.3 Bestimmung der erwünschten Features

Grundsätzlich soll GWA drei Aufgaben übernehmen: Nutzende sollen nach ihren Präferenzen gefragt werden und sie sollen die Möglichkeit bekommen, interaktiv mit vielen hilfreichen Informationen eine Konstellation von Werkzeugen und Bibliotheken zusammenzustellen, die sie in ihrem Projekt haben wollen.

Da aber nicht jede solche Kombination möglich oder sinnvoll ist (beispielsweise kann bis auf in Spezialfällen nur maximal ein Framework gleichzeitig benutzt werden) bzw. bestimmte Bibliotheken von anderen abhängen (Angular ist beispielsweise nicht ohne TypeScript nutzbar), ist es danach notwendig, dass die Angaben der Nutzenden validiert werden. Sind alle Wünsche umsetzbar, so ist als dritte und letzte Aufgabe die erwünschte Installation vorzunehmen.

In der Praxis soll die Abarbeitung der Aufgaben etwas mehr ineinander übergehen. Beispielsweise ist es wünschenswert, dass nach jedem Zwischenschritt bereits so viel Validierung vorgenommen wird, wie möglich, damit nicht erst nach einer langen Konfiguration festgestellt wird, dass die Auswahl gar nicht umsetzbar ist. Nach Möglichkeit sollte nach jeder unzulässigen Antwort dieselbe Frage erneut gestellt werden oder weiterhin beantwortbar bleiben, bis eine zulässige Antwort gegeben worden ist. Nur, wo das nicht möglich oder nicht praktikabel ist, soll GWA mit einem Fehlercode abgebrochen werden.

Erstmal ist es nicht notwendig, Projekte ohne ein Framework zu unterstützen. Diese Entscheidung rührt daher, dass bei frameworkunabhängigen Projekten von Anfang an viel mehr entschieden und eingerichtet werden muss. Beispielsweise muss der Compilerprozess manuell eingerichtet werden und es muss ein lokaler Entwicklungsserver aufgesetzt werden. Zudem liegen die Gründe dafür, kein Framework zu benutzen, häufig darin, dass ein sehr spezielles Projekt geplant wird, das den Rahmen von GWA ohnehin überschreiten würde (z.B. Entwicklung von Teilen der Applikation in WebAssembly). In diesen Fällen wäre die Unterstützung eines Projektes ohne Framework also auch nicht hilfreich. Da die Komplexität dieses Features aber erheblich größer wäre als bei den drei bereits diskutierten Frameworks, wir vorerst darauf verzichtet.

Wie bereits erwähnt, soll es möglich sein, den Paketmanager auszutauschen (also z.B. statt NPM Yarn zu verwenden). Diese Austauschbarkeit lässt sich leicht mit dem Strategiemuster umsetzen, da die im Rahmen von GWA genutzten Features von allen diesen Managern gleichermaßen unterstützt werden und ähnliche Schnittstellen anbieten. Nutzenden soll also zu Beginn die Wahl eines Paketmanagers gegeben werden, der dann auch genutzt werden soll.

Bei der Planung und Entwicklung von GWA muss auch bedacht werden, dass GWA nach seiner Ausführung das Projekt nicht mehr begleiten wird und es daher auch ohne GWA betrieben und gewartet werden können muss. Da zumindest im Fall von Angular und Vue die zugehörigen CLIs auch für die Pflege genutzt werden können, ist es sehr empfehlenswert, dafür zu sorgen, dass die Projekte nach der Installation über GWA damit kompatibel sind.

Daher kann es sinnvoll sein, bei der Installation so viele Schritte wie möglich über die zugehörigen CLIs zu machen. So gäbe es möglichst wenige Aktionen, die von GWA vorgenommen werden, die die Kompatibilität zu den CLIs gefährden könnten. Natürlich ist auch zu beachten, dass auf diese Weise viel Aufwand gespart werden kann, weil dann viele Aufgaben nicht manuell implementiert werden müssten, sondern auf bereits existierende und bewährte Lösungen gesetzt werden könnte.

Ein Nachteil dieses Ansatzes ist, dass er die verwendbaren Paketmanager erheblich einschränken würde. Das React-CLI unterstützt bereits nur NPM und Yarn [9] und somit könnten andere Paketmanager nicht von GWA angeboten werden.

Der alternative Ansatz wäre, für alle Frameworks die komplette Erzeugung des Frameworks zu übernehmen. Das hätte zwar den Vorteil, dass man klar regeln könnte, wann welches Werkzeug / welche Bibliothek installiert wird (sonst könnte z.B. die Komponentenbibliothek mal in der Initialeinrichtung beinhaltet sein und mal erst anschließend ergänzt werden). Außerdem hätte GWA so mehr Kontrolle über die Einheitlichkeit der erzeugten Projekte.

Nach diesen Überlegungen überwiegen die Vorteile des Ansatzes, möglichst viele der CLIs und sonstigen Werkzeuge zu verwenden. Diese Entscheidung kann natürlich für jedes einzelne Werkzeug / jede einzelne Bibliothek erneut abgewogen werden, aber im Allgemeinen wird dieser Ansatz verfolgt werden.

Ein weiterer Fokus der Entwicklung liegt auf Erweiterbarkeit. Da es in der Zukunft neue Werkzeuge und Bibliotheken geben wird, die auch über GWA installierbar sein sollen, muss es leicht möglich sein, GWA um ein Werkzeug oder eine Bibliothek zu erweitern. Dies soll über ein Erweiterungssystem erreicht werden, auf das im Folgenden noch genauer eingegangen werden wird. Erweiterungen sollen zum einen Metadaten über das entsprechende Werkzeug / die Bibliothek zur Verfügung stellen; zum anderen sollen sie aber auch über die Möglichkeit verfügen, Nutzenden Fragen zu stellen und anschließend entsprechend dieser Fragen und mit Kenntnis der anderen ausgewählten Erweiterungen eine entsprechende Teilinstallation durchzuführen.

Um die Erweiterbarkeit von GWA zu demonstrieren, sind exemplarisch einige Erweiterungen zu entwickeln. Hierbei besteht kein Anspruch auf Vollständigkeit, sondern vielmehr geht es darum, zu demonstrieren, dass das Schreiben von Erweiterungen verschiedener Arten möglich ist und wo möglich vom Grundgerüst unterstützt wird. Der Fokus bei der Entwicklung von Erweiterungen liegt daher nicht auf Quantität, sondern auf Vielseitigkeit.

3.4 Abwägung über Sonderstellungen für TypeScript und Frameworks

Aus den letzten Unterkapiteln geht hervor, dass sowohl TypeScript als auch das jeweils verwendete Framework eine wichtigere Funktion als sonstige Bibliotheken oder Werkzeuge haben. Bei TypeScript liegt das daran, dass für viele Bibliotheken je ein weiteres

NPM-Paket für die Typdefinitionen installiert werden muss und neu erzeugte Dateien sowohl eine andere Endung haben müssen als auch mit anderem Inhalt gefüllt werden müssen.

Daher wäre eine Möglichkeit, TypeScript nicht in Form einer herkömmlichen Erweiterung einzubauen, sondern fester in den Code zu integrieren. Der Code könnte dadurch leserlicher werden, da an mehr Stellen explizit klar würde, dass und warum Sonderfälle existieren. Dieses Vorgehen hätte jedoch den erheblichen Nachteil, dass in der Zukunft keine JavaScript-Dialekte mehr (ohne erheblichen Aufwand) unterstützt werden könnten, falls neue entwickelt und populär werden sollten.

Wenn also genügend Fokus auf die verständliche Entwicklung entsprechender Sonderfälle gelegt wird, ist die TypeScript-Unterstützung in Form einer normalen Erweiterung zu implementieren.

Im Fall der Frameworks ist eine ähnliche Überlegung aus anderen Gründen sinnvoll. Wie bereits erläutert worden ist, können manche Werkzeuge / Bibliotheken von bestimmten Frameworks initial mitinstalliert werden, aber von anderen nicht. Da diese Möglichkeit auch genutzt werden soll (siehe Kapitel 3.3), müssten also entweder solche Erweiterungen in den Ablauf der Installation eines Frameworks eingreifen und diesen manipulieren können, oder solche Erweiterungen müssten im Rahmen einer solchen Installation deaktiviert werden können, damit das Framework die Installation übernehmen kann und keine Konflikte entstehen.

Der Eingriff in die Installation der Frameworks wäre sehr komplex umzusetzen. Da Werkzeuge / Bibliotheken immer nach dem Framework installiert werden müssen (sofern sie eben installiert werden müssen), wäre also die Installation des Frameworks eigentlich schon dabei, bevor eine Erweiterung überhaupt dazu kommen könnte, sie zu manipulieren. Außerdem wäre eine solche Manipulation per Definition ein Seiteneffekt und würde damit, wie bereits erläutert, die Testbarkeit der Erweiterung und der Installation erschweren. Daher scheidet dieses Vorgehen aus.

Die Deaktivierung einer Erweiterung wäre zwar erstmal auch als Seiteneffekt zu betrachten. Allerdings müssen Erweiterungen bei der Installation sowieso eine Liste der anderen ausgewählten Erweiterungen erhalten. Eine Erweiterung könnte sich also aufgrund dieser Liste selber deaktivieren anstatt von außen deaktiviert zu werden. So wäre das Verhalten wieder leicht testbar.

Infolge dieser Überlegungen sind also sowohl TypeScript als auch andere Frameworks als normale Erweiterungen zu implementieren.

3.5 Planung der Erweiterungen

Um einheitlich mit allen Erweiterungen umgehen zu können, sollten diese ein gewisses Interface implementieren¹³. Neben einigen Metadaten (wie Name, Beschreibung oder

¹³In der tatsächlichen Implementierung wird aufgrund des Vorgehens nach funktionaler Programmierung ein TypeScript Type anstelle eines Interfaces verwendet. Dieser Unterschied spielt an dieser

Kategorie), die vor allem zu Anzeige- und Informationszwecken genutzt werden, müssen einige inhaltliche Eigenschaften deklarierbar sein.

Als erstes ist die Durchführung des Dialogs zu betrachten. Die initiale Auswahl der Erweiterungen kann von Außerhalb der einzelnen Erweiterungen erledigt werden. Dafür ist es neben den Metadaten notwendig, dass Erweiterungen ihre Abhängigkeiten und Exklusivitäten deklarieren können, damit die getätigte Auswahl überprüft werden kann.

Nach der initialen Auswahl der Erweiterungen muss jede der ausgewählten Erweiterungen die Möglichkeit bekommen, weitere Nachfragen zu stellen. Hierfür kann eine Erweiterung eine Methode definieren, die als Eingabeparameter alle notwendigen Objekte zum Stellen von Fragen und Analysieren der Antworten erhält und ein Objekt mit den bereits verarbeiteten Antworten des Users (also den gewünschten Einstellungen) zurück gibt. Dieser Funktion könnte auch als weiterer Eingabeparameter eine Liste aller CLI-Optionen gegeben werden, sodass zum einen bereits darüber beantwortete Fragen nicht erneut gestellt werden müssen und zum anderen diese Optionen in die Erzeugung des Einstellungsobjekts mit einfließen können.

Um die eigentliche Installation zu ermöglichen, muss es eine Methode geben, die genau diesen Schritt ausführt. Als Eingabe benötigt sie eine Liste aller ausgewählten Erweiterungen sowie deren Einstellungen (damit vor allem Frameworks die Mitinstallation anderer Bibliotheken oder Werkzeuge gemäß den angegebenen Präferenzen vornehmen können). Außerdem müssen auch sonstige über das Projekt bekannte Daten (z.B. der gewünschte Name des Projektes) übergeben werden (diese sind z.B. auch für die initiale Einrichtung des Frameworks erforderlich).

Anschließend kann es hilfreich sein, wenn abschließende Hilfsinformationen für Nutzende ausgegeben werden können, damit die nächsten Schritte mit einer Bibliothek oder einem Werkzeug erläutert werden können. Hierfür kann deshalb eine zusätzliche Methode angegeben werden, die als Parameter dieselben Daten wie die Installationsfunktion übergeben bekommt, um denselben Informationsstand zu haben und Aussagen über die durchgeführte Installation treffen zu können.

3.6 Zu verwendende Technologien

Aufgrund dessen, dass im Rahmen von GWA JavaScript-Projekte eingerichtet werden sollen, die auf NPM-Projekten aufbauen sollen, ist es sinnvoll, auch dieses Projekt innerhalb des NPM-Ökosystems aufzubauen. Somit können Entwickelnde, die eine Bibliothek oder ein Werkzeug erstellt haben, in derselben Programmiersprache auch an GWA mitarbeiten, wodurch die Pflege des CLIs erleichtert werden könnte.

Als primäre Programmiersprache wird anstelle reinen JavaScripts TypeScript zum Einsatz kommen. Die dadurch gewonnene Typsicherheit ist insbesondere aufgrund dessen nützlich, dass neue Entwickelnde beim Einstieg in das Projekt leichter verstehen können, an welcher Stelle welche Daten vorliegen.

Stelle jedoch keine Rolle.

Zur Durchführung des Dialogs (d.h. zum Stellen von Fragen und zur Darstellung von Informationen) werden `inquirer`¹⁴ und `chalk`¹⁵ verwendet. Die Wahl auf beide Bibliotheken ist dadurch gefallen, dass sie von einigen der zuvor vorgestellten Generatoren verwendet werden. Die Fragen der Vue- und des Angular-CLIs werden mithilfe von `inquirer` gestellt [12] [8] (wie bereits erwähnt, stellt CRA gar keine Fragen) und `inquirer` selbst verwendet `chalk` zum farblichen Hervorheben von (Teilen von) Ausgaben [10]. Die Verwendung von `inquirer` setzt aufgrund der benötigten Features die Verwendung von RxJS voraus. Dies wird im Kapitel der Implementierung näher erläutert.

Des weiteren werden einige Technologien verwendet, die keinen besonderen Einfluss auf das Programm selbst haben und die daher ohne besondere Recherche aufgrund von persönlicher Erfahrung und Präferenz ausgewählt werden. Diese Technologien umfassen insbesondere Jest, womit automatische Tests geschrieben und ausgeführt werden können, GitHub, worüber der Code des Projektes verwaltet wird, und Prettier.

Prettier wird aus zwei Gründen verwendet: einerseits lässt sich damit während der Entwicklung neu entstandener und modifizierter Code formatieren. Andererseits lässt sich Prettier auch programmatisch verwenden (es kann also direkt per JavaScript / TypeScript aufgerufen werden), weshalb Prettier ebenfalls zur Laufzeit verwendet werden kann, um erzeugte und modifizierte Dateien zu formatieren.

4 Implementierung

Der Fokus der Implementierung liegt darauf, eine ausreichende Infrastruktur zu schaffen, damit die Implementierung einzelner Erweiterungen keine neuen Herausforderungen aufweist, sofern sie zur Installation keine Schritte erfordern, die in der Konzeptionierung nicht berücksichtigt wurden. Die Implementierung einzelner Erweiterungen geschieht also nur exemplarisch um die Funktionalität des Grundgerüsts und des Ansatzes zu demonstrieren.

4.1 Umsetzung von Erweiterungen im Allgemeinen

Bei der Definition von und dem Umgang mit Erweiterungen sind einige allgemeine Probleme erschienen, die unabhängig von der konkreten Erweiterung waren. In diesem Unterkapitel wird auf diese Probleme eingegangen und es wird jeweils erläutert, wie sie gelöst werden konnten.

4.1.1 Probleme mit zyklischen Codeverweisen

Aus der Konzeptionierung geht hervor, dass es notwendig ist, dass Erweiterungen gegenseitig aufeinander verweisen können, beispielsweise im Rahmen der Definition von

¹⁴<https://www.npmjs.com/package/inquirer>

¹⁵<https://www.npmjs.com/package/chalk>

Abhängigkeiten oder Exklusivitäten. Außerdem wurde dargestellt, dass Exklusivitäten symmetrisch sind, d.h. dass *A* zu *B* genau dann exklusiv ist, wenn auch *B* zu *A* exklusiv ist.

Bei der Implementierung von Erweiterungen wurden die Exklusivitäten daher zunächst auch symmetrisch eingetragen. Dies führte jedoch zu dem Problem, dass die Definitionen dieser Erweiterungen zyklisch aufeinander verwiesen haben. Die Definition der Erweiterung *A* brauchte eine Referenz auf die Erweiterung *B*, aber *B* konnte nicht ohne eine Referenz auf *A* erstellt werden.

In JavaScript wird dieses Problem ohne Ausgabe einer Warnung dadurch gelöst, dass eine von beiden Referenzen den Wert `undefined` annimmt, also dass eine von beiden Erweiterungen eine Exklusivität zu `undefined` erhält.

Dasselbe Problem entsteht auch in anderen, gelegentlich notwendigen Situationen. Wenn beispielsweise ein Framework von einer Erweiterung abhängt (weil sie nicht-optional mit installiert wird, wie z.B. Angular nicht ohne TypeScript installiert werden kann) und dann die Deaktivierungsprüfung dieser Erweiterung beinhaltet, dass geprüft werden muss, ob das Framework ausgewählt wurde, erhält man wieder zyklische Verweise innerhalb der Definitionen der Erweiterungen.

Ein Lösungsansatz für dieses Problem wäre, die Definition und die „Füllung“ der Erweiterungen getrennt voneinander durchzuführen. So könnte erst für jede Erweiterung ein leeres Objekt definiert werden, auf das frei verwiesen werden könnte. Daraufhin würden diese Objekte erst mit den eigentlichen Inhalten der Erweiterung gefüllt.

Dieser Ansatz führt jedoch zu Konflikten mit TypeScript. Da dieselben Variablen verwendet werden sollen, um zunächst leere und dann gefüllte Objekte zu speichern, müssten diese Variablen entweder sehr allgemeine Typdefinitionen haben, die während der weiteren Implementierung zu größeren Aufwänden führen würde, oder sie müssten zu einem Zeitpunkt falsche Typdefinitionen besitzen. Auch dieser Ansatz ist suboptimal, da an entsprechenden Stellen Fehlermeldungen von TypeScript absichtlich deaktiviert oder umgangen werden müssten.

Stattdessen ist für jedes einzelne Problem, das zu einem zyklischen Codeverweis geführt hat, eine eigene Lösung gefunden worden. Die Identifikation von Erweiterungen wurde anhand des Namens der Erweiterung gewährleistet. Dazu muss sichergestellt sein, dass alle Namen einzigartig sind. Da Abhängigkeiten sowieso nicht zyklisch sein dürfen, ergibt sich hier kein Problem. Bei Exklusivitäten wurde auf die symmetrische Spiegelung verzichtet und so konnte durch geschickte Deklaration von Exklusivitäten jeder dadurch bedingte zyklische Codeverweis eliminiert werden.

Dieser Ansatz könnte noch dadurch erweitert werden, dass die Exklusivitäten an separater Stelle (nach der initialen Definition aller Erweiterungen) definiert werden. Da Exklusivitäten ohnehin optional sind (bei der Konzeptionierung wurde festgelegt, dass eine Erweiterung keine Exklusivitäten haben muss), würde dies sich nicht weiter auf die Typdefinition von Erweiterungen auswirken. Außerdem ist auch eine nachträgliche Herstellung der Symmetrie möglich, wenn sie für einen Algorithmus zwingend notwendig ist.

Da durch die nun umgesetzte Lösung das Problem der undefinierten Abhängigkeiten / Exklusivitäten nicht komplett ausgeschlossen werden konnte und zusätzlich noch notwendig geworden ist, dass die Namen von Erweiterungen stets einzigartig sind, wurden Überprüfungen eingeführt, die die Sinnhaftigkeit der definierten Erweiterungen sicherstellen sollen. Konkret wird hier sichergestellt, dass in keiner Liste von Abhängigkeiten oder Exklusivitäten der Wert `undefined` vorkommt und dass alle Erweiterungen einzigartige Namen besitzen.

Diese Sinnhaftigkeitsüberprüfung ließe sich leicht in die normale Ausführung des CLIs einbinden, sodass vor jeder Ausführung gewährleistet ist, dass die Erweiterungen sinnvoll definiert sind. Da sich die Erweiterungen aber nach dem Kompilierprozess nicht mehr verändern, würde es auch ausreichen, die Überprüfungen bei der Gelegenheit auszuführen. Dadurch würde bei Nutzenden etwas Rechenleistung gespart werden.

Eine weitere Möglichkeit ist die Ausführung der Überprüfungen im Rahmen der automatisierten Tests. Diese Variante hat den Nachteil, dass es möglich wäre, eine fehlerhafte Version von GWA zu veröffentlichen. Da die Ausführung der Tests im Rahmen dieser Arbeit aber bereits nach jeder Änderung automatisch geschieht und sie für zukünftige Fremdbeiträge ebenfalls über GitHub Actions¹⁶ automatisiert werden kann, ist dieses Risiko als vernachlässigbar einzustufen.

Da die Einbindung der Überprüfungen in die automatisierten Tests erheblich einfacher ist als die Einbindung in den Kompilierprozess und die Differenz des bleibenden Risikos nicht bedeutend ist, wurde die Einbindung in die automatisierten Tests vorgenommen.

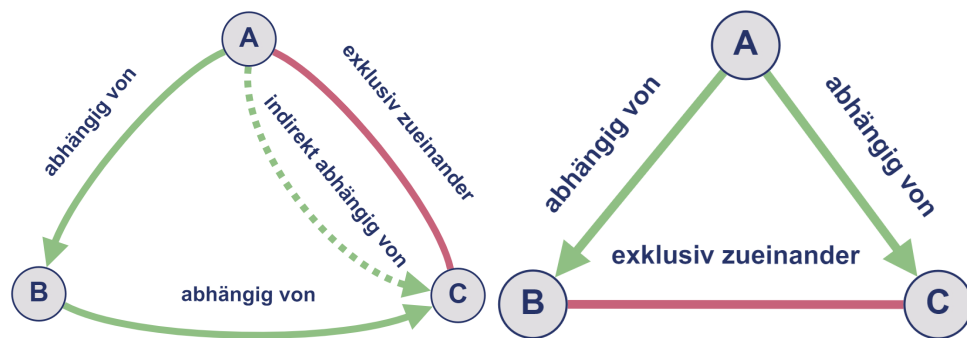
4.1.2 Prüfung, ob eine Erweiterung ausgewählt worden ist

Für verschiedene Features ist es notwendig, dass Erweiterungen feststellen können, ob andere Erweiterungen ebenfalls ausgewählt worden sind (z.B. damit ESLint seine Installation aussetzen kann, falls React ausgewählt wurde, da React bereits ESLint installiert). Diese Feststellung wird unter anderem im Rahmen von Schleifen aufgerufen und sollte daher eine möglichst geringe Laufzeit besitzen.

Ideal wäre daher eine Laufzeit in $\mathcal{O}(1)$. Diese könnte beispielsweise erreicht werden, indem eine Hashmap verwendet wird, sofern die zugehörige Hashfunktion in $\mathcal{O}(1)$ läuft. In dieser Hashmap würde dann für jede Erweiterung als bool'scher Wert gespeichert werden, ob die Erweiterung ausgewählt wurde.

Da aber beim Start des Programmes alle möglichen Erweiterungen bekannt sind, können diesen eindeutige IDs zugewiesen werden, die dann als Index eines Arrays gelten können, in dem Daten zu den Erweiterungen gespeichert werden. Da die ID jeder Erweiterung bekannt ist, kann so trivialerweise in $\mathcal{O}(1)$ festgelegt und überprüft werden, ob die Erweiterung ausgewählt wurde.

¹⁶<https://github.com/features/actions>



- (a) Eine Möglichkeit, wie ein Widerspruch zwischen indirekter Abhängigkeit und Exklusivität entstehen kann.
- (b) Hier entsteht ein Widerspruch dadurch, dass eine Erweiterung zwei Abhängigkeiten hat, die zueinander exklusiv sind.

Abbildung 1: Zwei Graphen zur Veranschaulichung von Konfliktmöglichkeiten zwischen Abhängigkeiten und Exklusivitäten.

4.1.3 Überprüfung der Umsetzbarkeit der Restriktionen

Während der Implementierung verschiedener Erweiterungen ist es versehentlich passiert, dass eine Konstellation von Restriktionen definiert wurde, die einen inneren Widerspruch enthält.

Da zu erwarten ist, dass verschiedene Leute ohne Kenntnis aller anderen Erweiterungen neue Erweiterungen erschaffen oder bestehende Erweiterungen modifizieren werden, sollte verhindert werden, dass versehentlich solche Kombinationen von Restriktionen erzeugt werden. Daher gilt es, mindestens zu jeder Kompilierzeit alle Restriktionen auf ihre Umsetzbarkeit zu überprüfen und im Falle eines Widerspruchs eine Warnung auszugeben. Da im verwendeten Projektumfeld die Ausführung während der Kompilierzeit sich als etwas schwieriger gestaltet hätte und die Überprüfung dieser Restriktionen nicht merklich lange dauert, wurde diese Überprüfung in die Ausführung der automatischen Tests verschoben.

Ein triviales Beispiel einer unmöglichen Kombination von Restriktionen wäre ein Paar von Abhängigkeiten A und B , wobei A und B exklusiv zueinander sind und gleichzeitig A von B abhängt. Es können in diesem Beispiel niemals beide Anforderungen gleichzeitig erfüllt werden.

Natürlich lassen sich aber auch kompliziertere Beispiele erzeugen. In Abbildung 1 werden zwei solche Konstellationen dargestellt. Bei der einen hängt A von B und B von C ab, aber A ist zu C exklusiv. Diese Kombination von Anforderungen lässt sich ebensowenig wie das erste Beispiel gleichzeitig erfüllen, aber das Problem ist hier nicht sofort offensichtlich. Man muss hierfür erkennen, dass Abhängigkeiten transitiv sind. Ist also die Erweiterung A von B und B von C abhängig, so ist auch indirekt A von C abhängig.

Der zweite in Abbildung 1 dargestellte Konflikt beruht im Gegensatz zu den bishe-

rigen Beispielen nicht darauf, dass eine Erweiterung zugleich (indirekt) abhängig von und exklusiv zu einer anderen Erweiterung ist. Hier wird eine weitere Kategorie von Problemen dargestellt: sind zwei (indirekte) Abhängigkeiten einer Erweiterung zueinander exklusiv, so können nicht beide dieser Abhängigkeiten erfüllt werden. Auch solche Konfigurationen sind also unzulässig.

Aus diesen fachlichen Überlegungen heraus ergibt sich eine erste Lösung. Für jede Erweiterung E sind zunächst alle direkten sowie indirekten Abhängigkeiten zu bestimmen. Für alle diese Abhängigkeiten ist dann zu prüfen, dass zum einen die Abhängigkeit A nicht zu E exklusiv ist, aber auch dass A zu keiner weiteren (indirekten) Abhängigkeit A' von E exklusiv ist.

Bei näherer Betrachtung dieser Lösung lässt sich jedoch einiges an Ineffizienz feststellen. Zum einen werden für jede Erweiterung erneut die transitiven Abhängigkeiten berechnet. Aufgrund eben dieser Transitivität sind jedoch die Berechnungen für alle Erweiterungen, die Abhängigkeit einer anderen Erweiterung sind, redundant. Im schlimmsten Fall (nämlich, wenn eine Erweiterung von insgesamt n Erweiterungen von allen anderen Erweiterungen (indirekt) abhängt) ist also genau eine der durchgeführten Berechnungen wirklich notwendig und $n - 1$ Berechnungen werden unnötigerweise durchgeführt. Aus ähnlichem Grund ist auch die Überprüfung der Exklusivität zweier paarweise unabhängiger (indirekter) Abhängigkeiten häufig überflüssig.

Die Lösung dieser Probleme ergibt sich aus einer theoretischeren Betrachtung des Problems. Wie bereits aus der Verwendung des Begriffs der Transitivität hervorgeht, lassen sich Abhängigkeit und Exklusivität als mathematische Relationen über der Menge aller Erweiterungen auffassen. Hierbei ist besonders hervorzuheben, dass die Exklusivität symmetrisch ist (ist A zu B exklusiv, so ist auch B zu A exklusiv) während Abhängigkeit nicht symmetrisch ist (im Gegenteil: bei der anfänglichen Analyse von möglichen Bibliotheken, Frameworks etc. ergab sich, dass Abhängigkeit nie symmetrisch zu sein scheint). Allerdings ist Exklusivität a priori nicht transitiv (auch, wenn A zu B und B zu C exklusiv ist, können A und C zusammen verwendet werden), während Abhängigkeit sehr wohl transitiv ist (wie bereits erläutert).

Vor diesem Hintergrund lässt sich erkennen, dass die Bestimmung der (indirekten) Abhängigkeiten der Bestimmung der Transitiven Hülle gleichkommt. Diese kann mittels des Floyd-Warshall-Algorithmus (in der Warshall-Variante) berechnet werden [19]. Hierfür muss zunächst ein gerichteter Graph erzeugt werden, in den alle deklarierten Abhängigkeiten als Kante eingefügt werden. Von diesem Graphen wird dann die transitive Hülle bestimmt, in der zwei Knoten A und B genau dann durch eine (von A nach B gerichtete) Kante verbunden sind, wenn die Erweiterung A von B abhängt.

Auch die Relation der Exklusivität lässt sich in einen Graphen überführen. In diesem Graphen gibt es ebenfalls pro Erweiterung einen Knoten und jede Exklusivität wird als Kante dargestellt. Aufgrund der Symmetrie der Exklusivität kann dieser Graph aber ungerichtet sein.

Das Problem der Überprüfung der Restriktionen reduziert sich nun darauf, sicherzustellen, dass es zwischen zwei Knoten (also zwischen zwei Erweiterungen) in maximal

einem der beiden Graphen eine Kante gibt, wobei die Richtung keine Rolle spielt (denn wenn die Knoten exklusiv zueinander sind, darf es zwischen beiden keine Abhängigkeit geben – egal, in welche Richtung).

Anders formuliert, darf es im Graphen der Exklusivitäten keine Kante $\{A, B\}$ geben, für die in der transitiven Hülle der Abhängigkeiten die Kante (A, B) oder die Kante (B, A) existiert. Aufgrund dessen, dass die transitive Hülle gerichtet ist, gibt es darin doppelt so viele Kanten wie in dem Graphen der Exklusivitäten. Außerdem liegt sie als Adjazenzmatrix vor, während die Exklusivitäten als Kantenliste vorliegen. Somit kann bei m Exklusivitäten in $\mathcal{O}(m)$ über die Exklusivitäten iterieren und jeweils in $\mathcal{O}(1)$ die Existenz einer (transitiven) Abhängigkeit prüfen. Die Umkehrung, also die Iteration über Abhängigkeiten und dann die Prüfung der Existenz einer Exklusivität, würde bei n Erweiterungen einen Aufwand von $\mathcal{O}(n^2)$ verursachen. Wenn man vorher die Kantenliste der Exklusivitäten in $\mathcal{O}(n^2)$ in eine Adjazenzmatrix überführt, ist die anschließende Existenzprüfung auch wieder in $\mathcal{O}(1)$ möglich, aber insgesamt

Der Algorithmus von Floyd-Warshall sorgt dafür, dass diese Überprüfung eine asymptotische Laufzeit von $\mathcal{O}(n^3)$ hat. Da die oben beschriebenen Probleme in beliebiger Tiefe von Abhängigkeiten auftreten können, ist jedoch die Bestimmung der transitiven Hülle nicht vermeidbar. Allerdings ist damit zu rechnen, dass die Anzahl aller Erweiterungen stets kleiner als 100 sein wird (andernfalls würde die Benutzbarkeit des Programms möglicherweise stark eingeschränkt). Daher ist diese Laufzeit in diesem Fall als unbedenklich einzustufen.

Da auch diese Überprüfung sich lediglich auf die Definitionen der Erweiterungen verlässt, wird auch sie in die Ausführung der automatisierten Tests eingebunden, damit sichergestellt werden kann, dass keine fehlerhaften Erweiterungsdefinitionen veröffentlicht werden.

4.1.4 Überprüfung der getroffenen Auswahl von Erweiterungen

Um die getroffene Auswahl an Erweiterungen auf die Einhaltung aller Restriktionen hin zu überprüfen, wurde ebenfalls zunächst eine fachliche Lösung gefunden. Für jede der ausgewählten Erweiterungen sind alle Abhängigkeiten und Exklusivitäten durchzugehen. Ist eine Abhängigkeit nicht mit ausgewählt worden oder ist eine Exklusivität mit ausgewählt worden, so ist (genau dann) die Kombination ungültig.

Die Laufzeit dieses Vorgehens liegt für n mögliche Erweiterungen in $\mathcal{O}(n^3)$ (mit Möglichkeit der Reduktion auf $\mathcal{O}(n^2)$). Dies liegt daran, dass zunächst über jede der (maximal n) ausgewählten Erweiterungen und dann über jede der Abhängigkeiten und Exklusivitäten iteriert werden muss. Da eine Erweiterung niemals zu einer Abhängigkeit exklusiv sein kann, beträgt die Summe der Abhängigkeiten und Exklusivitäten maximal n . Somit ist bereits eine Laufzeit von $\mathcal{O}(n^2)$ erreicht worden.

Nun bleibt noch die Prüfung, ob eine Abhängigkeit bzw. Exklusivität ausgewählt worden ist oder nicht, d.h. ob sie in der Liste der ausgewählten Erweiterungen liegt oder nicht. Im Falle einer normalen Liste läge die Laufzeit dieser Überprüfung bei $\mathcal{O}(n)$, da

die im schlimmsten Fall die gesamte Liste überprüft werden müsste. Allerdings kann man diese Information auch als Liste bool'scher Werte speichern, wobei unter jedem Index genau eingetragen ist, ob die Erweiterung mit demselben Index in der Liste aller möglichen Erweiterungen ausgewählt worden ist. So wäre die Überprüfung der Auswahl einer einzelnen Erweiterung in $\mathcal{O}(1)$ möglich und würde die Gesamtlaufzeit nicht weiter beeinflussen.

Ähnlich wie bei der Überprüfung der Umsetzbarkeit aller Restriktionen lässt sich jedoch auch für dieses Problem eine graphentheoretische Lösung finden. Da bereits Graphen bestimmt wurden, in denen Kanten für Abhängigkeit und Exklusivität eingetragen sind, können diese zunächst kopiert und dann zur Überprüfung dieses Problems verwendet werden. Die Kopie erfolgt in $\mathcal{O}(n^2)$, da eine $n \times n$ -Adjazenzmatrix und eine Kantenliste mit maximal Länge n^2 kopiert werden müssen.

Im Graphen der Abhängigkeiten sind Kanten nun als noch unerfüllte Bedingungen anzusehen. Die Streichung einer Kante bedeutet somit, dass die entsprechende Bedingung erfüllt ist. Nun ist über jede mögliche Erweiterung zu iterieren. Falls sie ausgewählt worden ist, so sind auf sie eingehende Kanten aus dem Graphen der Abhängigkeiten zu streichen. Andernfalls (d.h. wenn sie nicht ausgewählt worden ist) ist ihr Knoten mit samt seiner ausgehenden Kanten zu streichen. Es sind genau dann alle Abhängigkeiten erfüllt, wenn nach der Iteration über alle möglichen Erweiterungen keine Kanten mehr in dem Graphen vorhanden sind.

Ähnlich ist für die Exklusivitäten zu verfahren. In diesem Graphen bedeutet die Existenz einer Kante, dass die zugehörige Exklusivität in der aktuellen Auswahl noch nicht ausgeschlossen werden konnte. Während über alle Erweiterungen iteriert wird, kann also für jede nicht ausgewählte Erweiterung der zugehörige Knoten zusammen mit seinen (ungerichteten) Kanten gestrichen werden. Auch die Exklusivitäten sind alle genau dann erfüllt, wenn sich am Ende der Iterationen keine Kante mehr im Graphen befindet.

Der erste Aspekt der Laufzeit dieser Verfahren ergibt sich dadurch, dass die Graphen in verwendbarer und modifizierbarer Form vorliegen müssen (d.h. als neu angelegte Adjazenzmatrizen). Die im letzten Kapitel berechnete Adjazenzmatrix der Abhängigkeit muss also lediglich kopiert werden, während der Graph der Exklusivität in eine Adjazenzmatrix überführt werden muss. Dies kann in $\mathcal{O}(n^2)$ geschehen.

Danach muss über jede Erweiterung iteriert werden und ggf. (also falls sie nicht ausgewählt wurde) die zugehörige Zeile und Spalte der Adjazenzmatrix modifiziert werden. Anschließend muss die gesamte Matrix auf die Existenz von Kanten hin überprüft werden. Beide Schritte geschehen nacheinander in jeweils $\mathcal{O}(n^2)$. Somit fordert der gesamte Prozess eine Laufzeit von $\mathcal{O}(n^2)$.

Alternativ zu den Adjazenzmatrizen könnten auch Kantenlisten verwendet werden. Dann müsste aber für die Löschung von Knoten, die in beiden Prozessen maximal für jede Erweiterung vorkommen könnte, jeweils die gesamte Liste durchlaufen werden. Es entstünde also auch hier ein Aufwand von $\mathcal{O}(n^2)$.

Die Betrachtung der Graphentheoretischen Lösung ergibt also in diesem Fall keine Verbesserung der Laufzeit. Nach persönlichem Empfinden ist die erste Lösung jedoch

leichter zu verstehen und wurde angesichts der gleichen Laufzeit aufgrund dieses Kriteriums ausgewählt und umgesetzt.

4.2 Umsetzung des Dialogs

Der Dialog mit Nutzenden lässt sich grob in drei Aufgaben unterteilen. Die erste Aufgabe ist das Untersuchen der CLI-Argumente. In diesem Schritt werden keine Ausgaben erzeugt, aber da er auf einer manuellen Eingabe operiert, zählt er trotzdem zum Prozess des Dialogs.

Mit den vorgegebenen Optionen werden dann, sofern notwendig, weitere allgemeine Fragen gestellt. Hier haben Nutzende die Möglichkeit, den Namen des Projektes anzugeben, einen Paketmanager und die gewünschten Erweiterungen auszuwählen. Ist eine dieser Fragen bereits in den CLI-Argumenten beantwortet worden, so kann die entsprechende Frage übersprungen werden.

Als letzten Schritt des Dialogs sind dann weitere Fragen der einzelnen Erweiterungen zu stellen. Auch diese können bereits beantwortet worden sein und werden ggf. übersprungen.

4.3 Analyse der CLI-Argumente

Mithilfe der bereits erwähnten Bibliothek „commander“ können die CLI-Argumente mit sehr geringem Aufwand eingelesen und in gewissem Rahmen validiert werden. Dazu ist es notwendig, alle möglichen Argumente und ggf. die Typen der erwarteten zugehörigen Werte zu deklarieren.

Für die Angabe eines bool'schen Wertes reicht dann ein Parameter ohne zugehörigen Wert (z.B. `--use-some-library`), während Angabe einer Wahl von bestimmten Optionen zusätzlich die Angabe eines Wertes benötigt (z.B. `--package-manager npm`). Commander erzeugt bei der anschließenden Analyse der beim Befehlsaufruf übergebenen Argumente ein Objekt, bei dem jedem angegebenen Argument der entsprechende Wert (oder, falls kein Wert angegeben wurde, der bool'sche Wert `true`) zugewiesen ist. Wenn eine Option beim Befehlsaufruf nicht angegeben wurde, so wird sie in das ausgegebene Objekt nicht aufgenommen.

Bool'sche Optionen können daher nicht durch Weglassung auf den Wert `false` gesetzt werden. Um sie trotzdem negierbar zu machen, kann man eine weitere Option mit dem Präfix `no-` erstellen (z.B. `--no-use-some-library`). Diese wird von commander automatisch als Negation der zugehörigen Option (in dem Beispiel also der Option `--use-some-library`) erkannt. Natürlich kann dieselbe Funktionalität auch mit anderen Benamungen umgesetzt werden; dies ist dann aber mit höherem Aufwand verbunden, da die zugehörige Logik von Hand implementiert werden muss.

Um alle Fragen, die im Laufe des Dialogs gestellt werden könnten, durch solche CLI-Argumente abzudecken, muss es möglich sein, dass Erweiterungen ebenfalls Argumente

deklarieren können. Dies wird umgesetzt, indem Erweiterungen eine Funktion definieren können, die auf dem überreichten `commander`-Objekt frei Argumente definieren kann.

Daraufhin werden die Argumente von `commander` analysiert. Die allgemeinen Metadaten werden herausgefiltert und an die Stellung der entsprechenden Fragen weitergegeben. Alle weiteren Argumente können im allgemeinen Teil jedoch nicht weiter verarbeitet werden und müssen daher als Gesamtheit im Rahmen des weiteren Dialogs an die Erweiterungen überreicht werden (siehe Abschnitt 4.3.2).

Vor der Weitergabe der per CLI-Argumente übergebenen allgemeinen Daten müssen diese jedoch validiert werden. Insbesondere bezieht sich dies auf die Überprüfung der Auswahl der Erweiterungen (siehe Kapitel 4.1.4). Sofern keine Erweiterung per CLI-Argument ausgewählt wurde, kann diese Validierung übersprungen werden, da die zugehörige Frage später gestellt werden muss und die Validierung dabei erfolgen kann.

Weniger eindeutig ist jedoch, wie mit der Angabe einer oder mehrerer gewünschter Erweiterung(en) per CLI-Argumente umzugehen ist, da nicht eindeutig erkennbar ist, ob die so getroffene Auswahl vollständig sein soll. Falls sie nicht vollständig wäre, müsste die zugehörige Frage gestellt werden, was aber eine programmatische Verwendung von GWA deutlich erschweren würde. Daher ist davon auszugehen, dass die per CLI-Argumente getroffene Auswahl an Erweiterungen vollständig ist, sobald mindestens eine Erweiterung ausgewählt wurde. In diesem Fall ist dann die Überprüfung dieser Auswahl durchzuführen und im Fehlerfall ist die Ausführung von GWA abubrechen.

4.3.1 Stellen allgemeiner Fragen

In diesem Abschnitt des Dialogs werden sowohl Metadaten des zu erzeugenden Projektes gesammelt als auch die Auswahl der zu installierenden Erweiterungen getroffen. Die Sammlung der Metadaten ist ohne besonderen Aufwand umsetzbar, da alle hierfür relevanten Fragetypen (Freitext-Eingabe, Auswahl eines von n Elementen) von `inquirer` zur Verfügung gestellt werden und direkt verwendet werden können.

Bemerkenswert ist lediglich, dass vor der Frage, welcher Paketmanager verwendet werden soll, geprüft wird, welche Paketmanager überhaupt installiert sind. Auch, wenn nur ein Paketmanager installiert ist und somit gar keine Auswahl getroffen werden kann, wird die Frage gestellt und erhält lediglich ergänzende Bemerkungen, dass andere Paketmanager nicht zur Verfügung stehen. Diese Entscheidung wurde getroffen, um unerfahrene Entwickelnde zumindest darüber zu informieren, dass es weitere Paketmanager gibt. Außerdem kann diese Frage leicht übersprungen werden, indem die entsprechende Entscheidung per CLI-Argument `--package-manager` spezifiziert wird. Um dies weiter zu erleichtern, wurde auch der Alias `-p` eingerichtet.

Die Frage zur Auswahl der zu installierenden Erweiterungen gestaltet sich etwas schwieriger. Sie muss von einer allgemeinen Stelle aus gestellt werden, da sie erweiterungsübergreifend ist, aber muss dennoch erweiterungsspezifische Inhalte darstellen können. Um auch diese Frage möglichst verständlich und informativ zu gestalten, müssen Erweiterungen neben ihrem Namen auch eine Kurzbeschreibung, einen Link zu mehr In-

formationen und die Kategorie der Erweiterung angeben. Name, Kurzbeschreibung und Link werden in die Frage eingebaut, während die Kategorie lediglich für eine sinnvolle Gruppierung und Trennung der Erweiterungen genutzt wird.

Inquirer ermöglicht bei allen Fragen die Angabe einer Funktion, die eine Antwort auf die Frage entgegennimmt und diese validiert. Auf diesem Weg ist es möglich, die in Kapitel 4.1.4 beschriebene Überprüfung der getroffenen Auswahl durchzuführen. Wird ein Fehler festgestellt, so wird dieser von inquirer ausgegeben und die Frage wird beibehalten, bis eine Antwort die Validierung besteht. So wird garantiert, dass die weiteren Schritte von GWA nicht ausgeführt werden können, bis eine gültige Auswahl von Erweiterungen getroffen wurde.

4.3.2 Übergabe des Dialogs an Erweiterungen

Da auch Erweiterungen in der Lage sein müssen, im Rahmen des Dialogs Fragen zu stellen, können sie eine Funktion definieren, die nach dem Stellen eventueller Fragen ein Objekt zurück gibt, das die aus den Fragen resultierende Konfiguration enthält. Damit keine bereits beantworteten Fragen gestellt werden und die per CLI-Argumente spezifizierten Optionen bei der Erzeugung dieses Konfigurationsobjektes berücksichtigt werden können, müssen alle CLI-Argumente ebenfalls an diese Funktion übergeben werden.

Um den Dialog möglichst übersichtlich zu gestalten, ist es sinnvoll, die Fragen einzelner Erweiterungen visuell voneinander zu trennen, damit Nutzenden bewusst ist, worauf sich jede aktuelle Frage bezieht. Hierfür soll vor den Fragen jeder Erweiterung der Name der Erweiterung als Überschrift der folgenden Fragen ausgegeben werden. Außerdem soll zwischen den Fragen der letzten Erweiterung und der Überschrift der Fragen der nächsten Erweiterung eine Leerzeile eingefügt werden.

Um diese Formatierung zu garantieren und möglichst wiederholungsfreien Code zu erzeugen, ist diese Formatierung außerhalb der Erweiterungen umzusetzen. Dies hat zudem den Vorteil, dass Randfälle (z.B. Ausgaben vor der ersten / nach der letzten Erweiterung) leichter betrachtet werden können, da Erweiterungen beim Stellen von Fragen nicht wissen, ob vor oder nach ihnen Fragen gestellt wurden / werden.

Diese Umsetzung ist jedoch fehlerhaft, falls aufgrund von CLI-Argumenten bestimmte Fragen ausgesetzt werden. Wenn alle Fragen einer Erweiterung ausgesetzt werden, dann sollte für die Erweiterung auch keine Leerzeile oder Überschrift ausgegeben werden, was bei dem bisher beschriebenen Ansatz aber geschehen würde.

Um dieses Problem zu lösen, wird an die Funktionen zum Stellen von Fragen als weiterer Parameter eine Funktion übergeben, die zunächst nur inquirer ummantelt und jeden Aufruf direkt weiterleitet. Vor der Weiterleitung eines Aufrufes prüft sie jedoch, wie viele Fragen gestellt werden. Sobald eine Erweiterung (mindestens) eine Frage stellt, gibt sie vor dem Weiterleiten die trennende Leerzeile sowie die Überschrift aus. Stellt eine Erweiterung also keine Fragen, wird beides nicht ausgegeben.

Dieses Vorgehen hat auch den Vorteil, dass mit besonders geringem Aufwand im Rahmen der automatischen Tests das Stellen von Fragen gesteuert werden kann, ohne, dass

die Ausgaben der Tests von Ausgaben von `inquirer` unterbrochen werden. Hierfür muss lediglich die übergebene Funktion darauf verzichten, `inquirer` aufzurufen und im Rahmen der Tests gewünschte Antworten ohne das Stellen entsprechender Fragen zurück geben.

4.4 Installation von Erweiterungen

Nachdem alle benötigten Informationen eingeholt und verifiziert worden sind, ist nun die Installation der Erweiterungen durchzuführen. Hierfür wird in einer bestimmten Reihenfolge (siehe Kapitel 4.4.1) für jede der ausgewählten Erweiterungen zunächst geprüft, ob sie übersprungen werden soll. Falls dem nicht so ist, wird dann eine von ihr definierte Funktion zur Installation aufgerufen. Diese erhält als Parameter alle gesammelten Metadaten sowie die zum gewählten Paketmanager gehörige Strategie und eine Liste mit allen ausgewählten Erweiterungen mitsamt ihren speziell getroffenen Einstellungen.

Um Nutzenden den Fortschritt von GWA mitzuteilen, wird vor dem Aufruf der Installationsfunktion ausgegeben, dass nun mit der Installation der entsprechenden Erweiterung begonnen wird. Diese Meldung wird nicht ausgegeben, wenn die Erweiterung übersprungen wird. Dies ist der Grund, warum das Überspringen der Installation einer Erweiterung gesondert überprüft werden muss, anstatt ggf. in der Installationsfunktion nichts auszuführen.

Im Anschluss an alle Installationen haben Erweiterungen die Möglichkeit durch eine von ihnen definierte Funktion weitere Informationen auszugeben. Dies kann beispielsweise genutzt werden, um erste Schritte mit einem Framework zu erläutern oder erneut auf die Dokumentation zu verweisen.

4.4.1 Bestimmung der Installationsreihenfolge

Wie bereits in der Konzeptionierung erläutert wurde, muss es möglich sein, die Installationsreihenfolge der Erweiterungen festzulegen. Dabei ist nicht die Festlegbarkeit der gesamten Reihenfolge relevant, sondern es müssen lediglich gewisse Bedingungen eingehalten werden.

Beispielsweise muss das Framework zuerst installiert werden, da es die notwendige Ordnerstruktur erzeugt. Die Installation von `husky` und `ESLint` sollte sehr spät erfolgen, damit alle im Projekt existierenden Dateiendungen bei entsprechenden Skripten berücksichtigt werden können. Dies sind also zwei Beispiele für „globale“ Bedingungen die von einer Erweiterung an den gesamten Kontext gestellt werden.

Es gibt aber auch eine weitere Art von Bedingung, bei der eine Erweiterung nicht vor einer anderen Erweiterung installiert werden darf. Ein solches Beispiel ist, dass `husky` nach `Prettier` und `ESLint` installiert werden muss, da sonst bei den durch `husky` eingerichteten Automatisierungen die anderen Werkzeuge nicht einbezogen werden. Dieses Konzept ähnelt dem der Abhängigkeiten, kann jedoch ausdrücklich nicht durch die bereits existierenden Abhängigkeiten zwischen Erweiterungen abgebildet werden. Sonst

müsste beispielsweise TypeScript vor Angular installiert werden, da Angular von TypeScript abhängig ist. Dies ist jedoch nicht möglich, da (wie bereits erläutert) Frameworks als allererstes installiert werden müssen.

Obwohl sich diese Anforderungen wie geschehen kategorisieren lassen, stellen sie eher eine Ansammlung von Einzelfällen dar. Neben Frameworks gibt es keine Erweiterungen, die als erstes oder letztes zu installieren sind. Die Anforderung von Husky und ESLint ist nach bisherigem Kenntnisstand ebenfalls ein Einzelfall. Da nur begrenzt viele Bibliotheken und Werkzeuge analysiert wurden, kann es darüber hinaus sein, dass es noch weitere Arten von Bedingungen an die Installationsreihenfolge gibt, die für zukünftige Erweiterungen relevant werden. Die einzige Anforderung, die häufig festgestellt wurde, war die, dass eine Erweiterung nicht vor einer oder mehreren anderen installiert werden darf.

Alternativ kann durch händisches Festlegen der Installationsreihenfolge gewährleistet werden, dass sämtliche Anforderungen erfüllt werden. Dieser Ansatz hat den Nachteil, dass er unübersichtlicher ist und leichter bei der Ergänzung neuer Erweiterungen Fehler gemacht werden können. Dafür ist der Implementierungsaufwand deutlich geringer.

Insbesondere, da die Implementierung der bisher beobachteten Anforderungen in Form von inhaltlichen Regeln keine Zukunftssicherheit bietet und dennoch deutlich komplexer wäre, wurde nach der zweiten vorgestellten Methode vorgegangen. Alle Auswählbaren Erweiterungen wurden in einem Array derart aufgelistet, dass die Reihenfolge der Erweiterungen in dem Array die Installationsreihenfolge festlegt. Dieses Array wird zugleich überall dort verwendet, wo eine vollständige Liste aller Erweiterungen benötigt wird.

4.4.2 Erzeugung von TypeScript- und JavaScript-Dateien

Im Rahmen der Installation vieler Erweiterungen sind JavaScript- bzw. TypeScript-Dateien (mit entsprechendem Code) zu erzeugen. Hierfür lassen sich trivialerweise Beispieldateien anlegen, von denen dann gemäß der getroffenen Auswahl die richtige ausgewählt und in das neue Projekt kopiert wird. Dieser Ansatz hat jedoch den Nachteil, dass er viel Code verursachen würde, der (bis auf die Existenz von Typen) dupliziert wäre. Das würde die Wartung etwas aufwändiger und unangenehmer machen.

Zudem könnten zwei in diesem Sinne äquivalente Dateien versehentlich verschieden bearbeitet werden, sodass in nur einer der beiden Dateien ein Fehler eingeführt wird. Derartige Fehler können nur durch sehr ausgiebiges Testen gefunden werden und bleiben daher leicht unbemerkt.

Aus diesen Gründen wurde ein Ansatz entwickelt, der aus nur einer Vorlage sowohl entsprechende JavaScript- als auch TypeScript-Dateien erzeugen kann. Mithilfe des TypeScript-Kompilers, der ohnehin aus Entwicklungsgründen in das Projekt eingebunden ist, kann TypeScript-Code zu JavaScript-Code umgewandelt werden. Der Compiler akzeptiert dabei ein Konfigurationsobjekt, in dem festgelegt werden kann, mit welcher Version des EcmaScript-Standards das JavaScript kompatibel sein soll. Indem hier der neuste Standard spezifiziert wird, ähnelt der resultierende JavaScript-Code maximal dem

ursprünglichen Code.

Beim Kompilieren werden jedoch zuvor vorhandene Leerzeilen entfernt, die aus Formatierungs- und Verständlichkeitsgründen relevant sind. Viele dieser Zeichen lassen sich wieder einfügen, indem das Kompilat mit Prettier formatiert wird. Bei einigen Durchläufen dieses Prozesses mit verschiedenen Dateien ist jedoch aufgefallen, dass Leerzeilen, die zur gedanklichen Trennung von Codeblöcken eingefügt wurden, nicht von Prettier wiederhergestellt werden konnten.

Um auch diese Leerzeilen in die generierten JavaScript-Dateien aufnehmen zu können, wird nach dem Kompilieren das Kompilat mit dem ursprünglichen Code verglichen. Dadurch werden entfernte Leerzeichen / -zeilen gefunden und wieder in das Kompilat eingefügt. Hierbei werden jedoch in bestimmten Fällen zu viele Zeichen eingefügt (z.B. werden Leerzeilen, die aufgrund von Typdefinitionen notwendig waren, eingefügt, obwohl die Typdefinitionen nicht mehr vorhanden sind und damit die Leerzeichen / -zeilen nicht benötigt werden). Diese überflüssigen Zeichen können jedoch zuverlässig mit Prettier entfernt werden, solange der ursprüngliche Code den Formatierungsregeln entspricht, mit denen Prettier aufgerufen wird.

Auf diese Art und Weise ist es also möglich, Dateivorlagen in TypeScript zu schreiben und zu pflegen, die bei der Projektinstallation entweder direkt kopiert oder zunächst zu JavaScript umformatiert und dann in das neue Projekt eingefügt werden. Durch die Verwendung des tatsächlichen TypeScript-Kompilers kann garantiert werden, dass die auf diese Art erzeugten Dateien dieselbe Funktionalität bieten.

4.4.3 Framework-spezifische Codegeneration

Neben solchem Allgemeinen JavaScript- / TypeScript-Code muss für einige Erweiterungen auch Framework-spezifischer Code erzeugt werden. Dies ist vor allem dann der Fall, wenn Komponenten oder CSS-Bibliotheken zum Projekt hinzugefügt werden sollen.

Die Anforderungen, die hierfür existieren, sind zwischen den Frameworks teilweise unterschiedlich. In allen Frameworks gibt es jedoch eine Datei, in der die oberste Stufe der DOM-Struktur erzeugt wird (in Angular ist dies die `src/app/app.component.html`, in React ist es die `src/App.tsx` und in Vue.js ist es die `src/App.vue`), in die neue Komponenten einzufügen sind.

Dieses Einfügen neuer Komponenten soll immer an derselben Stelle erfolgen, sodass sich dort alle neuen, durch Erweiterungen hinzugefügten Komponenten nacheinander ansammeln. Eine fachlich orientierte Lösung wäre also in der Lage, den aus den genannten Dateien resultierenden DOM-Baum in ein JavaScript-Objekt zu überführen, dieses um die gewünschte Komponente zu ergänzen und dann den Objekt entsprechendem Code in die Datei zu schreiben. Die so vorgenommene Änderung sollte minimal und korrekt formatiert sein.

Alternativ dazu wäre es, da diese entsprechenden Dateien bei jeder Installation gleich sind, möglich, über reine Textsuche die Stelle zu finden, in der die neue Komponente einzusetzen ist. Dort würde der entsprechende Code eingefügt werden und anschließend

würde mithilfe von Prettier die korrekte Formatierung der Datei gewährleistet werden.

Aufgrund der verschiedenen Syntaxen der drei Frameworks müsste beschriebene fachliche Lösung für jedes Framework einzeln entwickelt werden, was nur mit hohem Aufwand umsetzbar wäre. Der Text-basierte Ansatz wäre sehr fragil, da bereits geringfügige Änderungen der durch Frameworks erzeugten Dateien dazu führen könnten, dass neue Komponenten nicht mehr eingefügt werden können.

Da aber auch der fachliche Ansatz von der genauen DOM-Struktur abhängt, die erzeugt werden würde, ist auch er für solche Fehler anfällig. Aufgrund der deutlich geringeren Komplexität wurde deshalb der Text-basierte Ansatz bevorzugt und für alle Frameworks implementiert.

Neben dem Einfügen von Komponenten gibt es noch weitere Änderungen, die vornehmbar sein müssen. In einem Angular-Projekt muss es möglich sein, bestimmte Arrays in einer TypeScript-Datei zu modifizieren. In React-Projekten hingegen muss es möglich sein, neue DOM-Elemente derart hinzuzufügen, dass sie andere (bereits existierende) Elemente umgeben. Beide Probleme wurden ebenfalls aus den bereits beschriebenen Gründen mit text-basierten Ansätzen gelöst.

4.5 Probleme

Während der Entwicklung sind einige Probleme aufgetreten, mit denen bei der Konzeptionierung nicht gerechnet worden ist. Die folgenden Unterkapitel beschreiben diese Probleme und wie sie gelöst worden sind.

4.5.1 Falsche Typdefinitionen von inquirer

Wie bereits beschrieben worden ist, wurde für die Entwicklung von GWA TypeScript verwendet. Da aber einige der verwendeten Bibliotheken, darunter inquirer **ZITAT FEHLT** (einfach package.json, Abwesenheit von typescript), nicht in TypeScript entwickelt wurden und auch keine eigenen von Hand erzeugten Typdefinitionen beinhalten, müssen diese Typdefinitionen aus dritten Quellen bezogen werden.

Dieses Problem ist weit verbreitet und kann häufig mittels des DefinitelyTyped-Projektes gelöst werden **ZITAT FEHLT** (vlt etwas darüber, dass NPM DT-Icon anzeigt). Dies ist ein großes Projekt, was für derartige NPM-Pakete Typdefinitionen sammelt und als gesonderte NPM-Pakete veröffentlicht **ZITAT FEHLT** (einfach README).

TS2345: Argument of type

'Observable<DistinctQuestion<Answers>>' is not assignable to parameter of type 'QuestionCollection<Answers>'.

Type 'Observable<DistinctQuestion<Answers>>' is missing the following properties from type

'Observable<DistinctQuestion<Answers>>':

```
_isScalar , _trySubscribe , _subscribe
```

Listing 2: TypeScript-Fehlermeldung bei der Verwendung von `inquirer` mit einem `RxJS-Observable`

Zu Beginn der Entwicklung ist der in Listing 2 aufgeführte TypeScript-Fehler entstanden, der sich nicht durch Fehler im selbstgeschriebenen Code erklären lies. Besonders ungewöhnlich an dem Fehler ist, dass zwei identisch benannte Typen (woraus in der Regel folgt, dass die Typen auch tatsächlich identisch sind) zueinander inkompatibel sind.

Bei Nachforschungen in den Abhängigkeiten, aus denen die beiden betroffenen Werte stammten, stellte sich heraus, dass zwei Verschiedene Versionen von `RxJS` verwendet wurden, deren Typdefinitionen zueinander inkompatibel waren. Konkret hatten die aus `DefinitelyTyped` stammenden Typdefinitionen von `inquirer` eine Abhängigkeit auf eine veraltete Version von `RxJS` deklariert, während `GWA` von der aktuellen Version abhängig war.

Dieser Fehler lässt sich auf zwei Wege lösen. Durch die einfache Löschung der lokalen Installation der veralteten Version von `RxJS` wird automatisch in den Typdefinitionen von `inquirer` die (weiterhin installierte) aktuelle Version von `RxJS` verwendet. Diese Lösung hat allerdings den Nachteil, dass die Löschung der veralteten Version nach jeder Installation oder Verlinkung von `GWA` mit `NPM` erneut vorgenommen werden muss, da sie im Rahmen der entsprechenden Befehle erneut installiert wird.

Alternativ kann die Abhängigkeit der Typdefinitionen von `inquirer` aktualisiert werden. Hierfür wurde in dem `DefinitelyTyped`-Repository ein Pullrequest gestellt¹⁷, der aber aufgrund mehrerer anderer Abhängigkeiten auf diese Typdefinitionen besondere Überprüfung erforderte.

Während diese Überprüfung stattfand, konnte das Problem mit dem ersten Lösungsansatz umgangen werden. Seit der am 6. September 2021 veröffentlichten Version 8.1.0 der Typdefinitionen von `inquirer` ist das Problem jedoch gänzlich gelöst.

4.5.2 Umgebungsbasierte Probleme

Im Rahmen der meisten Installationsprozesse müssen verschiedene `npm`-Pakete als Befehle ausgeführt werden. Beispielsweise wird zum Installieren von `React` der Befehl `npx create-react-app <Projektname>` ausgeführt. Hierfür wird die `Node.js`-interne Bibliothek `child_process` verwendet. Diese ermöglicht das Erzeugen und überwachen von neuen Prozessen.

In `Unix`-basierten Betriebssystemen wie `Linux` oder `macOS` lässt sich mit dieser Bibliothek problemlos der beispielhaft genannte Befehl `npx create-react-app <Projektname>` ausführen. Auf einem `Windows`-Computer hingegen führt der Aufruf zu einem Fehler.

Das durch diesen Befehl aufgerufene Programm `npx` kann in `Unix`-basierten Betriebssystemen direkt in einem neuen Prozess gestartet werden. Es liegt in Form einer `JavaScript`-

¹⁷<https://github.com/DefinitelyTyped/DefinitelyTyped/pull/54885>

Datei vor, an deren Beginn deklariert wird, dass diese von Node.js ausgeführt werden muss. Ist Node.js installiert, so geschieht dies automatisch.

Auf Windows-Computern wird `npx` jedoch als `.cmd`-Datei installiert. Diese sind in Windows nicht direkt in einem Prozess ausführbar, sondern können nur von der Kommandozeile (der CMD) ausgeführt werden [14]. Es muss also zunächst in einem neuen Prozess die CMD aufgerufen werden, damit diese dann `npx` ausführen kann.

Aufgrund dessen, dass die Entwicklung ausschließlich in einer Unix-basierten Umgebung¹⁸ stattfand, ist dieser Fehler nur aufgefallen, als Dritte GWA auf eigenen Computern getestet haben. Daraufhin wurde darauf geachtet, GWA sowohl in Unix-basierten Umgebungen als auch auf einem Windows-System ausgiebig zu testen, wobei weitere leicht zu lösende Probleme (wie die Verwendung falscher Trennzeichen für Ordnerstrukturen) aufgefallen sind.

Durch die Verwendung anderer Werkzeuge (wie der Framework-spezifischen CLIs) zur Installation bestimmter Bibliotheken kann es geschehen, dass auf einem Computer vorinstallierte Versionen dieser Werkzeuge einen Einfluss auf die Ausführung von GWA haben. Solche Einflüsse müssen ausgeschlossen werden, um ein einheitliches Verhalten des Programms gewährleisten zu können.

Außerdem muss natürlich sichergestellt werden, dass diese Abhängigkeiten auch dann verwendbar sind, wenn Nutzende sie nicht zuvor installiert haben. Da viele dieser Werkzeuge im Rahmen der Konzeptionierung installiert werden mussten, fällt es im Rahmen der Entwicklung nicht auf, wenn diese Installation nicht stattfindet sondern die Abhängigkeit fälschlicherweise als bereits installiert vorausgesetzt wird.

Um derartige Probleme auch während der Entwicklung feststellen zu können, lässt sich eine Virtuelle Maschine (VM) erzeugen, auf der GWA ausgeführt wird. Zum Erzeugen solcher VMs wird Docker verwendet, da so mit einer Konfigurationsdatei festgelegt werden kann, welche Abhängigkeiten vor der Ausführung von GWA bereits installiert sein sollen und welche nicht. Dieser Ansatz wird im Rahmen der Evaluierung (Kapitel 5) ausführlicher erläutert, aber bereits während der Entwicklung konnten auf diese Weise fehlende Abhängigkeitsinstallationen entdeckt werden.

4.6 Vorgehen nach TDD

Da ermöglicht werden soll, dass andere Entwickelnde an GWA mitarbeiten können sollen, ist es hilfreich, möglichst viel Funktionalität durch automatisierte Tests abzudecken. So können an beliebigen Stellen Änderungen vorgenommen werden und auch, wenn Entwickelnde keinen Überblick über den vollständigen Funktionsumfang haben, können sie durch Ausführung der Tests mit großer Zuversicht feststellen, ob sie bereits existierende Funktionalität versehentlich verändert oder ausgebaut haben.

¹⁸Es wurde zwar ein Windows-Computer genutzt, auf dem aber Windows Subsystem for Linux (<https://docs.microsoft.com/en-us/windows/wsl/about>) installiert war. Die Entwicklung hat ausschließlich in einer damit eingerichteten Virtuellen Maschine mit Ubuntu stattgefunden.

Um eine solche hohe Testabdeckung zu erreichen, wurde zu großen Teilen nach TDD vorgegangen (siehe Kapitel 2.5). Insgesamt wurden 174 Tests geschrieben, die 89,18% des Codes abdecken.

Wie diese Abdeckungsrate zeigt, wurde an einigen Stellen von TDD abgewichen. Dies ist insbesondere an Stellen der Installation geschehen, wo andere Werkzeuge aufgerufen werden um Dateien zu erzeugen oder zu modifizieren. Zwar können diese Aufrufe abgefangen werden, aber ohne die tatsächliche Durchführung dieser Abhängigkeiten ist nicht überprüfbar, ob die Aufrufe tatsächlich den gewünschten Effekt haben würden.

Im Rahmen von Ende-zu-Ende-Tests wäre es möglich, diese Abhängigkeiten auszuführen und am Ende einer kompletten Installation zu prüfen, ob diese wie gewünscht abgelaufen ist. Da Ende-zu-Ende-Tests im Kontext von GWA jedoch Änderungen am Dateisystem vornehmen würden, wurde zunächst auf sie verzichtet. Ihre Ausführung wäre im Rahmen von VMs problemlos möglich; jedoch hätte die Implementierung solcher Tests eine für den Rahmen dieser Arbeit zu hohe Komplexität gehabt. Auf die Ergänzung solcher Tests wird im Rahmen der weiteren Forschungsmöglichkeiten in Kapitel 6 eingegangen.

5 Evaluierung

Die Feststellung des Erfolgs der Implementierung erfolgt in zwei Schritten. Zunächst wird in Bezug auf die bei der Konzeptionierung erarbeiteten Features betrachtet, ob alle gewünschten Funktionalitäten implementiert wurden. Daraufhin ist anhand bestimmter Kriterien zu überprüfen, ob die implementierten Features funktionieren.

5.1 Evaluierung der umgesetzten Features

In Kapitel 3.3 ist eine vollständige Liste aller gewünschter Features erfasst worden. Diese umfasst vor allem Features, die für Nutzende von Interesse sind, erwähnt aber auch Aspekte der Erweiterbarkeit und Vorbereitung auf zukünftige Entwicklung. Im folgenden wird untersucht, ob all diese Ziele erreicht werden konnten.

Das erste erwähnte Feature ist die interaktive und informative Auswahl von Bibliotheken und Werkzeugen. Es konnte ein Fragebogen umgesetzt werden, der genau eine solche Auswahl ermöglicht. Vor und nach dieser Selektion sind weitere Fragen möglich, wobei bereits über CLI-Argumente beantwortete Fragen nicht erneut gestellt werden, sodass die Konfiguration standardmäßig komplett interaktiv ist, aber auf Wunsch auch ohne jegliche Interaktion stattfinden kann.

Aufgrund der Typdefinition von Erweiterungen ist es unmöglich, eine Erweiterung zu GWA hinzuzufügen, die nicht über grundlegende Metadaten verfügt, worunter insbesondere der Name der Erweiterung, ihre Beschreibung sowie ein Link zu weiteren Informationen zählen. Somit sind die Ziele der Interaktivität und der Informativität als erfüllt zu betrachten.

Bei der Umsetzung der Fragen war außerdem darauf zu achten, dass keine unzulässige Konfiguration auswählbar ist. Dieses Feature konnte umgesetzt werden, indem nach jeder Frage die gegebene Antwort überprüft und ggf. zurückgewiesen wird. Nutzende haben im Fall einer unzulässigen Antwort dann weiterhin die Möglichkeit, ihre Antwort zu bearbeiten.

Es wäre schöner gewesen, Nutzenden bereits während der Beantwortung jeder Frage Rückmeldung zu der Gültigkeit der aktuell geleisteten Antwort zu geben. Da die für den Dialog verwendete Bibliothek jedoch kein solches Feature anbietet, wird stattdessen bei jeder Frage und ggf. neben jeder Antwortmöglichkeit angegeben, welche Einschränkungen gelten. Nutzende können basierend auf diesen Informationen versuchen, unzulässige Antworten zu vermeiden, werden aber zusätzlich beim Versuch der Einreichung jeder Antwort daran gehindert, fehlerhafte Antworten zu geben.

Diese Validierung findet auch statt, falls bestimmte Antworten schon im Rahmen der CLI-Argumente gegeben werden. Dabei fehlschlagende Validierungen führen jedoch zu einem fehlerhaften Beenden von GWA, um eine automatische Benutzung zu ermöglichen. Diese Entscheidung hat sich bereits bei der Erstellung automatischer Tests als hilfreich erwiesen. Auch das gewünschte Feature der Antwortvalidierung wurde also erfüllt.

Ebenfalls umgesetzt werden konnten die Auswählbarkeit des Paketmanagers sowie die Installation über Framework-spezifische CLIs wo sie möglich ist. Damit kann, wie bereits bei der Konzeptionierung erläutert, die zukünftige Pflegbarkeit der erzeugten Projekte erleichtert werden, da die für das jeweilige Framework empfohlenen Werkzeuge genutzt werden können.

Die Umsetzung von Erweiterungen konnte nur etwas eingeschränkter geschehen. Zwar wurden mehrere verschiedene Kategorien von Erweiterungen umgesetzt, darunter Werkzeuge wie Frameworks und CSS-Präprozessoren, aber auch Bibliotheken wie RxJS. Allerdings ist darauf verzichtet worden, Erweiterungen für Werkzeuge und Bibliotheken zum automatischen Testen zu schreiben.

Diese Erweiterungen hätten sich von den implementierten Erweiterungen deutlich unterscheiden, da sie einen großen Einfluss auf andere Erweiterungen gehabt hätten. Bei ihrer Installation hätten nämlich nicht nur die entsprechenden Werkzeuge / Bibliotheken installiert und eingebunden werden müssen, sondern es hätten sämtliche bisher existierenden Tests auf die ausgewählte Erweiterung angepasst werden müssen. Darüber hinaus hätten auch, sofern sinnvoll, von allen anderen Erweiterungen Tests basierend auf den ausgewählten Werkzeugen und Bibliotheken erzeugt werden müssen. Beispielsweise müsste demnach die Redux-Erweiterung für die erzeugten Komponenten ebenfalls Tests generieren.

Aufgrund der hohen Komplexität dieses Features und dem begrenzten, im Rahmen dieser Arbeit leistbaren Arbeitsumfangs wurde vorerst darauf verzichtet, Erweiterungen für automatische Tests umzusetzen. Diese Erweiterungen sind jedoch für die weitere Entwicklung von GWA geplant und werden daher im Kapitel 6 erneut thematisiert werden.

Obwohl dies nicht explizit als Feature gewünscht wurde, wurde außerdem bei der Ent-

wicklung sehr darauf geachtet, dass der Code von Dritten erweiterbar und veränderbar ist. Dies ist vor allem durch automatische Tests gewährleistet worden.

5.2 Sicherstellung der Funktionalität

Die Überprüfung der Korrektheit einiger Features, insbesondere der Installation von Erweiterungen, ist nur mit hohem Aufwand ausführlich möglich. Für jede Erweiterung müsste betrachtet werden, welche Installationsschritte sie durchführen soll. Häufige Schritte wie die Installation eines NPM-Pakets sind leicht zu überprüfen, aber bei außergewöhnlicheren Schritten wie der Ergänzung einer Komponente im Falle der Redux-Erweiterung müsste die erzeugte Applikation lokal gestartet werden und es müsste von Hand verifiziert werden, dass die Komponente eingefügt worden ist und über ihre erwartete Funktionalität verfügt.

Außerdem werden einige Features durch die Auswahl bzw. die Nicht-Auswahl anderer Erweiterungen aktiviert bzw. deaktiviert. Es müssen zur vollständigen Überprüfung solcher Erweiterungen also mehrere Projekte eingerichtet und überprüft werden.

Aufgrund dieses hohen Umfangs wären manuelle Tests mit einem großen zeitlichen Aufwand verbunden. Außerdem wäre die Durchführung der Tests an vielen Stellen gleich und daher anfällig für Flüchtigkeitsfehler.

Gleichzeitig ist die Automatisierung dieser Tests aufgrund ihrer Vielfältigkeit nur schwer möglich. Für viele Erweiterungen müssten spezielle Tests angefertigt werden und die Überprüfung bestimmter Features (wie die Funktionalität der durch die Redux-Erweiterung eingebundene Komponente) ließe sich am besten über Ende-zu-Ende-Tests mithilfe entsprechender Werkzeuge realisieren. Genau solche Tests sollen künftig über Erweiterungen installierbar sein, aber dann müsste man diese Tests zu diesen Evaluierungszwecken auch dann in Projekte einbauen können, wenn sie eigentlich gar nicht Teil der zu testenden Konfiguration sein sollen.

Während der manuellen Tests, die im Rahmen der Entwicklung durchgeführt wurden, hat sich jedoch gezeigt, dass sich die meisten solcher detaillierter Fehler gut durch Unit-Tests abdecken ließen. Die danach übrig gebliebenen Fehler waren ausnahmslos derart schwerwiegend, dass entweder der Installationsprozess oder das Bauen der resultierenden Applikation fehlgeschlagen hat.

Auf diese Beobachtung hin wurde eine vereinfachte Prüfung von Installationen erarbeitet, die sich leicht automatisieren lässt. Eine zu prüfende Konfiguration wird an GWA weiter gegeben. Im Anschluss an eine erfolgreiche Installation wird das Projekt gebaut und sofern diese beiden Prozesse fehlerfrei laufen gilt das Projekt als erfolgreich eingerichtet.

Mithilfe von Docker können beliebige Konfigurationen innerhalb einer VM durchgeführt werden. Ein TypeScript-Skript orchestriert die Erzeugung und Löschung der VMs sowie die Ausführung der Installations- und Baubefehle innerhalb der VMs. Im Anschluss an die automatisch durchgeführten Befehle wird jede VM gespeichert, damit die getroffene Installation für weitere manuelle Tests zur Verfügung steht.

In diesem Skript kann festgelegt werden, welche Konfigurationen zu überprüfen sind. Da während der Entwicklung viele Probleme durch die Kombination bestimmter Erweiterungen entstanden sind, werden bei diesen Tests alle Erweiterungen in möglichst umfangreichen Projekten getestet. Unter Beachtung von Abhängigkeiten und Exklusivitäten werden also maximal viele Erweiterungen zusammen installiert.

Zum Zeitpunkt des Schreibens dieser Arbeit liegen vor allem¹⁹ zwei Arten von Exklusivitäten vor: die zwischen Frontend-Frameworks und die zwischen CSS-Präprozessoren. Um trotz dieser Exklusivitäten alle möglichen Konstellationen überprüfen zu können, wird jede (zulässige) Kombination eines Frameworks und eines CSS-Präprozessors überprüft.

Eine weitere große Fehlerquelle war die Installation von Erweiterungen mit oder ohne TypeScript, da abhängig von TypeScript weitere NPM-Pakete zu installieren waren, besonderer Code generiert werden musste oder ähnliches. Eine Ausnahme bilden hierbei jedoch die CSS-Präprozessoren. Daher wird mit ihrer Ausnahme jede Erweiterung sowohl mit, als auch ohne TypeScript überprüft.

Aus diesen Gründen werden insgesamt acht Konstellationen von Erweiterungen überprüft: Angular wird in Kombination mit jedem der vier CSS-Präprozessoren (hier wird die Abwesenheit eines CSS-Präprozessors auch als Präprozessor gezählt) und allen sonstigen Erweiterungen installiert. Da Angular von TypeScript abhängt, werden diese Konfigurationen nie ohne TypeScript überprüft. Außerdem wird React mit jedem CSS-Präprozessor außer Less und allen übrigen Erweiterungen überprüft. Dazu kommt lediglich eine Konfiguration ohne TypeScript, da die Anwesenheit von TypeScript keinen Einfluss auf CSS-Präprozessoren hat.

Durch das Durchführen dieser Tests sind mehrere Fehler aufgefallen, die vor allem in der Kombination von CSS-Präprozessoren mit React aufgetreten sind und die vollständig behoben werden konnten. Bei der weiteren manuellen Überprüfung der dabei erstellten Projekte sind keine weiteren Fehler aufgefallen. Abschließend sind also keine Fehler bei bisher implementierten Features bekannt.

6 Ausblick

Wie bereits an einigen Stellen erwähnt wurde, gibt es viele Aspekte, auf die sich weitere Forschung konzentrieren könnte. In keiner besonderen Reihenfolge werden hier einige dieser Aspekte aufgelistet.

Da im Rahmen dieser Arbeit die Entwicklung von Erweiterungen lediglich exemplarisch erfolgt ist, wäre es sinnvoll, weitere Erweiterungen zu ergänzen. Insbesondere fehlt Unterstützung für Vue.js. Da das Vue-CLI jedoch große Ähnlichkeiten zum Angular-CLI aufweist, müssen für die Vue-Erweiterung keine neuen Konzepte erarbeitet werden.

¹⁹Außerdem ist Less nicht zusammen mit React verwendbar. Diese Kombination wird im späteren Verlauf einfach ausgelassen.

Des weiteren fehlen Erweiterungen für weitere Werkzeuge und Bibliotheken. Größtenteils sind auch hier keine neuen Konzepte erforderlich, sodass sich an bereits implementierten Erweiterungen orientiert werden kann. Ausnahme hiervon sind jedoch, wie bereits im vorigen Kapitel beschrieben, die Erweiterungen für Werkzeuge und Bibliotheken zum Schreiben von automatisierten Tests.

Hier könnte beispielsweise eine Abstraktion über gängige Werkzeuge und Bibliotheken entwickelt werden, sodass andere Erweiterungen an einer Stelle Tests definieren können, die mittels der Abstraktion nur auf gemeinsame Features aufbauen. Aus diesen abstrahierten Definitionen könnten dann für jede Testbibliothek entsprechende Dateien erzeugt werden.

Auch im Bereich der automatisierten Tests für GWA ist weitere Arbeit zu leisten. Die Aussagekraft der Ende-zu-Ende-Tests ist bisher nur gering, da lediglich die Baubarkeit (d.h. der Erfolg des `npm run build`-Befehls) bestimmter Konfigurationen überprüft wird. Durch eine Einbindung inhaltlicher Tests für alle verwendeten Erweiterungen könnte diese Aussagekraft vergrößert werden. Außerdem könnte untersucht werden, ob die Tests parallelisiert ablaufen können, um die aktuell bei ca. 20 Minuten liegende Laufzeit zu verkürzen. Da ein großer Teil dieser Laufzeit von der Installation der NPM-Pakete kommt, könnte ein weiterer Ansatz zur Laufzeitverbesserung auch das Zwischenspeichern dieser Pakete sein. Eine weitere Verbesserungsmöglichkeit bietet die Erzeugung der Liste der zu überprüfenden Konfigurationen. Aktuell muss diese manuell gepflegt werden; es wäre jedoch sinnvoll, diese basierend auf der Menge aller verfügbaren Erweiterungen automatisch zu erzeugen. So müsste diese Liste nicht bei jedem Hinzufügen einer neuen Erweiterung oder bei jeder Modifikation der stellbaren Fragen modifiziert werden.

Im Rahmen der Erweiterungen besteht auch an allgemeinen Stellen Verbesserungsbedarf. Zum einen wäre es hilfreich, wenn Nutzenden nicht erst beim Versuch der Absendung einer Antwort auf eine Frage Feedback zur Validität der Antwort gegeben würde, sondern dies schon beim Ausfüllen der Antwort geschehen könnte. Zum anderen ist die manuelle Festlegung der Installationsreihenfolge der Erweiterungen suboptimal, da bei ihrer Erzeugung schwer zu findende Fehler unterlaufen können, die nur bei sehr bestimmten Kombinationen von Erweiterungen auftreten. Sinnvoller wäre hier eine automatische Generation basierend auf Restriktionen, die pro Erweiterung festgelegt werden können. Wie bereits erwähnt, müsste hierfür vorsichtig ein System von Restriktionen ausgearbeitet werden.

Die momentan sehr auf die Installation bestimmter Bibliotheken oder Werkzeuge konzentrierten Fragen könnten zukünftig um weitere Konfigurationsfragen erweitert werden. So wäre es möglich, beispielsweise ESLint von Anfang entsprechend Präferenzen des / der Nutzenden zu konfigurieren, sodass diese Konfiguration nicht noch nach der Installation vorgenommen werden muss.

Insbesondere, wenn derartig umfangreiche Konfigurationen getroffen werden können, aber auch schon im Rahmen der reinen Installation von Projekten wäre es für die wiederholte Nutzung von GWA hilfreich, wenn nach dem Beantworten aller Fragen die Antworten als „Preset“ gespeichert werden könnten, sodass bei späteren Aufrufen zuvor

erstellte Presets verwendet werden können. Dies würde die wiederholte Verwendung einer Lieblingskonfiguration erleichtern. Dabei sollte jedoch auch darauf geachtet werden, dass vor der Installation noch Anpassungen an das gewählte Preset vorgenommen werden können, sodass besondere Eigenheiten des zu erzeugenden Projektes berücksichtigt werden können.

Zukünftige Forschung könnte sich ebenso auf die weitere Verallgemeinerung von GWA konzentrieren. Eine mögliche Verallgemeinerung wäre die Entfernung der Notwendigkeit eines Frameworks. So könnten beispielsweise frameworklose Webprojekte oder auch Node.js-Projekte erzeugt werden. Außerdem lässt sich evaluieren, ob eine Erweiterung von GWA auf andere Programmiersprachen sinnvoll umsetzbar ist. Gerade im von konkreten Erweiterungen unabhängigen Teil des Codes müssten hierfür weitere Abstraktionen getroffen werden, aber erste Grundlagen für eine solche Entwicklung sind beispielsweise durch die Abstraktion des Paketmanagers bereits geschaffen worden.

7 Fazit

Im Rahmen dieser Arbeit sollte ein CLI namens „generate-web-app“ (GWA) entwickelt werden, das Programmierenden die initiale Einrichtung von Webprojekten abnimmt. Der Fokus sollte Anfängerfreundlichkeit und Interaktivität liegen.

Im Rahmen der Planung dieses CLIs wurden die Installationsvorgänge verschiedener Bibliotheken und Werkzeuge miteinander verglichen, sodass Gemeinsamkeiten und Unterschiede hervorgehoben werden konnten. Außerdem wurde analysiert, welche Möglichkeiten zur Projektinitialisierung bereits existieren. Diese wurden ebenfalls miteinander verglichen und insbesondere auf ihre Stärken und Schwächen hin untersucht.

Aus diesen Untersuchungen ergab sich der Plan, ein auf Erweiterungen basierendes System zu entwickeln, das zunächst allgemeine Fragen stellt und erfragt, welche Bibliotheken / Werkzeuge gewünscht sind. Zu dieser Auswahl werden dann weitere Fragen gestellt, bevor die Installation gemäß der angegebenen Wünsche erfolgt. Außerdem sollte eine Validierung der Antworten eingebaut werden.

Dieses System konnte wie geplant implementiert werden. Dabei wurden neben allgemeinen Techniken wie der asymptotischen Laufzeitanalyse Ansätze aus der funktionalen Programmierung und Konzepte aus der Graphentheorie verwendet. Die Implementierung erfolgte mehrheitlich testgetrieben in TypeScript.

Neben der Implementierung dieses Grundgerüsts wurden einige Erweiterungen entwickelt, um die Funktionalität des Grundgerüsts und die Fähigkeiten des Erweiterungssystems zu demonstrieren. Die bereits entwickelten Erweiterungen ermöglichen die Installation der beiden Frontend-Frameworks React und Angular, der beiden Werkzeuge TypeScript und ESLint, der CSS-Präprozessoren SCSS, Sass und Less, sowie der Bibliothek Redux.

Durch die bei der testgetriebenen Entwicklung entstandenen Unittests und weitere automatisierte Ende-zu-Ende-Tests konnte die Funktionalität der bereits implementier-

ten Features kontinuierlich überprüft werden. In Kombination mit weiteren manuellen Tests konnten bei keinem der erstellbaren Projekte Fehler festgestellt werden.

Trotz der geringen Anzahl der bisher entwickelten Erweiterungen bietet GWA bereits eine Zeitersparnis bei der Installation von unterstützten Konfigurationsmöglichkeiten. Somit kann das Ziel dieser Arbeit als erreicht betrachtet werden, obgleich noch weitere Arbeit notwendig ist, um GWA zur Generierung mehr Projekte verwenden zu können.

Literatur

- [1] Archivierungsbanner und Angabe des letzten Commits auf den master-Branch am 25. März 2021. <https://github.com/palantir/tslint> [Zugriff am 29. August 2021].
- [2] Adi D., John W., Robert F. und Stephanie Y., Feb. 2019. <https://blog.palantir.com/tslint-in-2019-1a144c2317a9> [Zugriff am 29. August 2021].
- [3] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [4] James Henry, Apr. 2021. <https://github.com/angular-eslint/angular-eslint/issues/395#issuecomment-821991202> [Zugriff am 28. August 2021].
- [5] L. M. Kerr. Github-issue: Jest schematic + cypress schematic conflicts. <https://github.com/bribug/jest-schematic/issues/51> [Zugriff am 30. August 2021].
- [6] R. Mello, Apr. 2021. <https://github.com/ricmello/prettier-schematics/blob/main/src/collection.json> [Zugriff am 28. August 2021].
- [7] A. Nassri. So long, and thanks for all the packages! <https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages.html> [Zugriff am 14. September 2021].
- [8] Open Source. Verwendung von inquirer.js im angular CLI, . <https://github.com/angular/angular-cli/blob/master/package.json#L162> [Zugriff am 06. September 2021].
- [9] Open Source. Getting started - selecting a package manager, . <https://create-react-app.dev/docs/getting-started/#selecting-a-package-manager> [Zugriff am 31. August 2021].
- [10] Open Source. Verwendung von chalk in inquirer.js/packages/core/index.js, . <https://github.com/SBoudrias/Inquirer.js/blob/master/packages/core/index.js#L6> [Zugriff am 06. September 2021].
- [11] Open Source. Mapscii - the whole world in your console., . <https://github.com/rastapasta/mapscii> [Zugriff am 15. September 2021].
- [12] Open Source. Verwendung von inquirer.js im vue CLI, . <https://github.com/vuejs/vue-cli/blob/dev/package.json#L59> [Zugriff am 06. September 2021].
- [13] Open Source. State management - information for react developers, . <https://vuejs.org/v2/guide/state-management.html#Information-for-React-Developers> [Zugriff am 30. August 2021].

- [14] Open Source. Spawning .bat and .cmd files on windows, . https://nodejs.org/dist/latest-v6.x/docs/api/child_process.html#child_process_spawning_bat_and_cmd_files_on_windows [Zugriff am 10. September 2021].
- [15] Sacha Greif und Raphaël Benitte. State of JS 2020, 2020. <https://2020.stateofjs.com/en-US/> [Zugriff am 28. August 2021].
- [16] Stanimira Vlaeva und Kamil Kisiela, Aug. 2018. <https://github.com/kamilkisiela/apollo-angular/blob/master/packages/apollo-angular/schematics/collection.json> [Zugriff am 28. August 2021].
- [17] M. K. Tom Fellmann. A command line interface versus a graphical user interface in coding vr systems. In *Proceedings of the Second IASTED International Conference on Human Computer Interaction. ACTA Press*, 2007.
- [18] P. Verma. Gracoli: a graphical command line user interface. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 3143–3146. 2013.
- [19] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1): 11–12, 1962. ISSN 0004-5411. doi: 10.1145/321105.321107.